FIT3143: The *Unofficial* Course Notes

Mithun Hunsur

October 28, 2018

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

There are fundamental limitations to what can be done with serial/non-parallel computing. If you want to process a great deal of data simultaneously, a serial computer will only be able to handle a subset of the data at any given point.

A parallel computing system combines multiple computers/computing units so that they can be used to work on the same problem simultaneously. If a problem is "parallelisable", then it can be split up and executed across multiple machines.

1.1 Architectures

There are two types of computer architectures that are built on top of interconnected multiple processors:

- **Tightly coupled systems** have a single system-wide memory space shared by all the processors (that is, memory addresses for one processor are valid in another processor).

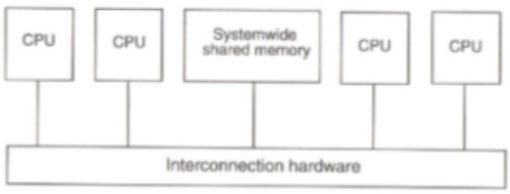


Figure 1.1: Tightly coupled system

- **Loosely coupled systems** do not share memory; each unit in the system has its own memory store that no other unit can access.

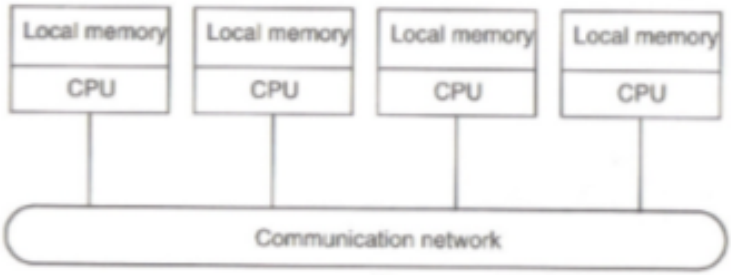


Figure 1.2: Loosely coupled system

1.2 Distributed Systems

The textbook definition of a distributed system:

> A distributed system is a collection of independent computers that appears to its users as a single coherent system.

In this definition, it is important that each computer is **independent** - that is, they can operate individually - and that the entire system appears as a **single coherent system** - users are not necessarily aware that the system is composed of multiple computing units.

Additionally, it is worth noting that components that are located on separate networked computers (e.g. Process A running on Computer 1, Process B running on Computer 2) communicate using message passing only - that is, they do not share memory.

The rise of distributed systems has been attributed to a variety of causes, including the prevalence of powerful microprocessors (especially conventional consumer processors, such as the Intel x86 line) and the availability of high-speed computer networking technology.

1.2.1 Models

A distributed computing system can be structured in a variety of different ways. These can be roughly categorised into one of the following five models.

1.2.1.1 Minicomputer

The minicomputer model is an extension of the centralised time-sharing systems of the 1970s. There are several minicomputers connected by a communication network, with each minicomputer having several users logged in simultaneously.

This model is useful when resources must be shared with remote users, and was used for early ARPAnet (the military/university precursor to the Internet).

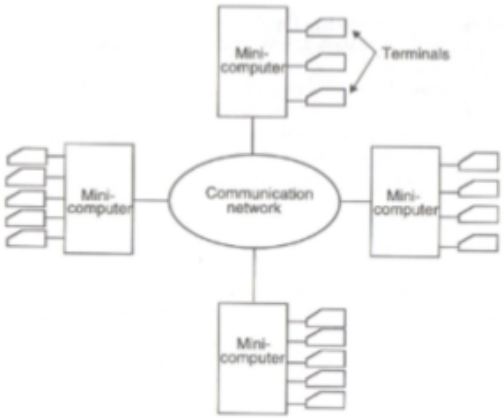


Figure 1.3: Minicomputer model

1.2.1.2 Workstation

The workstation model uses multiple workstations that communicate to each other using a communication network. These workstations often have spare computing power available, which is leveraged by the system as part of its operation.

Essentially, a user uses their workstation to submit a job to be run. These jobs are then distributed across the workstations on the network, allowing for work to be distributed and a more efficient allocation of resources than letting workstations idle.

Figure 1.4: Workstation model

However, there are several issues that must be resolved for efficient use of this model. These include:

- finding an idle workstation: the network must be able to allocate a workstation for execution of the job. If there is no such workstation available, the execution of the job may be delayed.

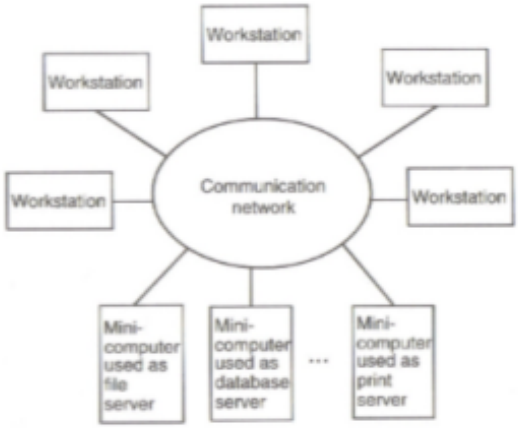


Figure 1.5: Workstation-server model

- transferring the job: when a workstation is available, the job can be transferred to it; but the actual method of transfer may be complicated, as the state of the job (i.e. the work completed to date) will also need to be transferred.
- control of a remote process: the system must be able to efficiently allocate and control processes on other workstations from the current workstation, which may result in complications

This model has been used for the Sprite system, as well as in Xerox PARC.

1.2.1.3 Workstation-server

The workstation-server model is a variant on the workstation model that adds several minicomputers to the system. These minicomputers provide services that all workstations may need at any given moment (and thus cannot be treated as a temporary job as would be the case with the workstation model). These services include file servers, database servers, and more.

There does not need to be a one-to-one allocation between a service and a minicomputer server; for example, multiple servers can provide a single service. This increases redundancy, which in turn increases reliability and provides better scalability.

An example of this model is the V-System.

1.2.1.4 Processor-pool

The processor-pool model is based around the observation that a user may have extremely varying computing demands, ranging from no computing power to a significant amount for a short period of time.

Given this, users are given terminals that have no independent computing power of their own. These terminals connect to the processor-pool, which is managed by a *run server*. This run

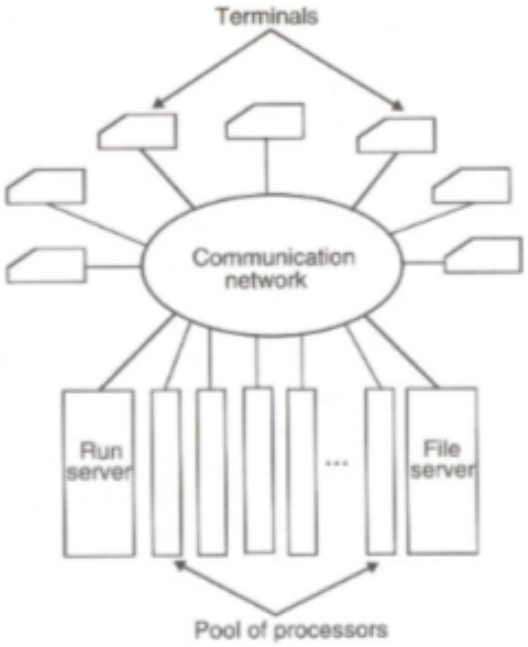


Figure 1.6: Processor-pool model

server allocates processors to users as required, allowing for efficient utilisation of the servers.

Users do not have a home machine; instead, they log onto the entire system and are allocated processing power as required.

Examples include Amoeba, Plan 9, and Cambridge DCS.

1.2.1.5 Hybrid

The hybrid model combines the workstation-server and processor-pool models with the aim of maximising the advantages of both. It augments the workstation-server model with a pool of processors.

This means that in a hybrid system, there are powerful workstations, servers and a pool of processors. These workstations can access resources on the servers, conduct work amongst themselves, or schedule high-load work on the processor pool.

As this requires the most resources out of all the models discussed, it is also the most expensive to implement.

1.2.2 Distribution Model

There are several models for how resources are distributed and/or made visible to nodes of a distributed system. Some of these include:

- **File Model**: resources are modelled as files on the file system, and are accessible through regular file APIs.
- **Remote Procedure Call Model**: resources are modelled as function calls, and can be accessed by calling the functions and retrieving their results.
- **Distributed Object Model**: resources are modelled as objects (a combination of data and functions relating to the data), and can be accessed by accessing the representation of the object.

1.2.3 Advantages

- **Economics**: Microprocessors provide a better performance/price ratio than mainframes.
- **Speed**: A distributed system can offer more computing power than a mainframe, especially as the number of the nodes in the system go up. A concrete example of this is splitting a database into many small databases, which can reduce the average response time by increasing the ratio of database servers to clients.

Figure 1.7: Single DB server vs multiple DB servers

- **Reliability**: The system can operate in a degraded state when it is partially unavailable; that is, a perfectly-redundant system with 5% of the system down would only experience 5% performance degradation.
- **Incremental growth**: The system can be incrementally grown by adding new nodes.
- **Sharing**: Many users can be granted access to common resources, such as a database and peripherals.
- **Communication**: Humans using the system can communicate to each other using the abstraction offered by the system.
- **Effective Resource Utilisation**: A distributed system can distribute any given workload over available machines in the most cost effective and efficient way possible.

1.2.4 Disadvantages

- **Software**: Distributed software is harder to develop than centralised software.
- **Networking**: The networking link cannot be assumed perfect; it may saturate (i.e. the network is completely used and cannot accept new connections) or experience other issues.
- **Security**: Data distributed across multiple nodes cannot be assumed to be secure, as the "easy access" nature of distributed data also applies to secure data.

1.2.5 Challenges

1.2.5.1 Heterogenity

A distributed system can feature variation in, but not limited to:

- **Networks**: The same protocol may not be used to communicate between nodes. Even if it is, the performance may vary within the network (especially with regards to network topology).
- **Computing hardware**: Different processors represent data differently (little vs big-endian, as an example).
- **Operating systems**: Message exchange may work differently between operating systems.
- **Programming languages**: Programming languages represent characters and data structures differently from each other.
- **Implementations by different developers**: Developers must have a common specification or standard; otherwise, their implementations of the system may be unable to co-operate.

1.2.5.2 Openness

New nodes and services can be added to a distributed system, but existing nodes may not know how to interface or interact with these new nodes. To alleviate this, specifications and/or interfaces for the new components should be published ahead of time.

1.2.5.3 Transparency

A distributed system, as defined, conceals its distributed nature by masquerading as a single computer. An example of this might be the World Wide Web, which largely hides its distributed nature.

In the context of distributed systems, the term *transparency* refers to hiding something [1].

Classifications of transparency:

- **Access transparency**: Data and resources can be used in a consistent way.
- **Location transparency**: A user cannot determine where resources are located.
- **Migration transparency**: Resources can move without changing the identifier used to access them.
- **Replication transparency**: A user cannot determine how many copies exist of a resource.
- **Concurrency transparency**: Multiple users can share resources automatically.
- **Failure transparency**: A user does not recognise resource failure.

[1] This is clearly wrong by any English dictionary - *opaqueness* is the right term to use - but someone made a mistake a long time ago and we have to live with it now.

- **Performance transparency**: Systems are reconfigured to improve performance as loads vary.
- **Scaling transparency**: Systems can expand in size without changing the structure of the system or the programs to run on the system.

1.2.5.4 Performance

An effective distributed system aims to produce high performance from a collection of cheap computers; this will be largely dependent on the workload and the distribution scheme.

There are two kinds of parallelism:

- **Fine-grained parallelism**: Small programs are executed in parallel. This results in a large number of messages; the communication overhead may result in a reduction in the performance gain from parallel processing.
- **Coarse-grained parallelism**: Large, compute-intensive programs are executed in parallel. As these programs are mostly independent, the communication overhead is limited compared to fine-grained parallelism.

The maximum aggregate performance of the system can be quantified in terms of the maximum aggregate floating-point operations per second, as shown below:

$$P = N \times C \times F \times R$$

where P is the performance in FLOPS (floating point operations per second), N is the number of nodes, C is the number of CPUs, F is the number of floating point operations per clock period, and R is the clock rate of each CPU.

Similar values can be calculated for MIPS (million instructions per second).

1.2.5.5 Scalability

A distributed system should naturally scale with increasing numbers of nodes. However, the scaling algorithm may break down at increasingly large scales of thousands of nodes or more.

Challenges may arise from having centralised the following:

- **Centralised components**: A single server (e.g. a mail server) for all users.
- **Centralised tables**: A central data store (e.g. a single online telephone book) for all nodes.
- **Centralised algorithms**: Algorithms dependent on information available only from the complete system (e.g. routing based on complete information).

To mitigate these, the following techniques can be used:

- No node in the system is fully aware of the state of the system.
- Nodes are only allowed to make decisions based on local state.
- Algorithms should be designed in such a way to avoid their invalidation upon the failure of a single node.

Scalability can be quantified using the following formula:

$$S = \frac{T_1}{T_N}$$

where T_1 is the wall-clock time for a single processor, and T_N is the wall-clock time over N processors.

A scalability figure close to N implies that the program scales well. This metric can be used to estimate the optimal number of processors for an application.

Additionally, the utilisation of the system can be calculated as such:

$$U = \frac{S(N)}{N}$$

Ideally, U would be close to 1 (or $U \times 100\%$ would be close to 100%).

1.2.5.6 Reliability

Due to the large number of components in a distributed system, the probability of *any* component failing is much higher than the probability of a non-distributed system failing. The system will need to handle this effectively.

The *availability* of a system (the fraction of time the system is available) can be calculated using the following equation:

$$R = \frac{\text{usable time}}{\text{total time}}$$

A *fault-tolerant* system can hide failures in individual components from the users; this is usually done by using another node to replace the failed node.

1.2.5.7 Security

As data is shared across the distributed system, it must be secured to ensure that it is not tampered with or viewed by unauthorized parties.

1.2.5.8 Concurrency

Jobs running simultaneously throughout the system should not interfere with each other, even when they are sharing resources.

Chapter 2

Inter-Process Communication

2.1 Introduction

Processes in a distributed system require the ability to communicate with each other; this is called Inter-Process Communication (IPC). There are two basic approaches underlying IPC (as can be seen in Figure 2.1):

- **Original sharing**: there is a shared memory area, and both processes access it.
- **Copy sharing**: messages are sent between the processes through some mechanism, and no memory is shared to facilitate this.

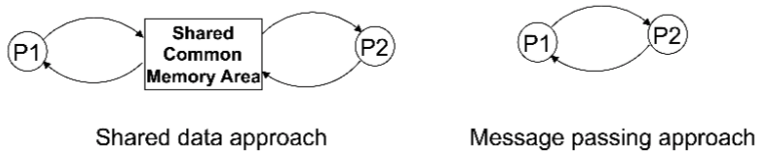


Figure 2.1: Comparison of approaches to IPC

A *message-passing system* is a component of a distributed system that provides a message-based IPC system for the rest of the system. In doing so, it abstracts away the details of the underlying network protocols and heterogeneous computing platforms.

Processes can communicate by exchanging messages; these messages are transferred using simple primitives such as `send` and `receive`.

Higher-level IPC systems, such as Remote Procedure Call (RPC) and Distributed Shared Memory (DSM), can be built on top of a message-passing system.

2.2 Design Constraints

A "good" message-passing system will generally have the following features:

- **Simplicity**
- **Uniform semantics**: The same primitives are used for both local and remote communication
- **Efficiency**: The system should aim to reduce the number of messages where possible, and make each message as efficient as possible. Ways to accomplish this include:
  - Avoiding the cost of establishing and terminating connections between a pair of processes for every new message between them
  - Minimising the cost of maintaining a connection
  - Bundling an acknowledgement of a previous message and a response together
- **Reliability**: The system should handle lost and duplicated messages reliably.
- **Correctness**: The system should obey the following properties:
  1. **Atomicity**
  2. **Ordered delivery**
  3. **Survivability**
- **Flexibility**: Not all systems require the full suite of correctness properties. A system should be able to disable one or more of these properties in exchange for better performance or improvement along some other metric.
- **Security**: The sender and receiver of a message should be authenticated, and there should be facilities for encrypting a message.
- **Portability**: The message-passing system should be portable (i.e. deployable across a wide variety of platforms); similarly, applications using IPC protocol primitives should also be portable.

The structure of a typical message in a message-passing system may look something like this:

- Header
  - Addresses
    - Sender address
    - Receiver address
  - Sequence number
  - Structural information
    - Type
    - Number of bytes
- Message

Issues that need to be considered while designing an IPC protocol include:

- **Identity**: Who is the sender? Who is the receiver?
- **Network Topology**: Is there one receiver for a message, or many? If so, how do you handle this?

- **Flow control**: Is acknowledgement of a message guaranteed by the receiver? Should the sender wait for acknowledgement?
- **Error control and channel management**: What happens if a node crashes? What happens if the receiver's not ready? How does a node deal with multiple outstanding messages?

2.3 Synchronisation

As processes run independently, they must synchronise in order to communicate. To do this, their send/receive primitives can operate in one of two modes: blocking, or non-blocking.

When using blocking semantics, the primitive will halt execution of the program until the operation has terminated. In the case of a send, this may mean that execution will be blocked until either the message has been sent or until it has been acknowledged by the receiving party. In the case of a receive, this means that execution will halt until a message is received.

When using non-blocking semantics, the program will attempt to complete the operation, but program execution will not be halted. The status of an operation can be checked to determine whether an operation has completed. This allows operations to be interleaved with work, which increases efficiency. A non-blocking send will queue up a message send with the operating system, and will then exit. A non-blocking receive will attempt to receive the data, but will do nothing if there is no message or if the data cannot be received yet.

When both sending and receiving of a communication between processes is blocking, the communication is *synchronous*; otherwise, it is *asynchronous*. A synchronous workflow can be seen in Figure 2.2.

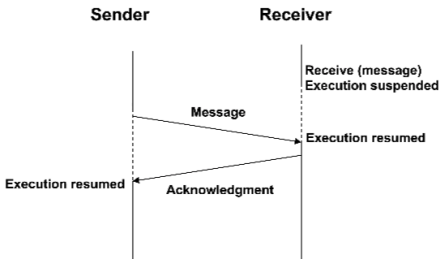


Figure 2.2: Synchronous workflow for send/receive

Advantages of synchronous communication include

- being simple and easy to implement
- increased reliability
- not requiring backward error recovery

Advantages of asynchronous communication include

- high concurrency
- being more flexible than synchronous communication

- having a lower deadlock risk than synchronous communication (but not impossible!)

A synchronous system uses no buffer for sending, and a single buffer for receiving. An asynchronous system uses an unbounded capacity buffer for both sending and receiving where the buffer can contain multiple messages. This can be seen in Figure 2.3.

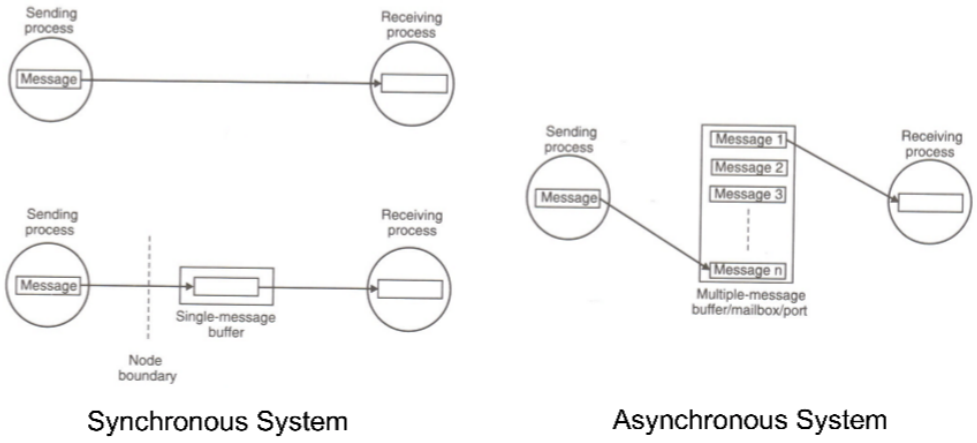


Figure 2.3: Buffering in a synchronous system versus an asynchronous system

2.4 Message Delivery Concerns

2.4.1 Multi-datagram messages

Almost all networks have an upper bound on the size of data that can be transferred in a single message; this quantity is known as the *MTU* (maximum transfer unit).

A message with a size greater than the MTU must be split apart (*fragmented*) into multiple packets, each of which is called a *datagram*.

Messages can thus be classified as a *single-datagram message* or a *multi-datagram message*. Disassembly and re-assembly of a multi-datagram message is the responsibility of a message-passing system.

A potential algorithm for re-assembly of a multi-datagram message is shown in Figure 2.4. It uses a *bitmap* to represent each datagram of the packet. When a datagram comes in, the corresponding bit in the bitmap is set. If the last datagram of a message arrives and the message is not complete, a request for the missing datagrams is sent to the sender. When all bits are set and the message can be reconstructed, an acknowledgement is sent to the sender.

2.4.2 Encoding/Decoding

Encoding and decoding are required if the sender and the receiver have a different computing architecture. However, it may still be required if the message to be sent uses an absolute pointer

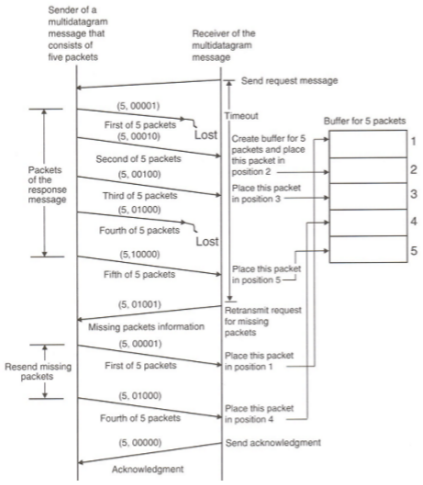


Figure 2.4: Bitmap algorithm for multi-datagram message reconstruction

to memory on the sender's machine, or if the message contains a user-defined object that the receiver does not necessarily know how to identify.

2.4.3 Process Addressing

When sending or receiving a message, explicit or implicit addressing can be used. Explicit addressing requires a specific process ID to send to or receive from, while implicit addressing requires the ID of a *service* (a group of processes belonging to a particular use-case [1]) to send to or receive from.

2.4.4 Failure Handling

There are three kinds of failure that can potentially occur in a message-passing system: the request could be lost, the response could be lost, or the request may have failed to execute on the receiver side. These are illustrated in Figure 2.5.

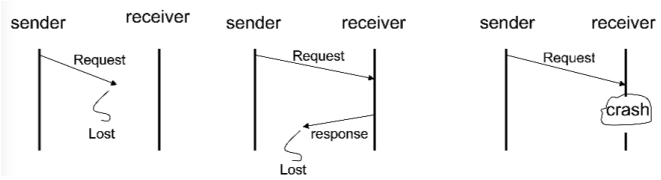
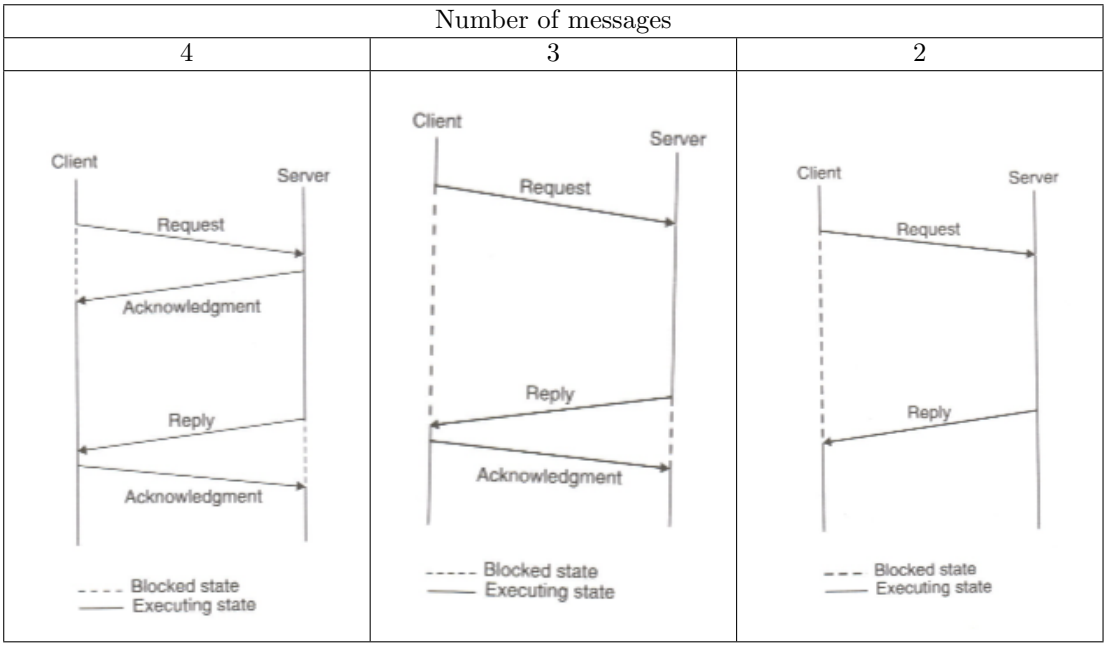


Figure 2.5: Potential failures in a message-passing system.

The following diagrams illustrate "reliable" IPC protocols to accommodate failure. However,

[1] I don't know if this is actually correct. I'm guessing based on my use of MPI, but the slides don't go into detail on this.

one side may end up blocked indefinitely [2] ; these protocols do not illustrate intelligent resend/waiting behaviour.



An example of a fault-tolerant system is shown in Figure 2.6. Note how the system deals with request loss, request execution failure, and loss of execution. The second execution of the request may result in different behaviour; this is covered in subsection 2.4.5.

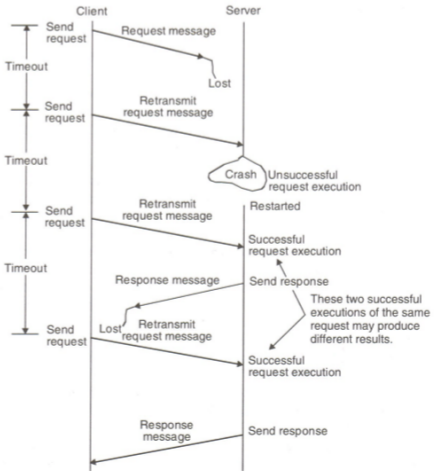


Figure 2.6: An example of a fault-tolerant system.

[2] Unless a timeout is used to terminate the blocking operation after some time. This is not mentioned in the slides, but it is the obvious way to make these at least *somewhat* realistic.

2.4.5 Idempotency

An idempotent function will return the same result given the same input, even when executed multiple times. For example, an idempotent function is illustrated in Algorithm 1; a non-idempotent function is illustrated in Algorithm 2.

Algorithm 1 An example of an idempotent function.

```
function GETSQRT(n)
    return SQRT(n)
end function
```

Algorithm 2 An example of a non-idempotent function. This function will return different values when given the same input. An illustration of the flow can be seen in Figure 2.7.

```
function DEBIT(amount)
    if balance > amount then
        balance ← balance − amount
        return ("success", balance)
    else
        return ("failure", balance)
    end if
end function
```

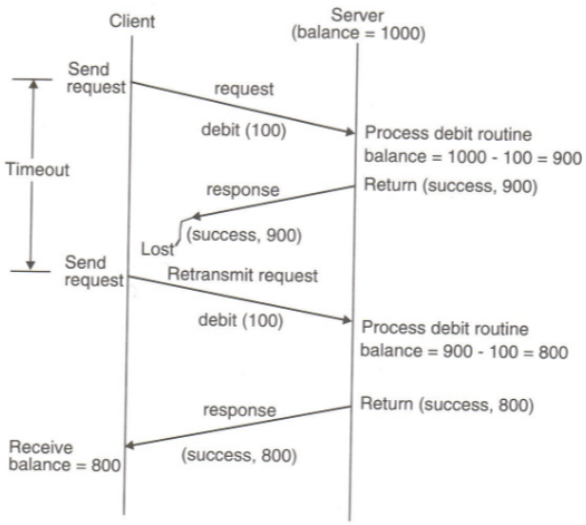


Figure 2.7: Non-idempotent function in a distributed system.

To mitigate the effects of non-idempotent functionality in a distributed system, a layer can be implemented on top of a non-idempotent function to make it appear idempotent. This can be done by adding a sequence number for the request message, and then adding a *reply cache* that stores response for a sequence number.

When a request is handled by a receiver, the response is stored in the reply cache and is indexed by the sequence number. If the request is re-received (i.e. as the sender failed to receive the

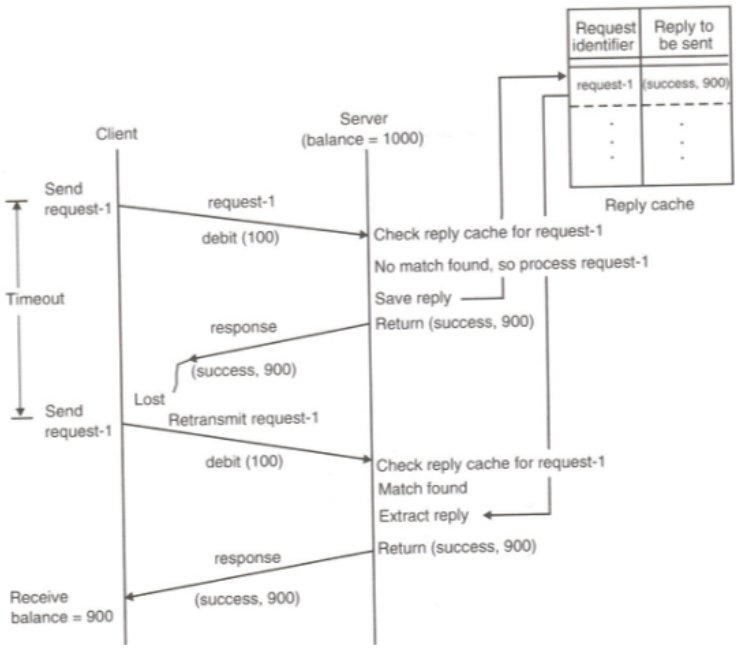


Figure 2.8: Conversion of a non-idempotent operation to an idempotent operation using a reply cache.

response), the reply cache will return the saved response instead of running the non-idempotent operation again. This is illustrated in Figure 2.8.

2.5 Group Communication

Group communication may involve one-to-many, many-to-one, and many-to-many communication.

In the case of one-to-many, the following concerns [3] may arise:

- Group management
- Group addressing
- Buffered and unbuffered multicast
- Send-to-all and Bulletin-Board semanntics
- Flexible reliability in multicast communication
- Atomic multicast

Many-to-many communication includes all of the issues involved in one-to-many and many-to-one, but adds a few of its own. A significant issue in many-to-many is *ordered message delivery*;

[3] I say concerns, but the slides are unclear on what this list *means*.

messages are no longer guaranteed to arrive in order, as they would in one-to-many or many-to-one.

A concrete example of this issue can be seen when considering two server processes maintaining a single salary database. Two client processes send an update for a salary record. Depending on when the messages arrive, the resulting behaviour may be different (i.e. the final version of the record may reflect update 1 *or* update 2).

2.5.1 Ordering Semantics

To resolve this, a variety of ordering semantics can be examined.

2.5.1.1 Absolute Ordering

All messages are delivered to all receiver processes in the exact order in which they were sent. The global timestamp is used as a message identifier with the sliding window protocol.

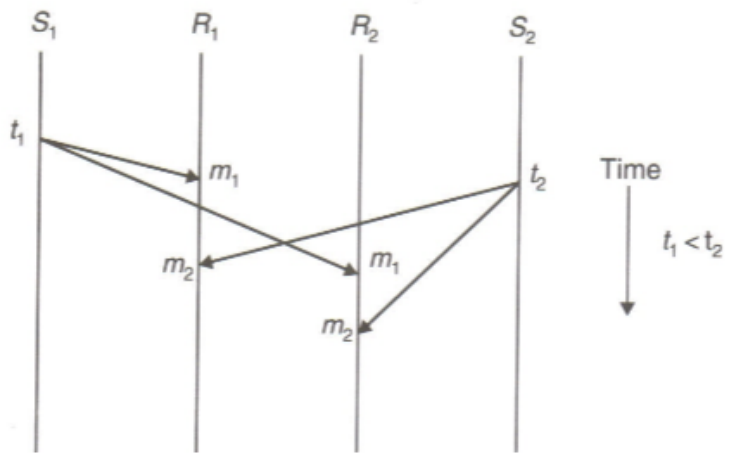


Figure 2.9: Absolute ordering of many-to-many messages.

2.5.1.2 Consistent Ordering

All messages are delivered to all receivers in the same order. However, this order may be different from the order in which the messages were originally sent.

A potential implementation of this follows:

- Convert the communication to many-to-one and one-to-many by using a central node, called a *sequencer*.
- The sequencer will then receive messages from all of the sending nodes, assign a sequence number to each message, and then multicast it to all receiving nodes.

However, it is subject to a single point of failure and thus has poor reliability.

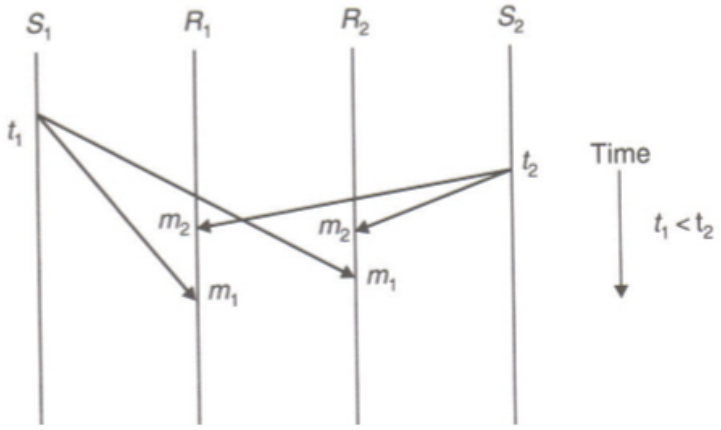


Figure 2.10: Consistent ordering of many-to-many messages.

2.5.1.3 Causal Ordering

If the event of sending one message is causally related to the event of sending another message, the two messages are delivered to all receivers in the correct order.

Two message-sending events are said to be causally related if they are correlated by the *happened-before* relation.

The happens-before relationship $a \rightarrow b$ is read "a happens before b" and means that all processes agree that a occurs, and then b occurs afterwards. It is transitive, so $a \rightarrow b$ and $b \rightarrow c$ implies $a \rightarrow c$. It can be directly observed in two scenarios:

1. If a and b are events in the same process, and a occurs before b, then $a \rightarrow b$ is true.
2. If a is the event of a message being sent by a process, and b is the event of a message being received by another process, then $a \rightarrow b$ is true.

An example of causal ordering can be found in CBCAST in the ISIS system.

2.6 Remote Procedure Call (RPC)

IPC in a distributed system is well-handled by a message-passing system, but any such IPC system is likely to be bespoke and will thus not be reusable across applications. As a result, a general IPC protocol was needed that could be reused; from this, the Remote Procedure Call (RPC) facility was created.

Remote procedure calls are similar to local procedure calls (which invoke a procedure in a single process on a single host), but invoke a procedure on a remote machine (see Figure 2.11). This allows them to offer a variety of features, including

- simple call syntax
- familiar syntax

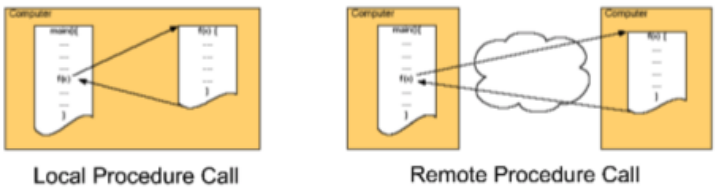


Figure 2.11: Local procedure calls vs remote procedure calls

Figure 2.12: A flow for the RPC model.

- specified well-defined interface
- ease of use
- generality
- efficiency

The average RPC flow can be seen in Figure 2.12.

To achieve semantic transparency (see subsubsection 1.2.5.3), RPC implementations use *stubs*. A stub is a normal local procedure that translates the parameters from the client into a network-friendly format, and then submits them to the *RPCRuntime*, responsible for sending it across to the network. This is illustrated in Figure 2.13.

These stubs are typically generated from the interface definition (similar to a class definition in an object-oriented language) of the server routines using development tools.

There are generally two types of RPC messages: call messages and reply messages. A call message

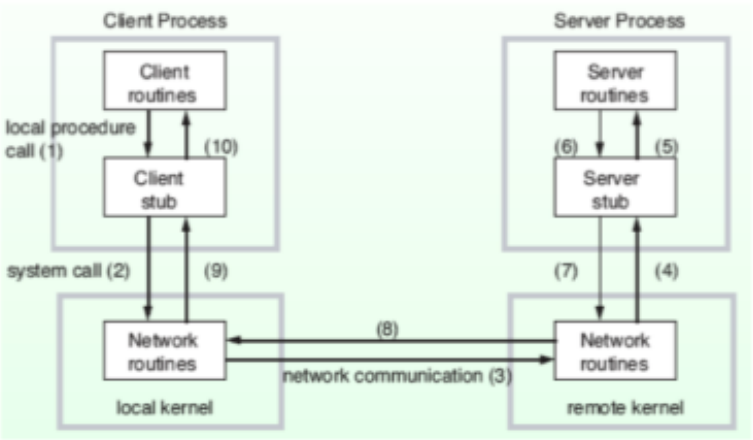


Figure 2.13: RPC in detail.

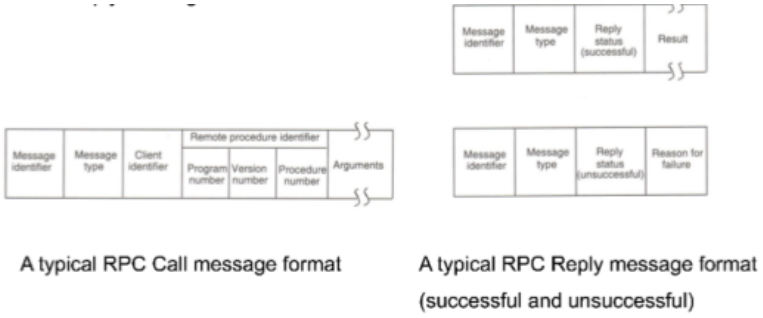


Figure 2.14: RPC Call vs RPC Reply.

is responsible for telling the receiver to call a particular procedure, while the reply message is the message from the receiver to the sender notifying them of the result of the call. This is illustrated in Figure 2.14.

2.6.1 Parameter Handling

Parameters may be passed to the stub in one of three ways: call-by-value, call-by-reference, and call-by-copy/restore.

In call-by-value, the value of the arguments are copied to the stack and passed to the procedure. The called procedure can modify its copy, but these modifications will not be propagated to the original value at the call site.

In call-by-reference, the memory addresses of the variables for the arguments are passed to the procedure. Changes made to the variables in the procedure will be propagated to the call site, as the call site variables *are* the procedure variables.

In call-by-copy/restore, the values of the arguments are copied to the stack and are passed to the

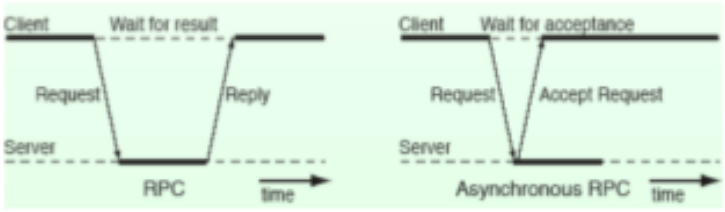


Figure 2.15: Asynchronous RPC.

procedure (similar to call-by-value). However, when the procedure finishes executing, the values are copied back to the call site, providing similar behaviour to call-by-reference.

Call-by-value and call-by-copy/restore are relatively trivial to implement for RPCs, but call-by-reference is significantly harder as local memory addresses do not map to remote memory addresses. Implementation of call-by-reference is done by copying to the remote machine, using the copy's address, and then copying back at the end of execution (similar to call-by-copy/restore).

As parameters may have different encodings on different machines, it is important to encode them to a common format to transmit over the network. If a standard format is not used, both sides will have to encode and decode the message as appropriate. Alternatively, the message can be annotated with the format in use, and the receiving computer can decode the message knowing its format; but the receiver will need to be able to handle arbitrary formats.

2.6.2 Variations

There are multiple variations on RPCs. These are:

- **Asynchronous RPC**: Typically, a RPC is synchronous; the calling code will wait for the call to finish executing on the remote machine. As this can lead to unnecessarily waiting, the remote machine can instead inform the calling code that it has accepted the request immediately. This dramatically reduces the amount of time spent waiting, as seen in Figure 2.15.
- **One-way RPC**: After sending the RPC to the remote machine, the calling code will immediately continue and not wait for a response.
- **Callback RPC, Broadcast RPC, Batch-mode RPC, Lightweight RPC**: These are not explained.

2.6.3 Optimisations For Better Performance

To optimise RPC performance, the following can be investigated:

- **Concurrent access to multiple servers**: A client can connect to multiple servers. To do this, it can use threads (where each thread can make independent RPCs to different servers), an early reply approach (where RPCs are made, and then the response is retrieved at a later stage), or a call buffering approach (where an intermediary is used to dispatch RPCs across clients and servers appropriately).

- **Serving multiple requests simultaneously**: A server may be blocked on other resources while attending to a RPC, and will thus end up underutilised. To mitigate this, the server can handle other RPCs while waiting for the original resource to be freed. Threads can also help with this.
- **Reducing per-call workload of servers**: Reducing the work associated with any given call will help performance in general.
- **Reply caching of idempotent remote procedures**: See subsection 2.4.5.
- **Proper selection of timeout values**: If a RPC fails to respond, a long timeout could cause the client to stall for an extended period of time. A shorter timeout will reduce the delay before the client can choose its next course of action.
- **Proper design of RPC protocol specification**: The RPC protocol should be designed for performance; otherwise, it may be unnecessarily inefficient.

Chapter 3

Message Passing Interface

The Message Passing Interface (MPI) is a specification for a message passing standard. It is *not* a library; however, there are multiple library implementations. The interface aims to be practical, portable, efficient and flexible.

Reasons for using MPI include:

- **Standardisation**: MPI is the only message passing library that is considered a standard. It is supported on virtually all major platforms and many specialised HPC systems.
- **Portability**: There is no need to modify your source code if porting your application to a different platform that is compliant with the MPI standard.
- **Performance Opportunities**: Vendor implementations are allowed to use native hardware features to optimise implementations.
- **Functionality**: MPI-1 has over 115 routines.
- **Availability**: There are a variety of implementations, including vendor and public domain/open-source.

MPI can be used to implement the majority of distributed memory parallel programming models, and can be used as the backend for some shared memory models on distributed memory architectures. Originally, MPI was targeted towards distributed memory platforms, but has since grown to support shared and hybrid memory platforms.

In MPI, all parallelism is explicit; the programmer is responsible for identifying opportunities for parallelism and implementing parallel algorithms using MPI constructs. Additionally, the number of tasks dedicated to running a parallel program is static and cannot be changed (unless MPI 2 is used, which addresses this issue).

3.1 Communicators and Groups

MPI uses objects called communicators and groups to define which collection of processes may communicate with each other. Most MPI routines require you to specify a communicator as an argument. `MPI_COMM_WORLD` is the predefined communicator that includes all MPI processes.

Within a communicator, every process has its own unique, integer identifier, called the *rank*, assigned by the system when the process initializes. A rank is sometimes also called a "task ID". Ranks are contiguous and begin at zero. They are used by the programmer to specify the source and destination of messages. They are also used conditionally by the application to control program execution (if rank=0, do this, else do something else).

3.2 Environment Management

3.2.1 Routines [1]

3.2.1.1 MPI_Init

```
MPl_lnit(&argc, &argv)
```

Initializes the MPI execution environment. This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program. For C programs, `MPI_Init` may be used to pass the command line arguments to all processes, although this is not required by the standard and is implementation dependent.

3.2.1.2 MPI_Comm_size

```
MPI_Comm_size(comm, &size)
```

Determines the number of processes in the group associated with a communicator. Generally used within the communicator `MPI_COMM_WORLD` to determine the number of processes being used by your application.

3.2.1.3 MPI_Comm_rank

```
MPI_Comm_rank(comm, &rank)
```

Determines the rank of the calling process within the communicator. Initially, each process will be assigned a unique integer rank between 0 and number of processors - 1 within the communicator `MPI_COMM_WORLD`. This rank is often referred to as a task ID.

3.2.1.4 MPI_Abort

```
MPI_Abort(comm, errorcode)
```

Terminates all MPI processes associated with the communicator. In most MPI implementations it terminates ALL processes regardless of the communicator specified.

[1] C'mon, you can't expect me to rewrite documentation.

3.2.1.5 MPI_Get_processor_name

```
MPI_Get_processor_name(&name, &resultlength)
```

Returns the processor name. Also returns the length of the name. The buffer for `name` must be at least `MPI_MAX_PROCESSOR_NAME` characters in size. What is returned into `name` is implementation dependent - may not be the same as the output of the `hostname` or `host` shell commands.

3.2.1.6 MPI_Initialized

```
MPI_Initialized(&flag)
```

Indicates whether `MPI_Init` has been called - returns flag as either logical true (1) or false (0). MPI requires that `MPI_Init` be called once and only once by each process. This may pose a problem for modules that want to use MPI and are prepared to call `MPI_Init` if necessary. `MPI_Initialized` solves this problem.

3.2.1.7 MPI_Wtime

```
MPI_Wtime()
```

Returns an elapsed wall clock time in seconds (double precision) on the calling processor.

3.2.1.8 MPI_Wtick

```
MPI_Wtick()
```

Returns the resolution in seconds (double precision) of `MPI_Wtime`.

3.2.1.9 MPI_Finalize

```
MPI_Finalize()
```

Terminates the MRI execution environment. This function should be the last MPI routine called in every MRI program - no other MPI routines may be called after it.

3.3 Point to Point Communication

MPI point-to-point operations typically involve message passing between two, and only two, different MPI tasks. One task is performing a send operation and the other task is performing a matching receive operation.

There are different types of send and receive routines used for different purposes. For example:

- Synchronous send
- Blocking send / blocking receive
- Non-blocking send / non-blocking receive

- Buffered send
- Combined send/receive
- "Ready" send

Any type of send routine can be paired with any type of receive routine. MPI also provides several routines associated with send-receive operations, such as those used to wait for a message's arrival or probe to find out if a message has arrived.

In a perfect world, every send operation would be perfectly synchronized with its matching receive. This is rarely the case. Somehow or other, the MPI implementation must be able to deal with storing data when the two tasks are out of sync.

Consider the following two cases:

- A send operation occurs 5 seconds before the receive is ready - where is the message while the receive is pending?
- Multiple sends arrive at the same receiving task which can only accept one send at a time - what happens to the messages that are "backing up"?

The MPI implementation (not the MPI standard) decides what happens to data in these types of cases. Typically, a *system buffer* area is reserved to hold data in transit.

This system buffer space is:

- Opaque to the programmer and managed entirely by the MPI library
- A finite resource that can be easy to exhaust
- Often mysterious and not well documented
- Able to exist on the sending side, the receiving side, or both
- Something that may improve program performance because it allows send-receive operations to be asynchronous.

User managed address space (i.e. your program variables) is called the application buffer. MPI also provides for a user managed send buffer.

MPI guarantees that messages will not overtake each other. If a sender sends two messages (Message 1 and Message 2) in succession to the same destination, and both match the same receive, the receive operation will receive Message 1 before Message 2. If a receiver posts two receives (Receive 1 and Receive 2), in succession, and both are looking for the same message, Receive 1 will receive the message before Receive 2. Order rules do not apply if there are multiple threads participating in the communication operations.

MPI does not guarantee *fairness* - it's up to the programmer to prevent "operation starvation". Example: task 0 sends a message to task 2. However, task 1 sends a competing message that matches task 2's receive. Only one of the sends will complete.

Most of the MPI point-to-point routines can be used in either blocking or non-blocking mode.

If the routines are blocking, they are subject to the following:

- They will only "return" after it is safe to modify the application buffer (your send data) for reuse. Safe means that modifications will not affect the data intended for the receive

task. Safe does not imply that the data was actually received - it may very well be sitting in a system buffer.

- A send can be synchronous which means there is a handshake occurring with the receive task to confirm a safe send.
- A send can be asynchronous if a system buffer is used to hold the data for eventual delivery to the receive.
- A receive only "returns" after the data has arrived and is ready for use by the program.

If the routines are non-blocking, they are subject to the following:

- Send and receive routines behave similarly - they will return almost immediately. They do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message.
- They simply "request" the MPI library to perform the operation when it is able. The user can not predict when that will happen.
- It is unsafe to modify the application buffer (your variable space) until you know for a fact the requested non-blocking operation was actually performed by the library. There are "wait" routines used to do this.
- They are primarily used to overlap computation with communication and exploit possible performance gains.

3.3.1 Communication Routines and Arguments

Sends	
Blocking sends	`MPI_Send(buf, count, datatype, dest, tag, comm)` Basic blocking send operation. Routine returns only after the application buffer in the sending task is free for reuse. Note that this routine may be implemented differently on different systems. The MPI standard permits the use of a system buffer but does not require it. Some implementations may actually use a synchronous send (discussed below) to implement the basic blocking send.
Synchronous blocking sends	`MPI_Ssend(buf, count, datatype, dest, tag, comm)` Send a message and block until the application buffer in the sending task is free for reuse and the destination process has started to receive the message.
Buffered blocking sends	`MPI_Bsend(buf, count, datatype, dest, tag, comm)` Permits the programmer to allocate the required amount of buffer space into which data can be copied until it is delivered. Insulates against the problems associated with insufficient system buffer space. Routine returns after the data has been copied from application buffer space to the allocated send buffer. Must be used with the `MPI_Buffer_attach` routine.
Blocking ready sends	`MPI_Rsend(buf, count, datatype, dest, tag, comm)` Should only be used if the programmer is certain that the matching receive has already been posted.
Non-blocking sends	`MPI_Isend(buf, count, datatype, dest, tag, comm, request)` Identifies an area in memory to serve as a send buffer. Processing continues immediately without waiting for the message to be copied out from the application buffer. A communication request handle is returned for handling the pending message status. The program should not modify the application buffer until subsequent calls to `MPI_Wait` or `MPI_Test` indicate that the non-blocking send has completed.
Non-blocking synchronous sends	`MPI_Issend(buf, count, datatype, dest, tag, comm, request)` Similar to `MPI_Isend()`, except `MPI_Wait()` or `MPI_Test()` indicates when the destination process has received the message.
Non-blocking buffered sends	`MPI_Ibsend(buf, count, datatype, dest, tag, comm, request)` Similar to `MPI_Bsend()`, except `MPI_Wait()` or `MPI_Test()` indicates when the destination process has received the message.
Non-blocking ready sends	`MPI_Irsend(buf, count, datatype, dest, tag, comm, request)` Similar to `MPI_Rsend()`, except `MPI_Wait()` or `MPI_Test()` indicates when the destination process has received the message.

Receives	
Blocking receive	`MPI_Recv(buf, count, datatype, source, tag, comm, status)` Receive a message and block until the requested data is available in the application buffer in the receiving task.
Non-blocking receive	`MPI_Irecv(buf, count, datatype, source, tag, comm, request)` Identifies an area in memory to serve as a receive buffer. Processing continues immediately without actually waiting for the message to be received and copied into the the application buffer. A communication request handle is returned for handling the pending message status. The program must use calls to `MPI_Wait` or `MPI_Test` to determine when the non-blocking receive operation completes and the requested message is available in the application buffer.

Other	
Send-receive	`MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)` Send a message and post a receive before blocking. Will block until the sending application buffer is free for reuse and until the receiving application buffer contains the received message.
Buffer attaching	`MPI_Buffer_attach(buffer, size)` Used by programmer to allocate/deallocate message buffer space to be used by the `MPI_Bsend` routine. The size argument is specified in actual data bytes - not a count of data elements. Only one buffer can be attached to a process at a time.

Blocking wait for non-blocking	`MPI_Wait(request, status)`
	`MPI_Waitany(count, arrayofrequests, index, status)`
	`MPI_Waitall(count, arrayofrequests, arrayofstatuses)`
	`MPI_Waitsome(incount, arrayofrequests, outcount, arrayofoffsets, arrayofstatuses)`
	Blocks until a specified non-blocking send or receive operation has completed. For multiple non-blocking operations, the programmer can specify any, all or some completions.

Blocking test for a message	`MPI_Probe(source, tag, comm, status)` Performs a blocking test for a message. The "wildcards" `MPI_ANY_SOURCE` and `MPI_ANY_TAG` may be used to test for a message from any source or with any tag.

Status of non-blocking	`MPI_Test(request, status)`
	`MPI_Testany(count, arrayofrequests, index, flag, status)`
	`MPI_Testall(count, arrayofrequests, arrayofstatuses)`
	`MPI_Testsome(incount, arrayofrequests, outcount, arrayofindices, arrayofstatuses)`
	Checks the status of a specified non-blocking send or receive operation. The "flag" parameter is returned logical true (1) if the operation has completed, and logical false (0) if not. For multiple non-blocking operations, the programmer can specify any, all or some completions.

Non-blocking test for a message	`MPI_Iprobe(source, tag, comm, status)` Performs a non-blocking test for a message. The "wildcards" `MPI_ANY_SOURCE` and `MPI_ANY_TAG` may be used to test for a message from any source or with any tag. The "flag" parameter is returned logical true (1) if the operation has completed, and logical false (0) if not.

- **buf**: Program (application) address space that references the data that is to be sent or received. In most cases, this is simply the variable name that is be sent/received. For C programs, this argument is passed by reference and usually must be prepended with an ampersand: `&var1`.
- **count**: Indicates the number of data elements of a particular type to be sent.
- **datatype**: For reasons of portability, MPI predefines its elementary data types. These are listed in Table 3.1.
- **dest**: An argument to `send` routines that indicates the process where a message should be delivered. Specified as the rank of the receiving process.

Table 3.1: A list of MPI data types and their descriptions.

Data type	Description
MPI_CHAR	8-bit character
MPI_BYTE	8-bit byte
MPI_DOUBLE	64-bit floating point
MPI_FLOAT	32-bit floating point
MPI_INT	32-bit integer
MPI_LONG	32-bit integer
MPI_LONG_DOUBLE	64-bit floating point
MPI_LONG_LONG	64-bit integer
MPI_LONG_LONG_INT	64-bit integer
MPI_SHORT	16-bit integer
MPI_SIGNED_CHAR	8-bit signed character
MPI_UNSIGNED	32-bit unsigned integer
MPI_UNSIGNED_CHAR	8-bit unsigned character
MPI_UNSIGNED_LONG	32-bit unsigned integer
MPI_UNSIGNED_LONG_LONG	64-bit unsigned integer
MPI_UNSIGNED_SHORT	16-bit unsigned integer
MPI_WCHAR	Wide (16-bit) unsigned character
MPI_PACKED	Data packed or unpacked with MPI_Pack or MPI_Unpack

- **source**: An argument to `receive` routines that indicates the originating process of the message. Specified as the rank of the sending process. This can be set to `MPI_ANY_SOURCE` to receive messages from any task.
- **tag**: Arbitrary non-negative integer assigned by the programmer to uniquely identify a message. Send and receive operations should match message tags. For a `receive` operation, the wild card `MPI_ANY_TAG` can be used to receive any message regardless of its tag.

  The MPI standard guarantees that `0` to `32767` can be used as tags, but most implementations allow for a much larger range.
- **comm**: Indicates the communication context, or set of processes for which the source or destination fields are valid. Unless the programmer is explicitly creating new communicators, the predefined communicator `MPI_COMM_WORLD` is usually used.
- **status**: For a receive operation, indicates the source of the message and the tag of the message. In C, this argument is a pointer to a predefined structure `MPI_Status`. Additionally, the actual number of bytes received are obtainable from `status` via the `MPI_Get_count` routine.
- **request**: Used by non-blocking send and receive operations. Since non-blocking operations may return before the requested system buffer space is obtained, the system issues a unique "request number". The programmer uses this system assigned "handle" later (in a `WAIT` type routine) to determine completion of the non-blocking operation. In C, this argument is a pointer to a predefined structure `MPI_Request`.

3.4 Collective Communication

Collective communication must involve all processes in the scope of a communicator. All processes are by default, members in the communicator `MPI_COMM_WORLD`.

It is the programmer's responsibility to ensure that all processes within a communicator participate in any collective operations.

Types of collective operations include:

- **Synchronization**: processes wait until all members of the group have reached the synchronization point.
- **Data Movement**: broadcast, scatter/gather, all to all.
- **Collective Computation (reductions)**: one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data.

Considerations include:

- Collective operations are blocking.
- Collective communication routines do not take message tag arguments.
- Collective operations within subsets of processes are accomplished by first partitioning the subsets into new groups and then attaching the new groups to new communicators (discussed in the Group and Communicator Management Routines section).
- Can only be used with MPI predefined datatypes - not with MPI Derived Data Types.

3.4.1 Routines

3.4.1.1 `MPI_Barrier`

```
MPI_Barrier(comm)
```

Creates a barrier synchronization in a group. Each task, when reaching the `MPI_Barrier` call, blocks until all tasks in the group reach the same `MPI_Barrier` call.

3.4.1.2 `MPI_Bcast`

```
MPI_Bcast(buffer,count,datatype,root,comm)
```

Broadcasts/sends a message from the process with rank "root" to all other processes in the group.

3.4.1.3 `MPI_Scatter`

```
MPI_Scatter(sendbuf,sendcnt,sendtype,recvbuf,recvcnt,recvtype,root,comm)
```

Distributes distinct messages from a single source task to each task in the group.

3.4.1.4 MPI_Gather

`MPI_Gather(sendbuf,sendcnt,sendtype,recvbuf,recvcount,recvtype,root,comm)`

Gathers distinct messages from each task in the group to a single destination task. This routine is the reverse operation of `MPI_Scatter`.

3.4.1.5 MPI_Allgather

`MPI_Allgather(sendbuf,sendcount,sendtype,recvbuf,recvcount,recvtype,comm)`

Concatenation of data to all tasks in a group. Each task in the group, in effect, performs a one-to-all broadcasting operation within the group.

3.4.1.6 MPI_Reduce

`MPI_Reduce(sendbuf,recvbuf,count,datatype,op,root,comm)`

Applies a reduction operation on all tasks in the group and places the result in one task.

3.4.1.7 MPI_Allreduce

`MPI_Allreduce(sendbuf,recvbuf,count,datatype,op,comm)`

Applies a reduction operation and places the result in all tasks in the group. This is equivalent to an MPI_Reduce followed by an MPI_Bcast.

3.4.1.8 MPI_Reduce_scatter

`MPI_Reduce_scatter(sendbuf,recvbuf,recvcount,datatype, op,comm)`

First does an element-wise reduction on a vector across all tasks in the group. Next, the result vector is split into disjoint segments and distributed across the tasks. This is equivalent to an `MPI_Reduce` followed by an `MPI_Scatter` operation.

3.4.1.9 MPI_Alltoall

`MPI_Alltoall(sendbuf,sendcount,sendtype,recvbuf,recvcnt,recvtype,comm)`

Each task in a group performs a scatter operation, sending a distinct message to all the tasks in the group in order by index.

3.4.1.10 MPI_Scan

`MPI_Scan(sendbuf,recvbuf,count,datatype,op,comm)`

Performs a scan operation with respect to a reduction operation across a task group.

3.4.2 Derived data types

MPI provides facilities for you to define your own data structures based upon sequences of the MPI primitive data types. Such user defined structures are called derived data types.

Primitive data types are contiguous. Derived data types allow you to specify non-contiguous data in a convenient manner and to treat it as though it was contiguous.

MPI provides several methods for constructing derived data types:

- Contiguous
- Vector
- Indexed
- Struct

3.4.2.1 `MPI_Type_contiguous`

`MPI_Type_contiguous(count,oldtype,newtype)`

The simplest constructor. Produces a new data type by making count copies of an existing data type.

3.4.2.2 `MPI_Type_vector`

`MPI_Type_vector(count,blocklength,stride,oldtype,newtype)`

Similar to contiguous, but allows for regular gaps (stride) in the displacements. `MPI_Type_hvector` is identical to `MPI_Type_vector` except that stride is specified in bytes.

3.4.2.3 `MPI_Type_indexed`

`MPI_Type_indexed(count,blocklens,offsets,old_type,newtype)`

An array of displacements of the input data type is provided as the map for the new data type. `MPI_Type_hindexed` is identical to `MPI_Type_indexed` except that offsets are specified in bytes.

3.4.2.4 `MPI_Type_struct`

`MPI_Type_struct(count,blocklens,offsets,old_types,newtype)`

The new data type is formed according to completely defined map of the component data types.

3.4.2.5 `MPI_Type_extent`

`MPI_Type_extent(datatype,extent)`

Returns the size in bytes of the specified data type. Useful for the MPI subroutines that require specification of offsets in bytes.

3.4.2.6 MPI_Type_commit

`MPI_Type_commit(datatype)`

Commits new datatype to the system. Required for all user constructed (derived) datatypes.

3.4.2.7 MPI_Type_free

`MPI_Type_free(datatype)`

Deallocates the specified datatype object. Use of this routine is especially important to prevent memory exhaustion if many datatype objects are created, as in a loop.

3.5 Groups and Communicators

A group is an ordered set of processes. Each process in a group is associated with a unique integer rank. Rank values start at zero and go to $N-1$, where N is the number of processes in the group. In MPI, a group is represented within system memory as an object. It is accessible to the programmer only by a "handle". A group is always associated with a communicator object.

A communicator encompasses a group of processes that may communicate with each other. All MPI messages must specify a communicator. In the simplest sense, the communicator is an extra "tag" that must be included with MPI calls. Like groups, communicators are represented within system memory as objects and are accessible to the programmer only by "handles". For example, the handle for the communicator that comprises all tasks is `MPI_COMM_WORLD`. A diagram showing an example use can be seen in Figure 3.1.

From the programmer's perspective, a group and a communicator are one. The group routines are primarily used to specify which processes should be used to construct a communicator.

The primary purposes of group and communicator objects are:

- Allow you to organize tasks, based upon function, into task groups.
- Enable Collective Communications operations across a subset of related tasks.
- Provide basis for implementing user defined virtual topologies
- Provide for safe communications

Considerations when using groups and communicators are:

- Groups/communicators are dynamic - they can be created and destroyed during program execution.
- Processes may be in more than one group/communicator. They will have a unique rank within each group/communicator.
- MPI provides over 40 routines related to groups, communicators, and virtual topologies.
- Typical usage:
  - Extract handle of global group from `MPI_COMM_WORLD` using `MPI_Comm_group`
  - Form new group as a subset of global group using `MPI_Group_incl`

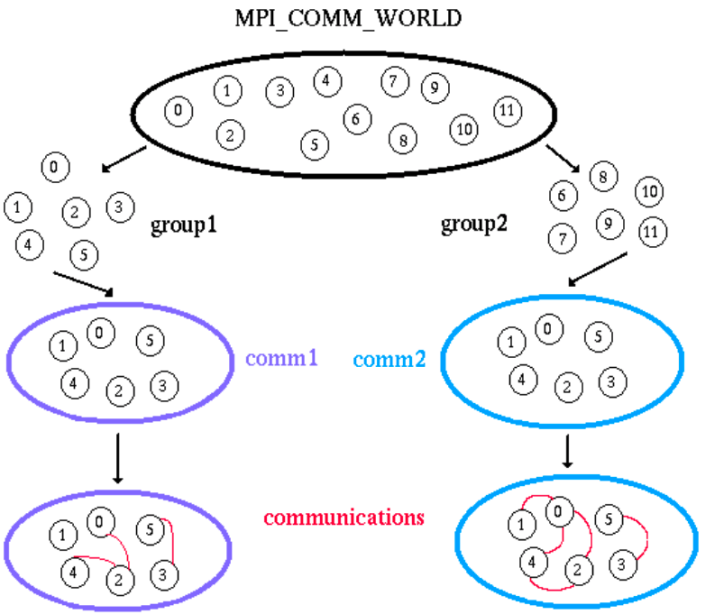


Figure 3.1: A diagram of groups and communicators.

- Create new communicator for new group using `MPI_Comm_create`
- Determine new rank in new communicator using `MPI_Comm_rank`
- Conduct communications using any MPI message passing routine
- When finished, free up new communicator and group (optional) using `MPI_Comm_free` and `MPI_Group_free`

MPI includes routines for accessing information on groups or communicators, for creating new groups or communicators from existing ones, and for deleting groups or communicators.

Communicator creation routines are collective. They require all processes in the input communicator to participate, and may require communication amongst processes. All other group and communicator routines are local. It often makes sense to have all members of an input group call a group creation routine, if a communicator will later be created for that group.

3.5.1 Group Accessors

- `MPI_Group_size` returns number of processes in a group.
- `MPI_Group_rank` returns rank of calling process in a group.
- `MPI_Group_translate_ranks` translates ranks of processes in one group to those in another group.
- `MPI_Group_compare` compares group members and group order.

3.5.2 Group Constructors

- `MPI_Comm_group` returns the group associated with a communicator.
- `MPI_Group_union` creates a group by combining two groups.
- `MPI_Group_intersection` creates a group from the intersection of two groups.
- `MPI_Group_difference` creates a group from the difference between two groups.
- `MPI_Group_incl` creates a group from listed members of an existing group.
- `MPI_Group_excl` creates a group excluding listed members of an existing group.
- `MPI_Group_range_incl` creates a group according to first rank, stride, last rank.
- `MPI_Group_range_excl` creates a group by deleting according to first rank, stride, last rank.

3.5.3 Group Destructors

- `MPI_Group_free` marks a group for deallocation.

3.5.4 Communicator Accessors

- `MPI_Comm_size` returns number of processes in communicator's group.
- `MPI_Comm_rank` returns rank of calling process in communicator's group.
- `MPI_Comm_compare` compares two communicators.

3.5.5 Communicator Constructors

- `MPI_Comm_dup` duplicates a communicator.
- `MPI_Comm_create` creates a new communicator for a group.
- `MPI_Comm_split` splits a communicator into multiple, non-overlapping communicators.

3.5.6 Communicator Destructors

- `MPI_Comm_free` marks a communicator for deallocation.

3.6 Virtual Topologies

In terms of MPI, a virtual topology describes a mapping/ordering of MPI processes into a geometric "shape". The two main types of topologies supported by MPI are Cartesian (grid) and Graph.

MPI topologies are virtual - there may be no relation between the physical structure of the parallel machine and the process topology. Virtual topologies are built upon MPI communicators and groups, and must be "programmed" by the application developer.

Why use them? Virtual topologies may be useful for applications with specific communication patterns - patterns that match an MPI topology structure. For example, a Cartesian topology might prove convenient for an application that requires 4-way nearest neighbor communications for grid based data.

Additionally, some hardware architectures may impose penalties for communications between successively distant "nodes". A particular implementation may optimize process mapping based upon the physical characteristics of a given parallel machine. The mapping of processes into an MPI virtual topology is dependent upon the MPI implementation, and may be totally ignored.

Chapter 4

Synchronisation, Mutexes and Deadlocks

A Distributed System consists of a collection of distinct processes that are spatially separated and run concurrently. In systems with multiple concurrent processes, it is economical to share the system resources. This sharing may be *cooperative* or *competitive*.

Both competitive and cooperative sharing require adherence to certain rules of behavior that guarantee that correct interaction occurs. The rules of enforcing correct interaction are implemented in the form of synchronization mechanisms.

In single CPU systems, synchronization problems such as mutual exclusion can be solved using semaphores and monitors. These methods rely on the existence of shared memory.

We cannot use semaphores and monitors in distributed systems since two processes running on different machines cannot expect to have shared memory. Even simple matters such as determining one event happened before the other event requires careful thought.

In distributed systems, it is usually not possible or desirable to collect all the information about the system in one place and synchronization among processes is difficult due to the following features of distributed systems:

- The relevant information is scattered among multiple machines.
- Processes make decisions based only on local information.
- A single point of failure in the system should be avoided.
- No common clock or other precise global time source exists.

4.1 Clocks

Why do we want to distribute the current time across the system? There are external reasons; we often want to measure time accurately:

- For billings: How long was computer X used?

- For legal reasons: When was credit card W charged?
- For traceability: When did this attack occurred? Who did it?

To achieve this, the system must be in sync with an external time reference; this is usually the world time reference UTC (Coordinated Universal Time).

There are also internal reasons; many distributed algorithms use time:

- Kerberos (authentication server) uses time-stamps
- Internal time can be used to serialise transactions in databases
- Internal time can be used to minimise updates when replicating data

To achieve this, the system must be synchronised internally; this *does not* require synchronisation to an external time reference!

Time is unambiguous in a centralised system. A process can always make a system call to know the time; if two processes A and B are running on the same system and B requests the current time B_{time} after A requests A_{time}, then it is guaranteed that $B_{time} > A_{time}$.

However, this is not the case in a distributed system. If the two processes are on different machines, B_{time} may not be greater than A_{time}.

In general, human-made clocks are imperfect. They run slower or faster than "real" physical time. The amount of variation from "real time" is called the *drift*; a drift of 1% implies that the clock gains or loses a second every 100 seconds.

This can be quantified. If the real time is t, the measured time value of a clock is $C_p(t)$, and the maximum drift rate allowable is ρ, a clock can be said to be non-faulty [1] if the following condition holds:

$$1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho$$

4.1.1 Cristian's Algorithm

Cristian's algorithm, as depicted in Figure 4.1, synchronizes the clocks of all other machines to the clock of one machine, the *time server*. If the clock of the time server is adjusted to the real time, all the other machines are synchronized to the real time.

Every machine requests the current time to the time server. The time server responds to the request as soon as possible. The requesting machine sets its clock to

$$C_s + \frac{T_1 - T_0 - I}{2}$$

To avoid clocks moving backwards for any machine, a clock adjustment must be introduced gradually.

[1] It is always assumed that any clock has some amount of drift. The nature of physics makes it implausible that a "perfect" clock will ever exist.

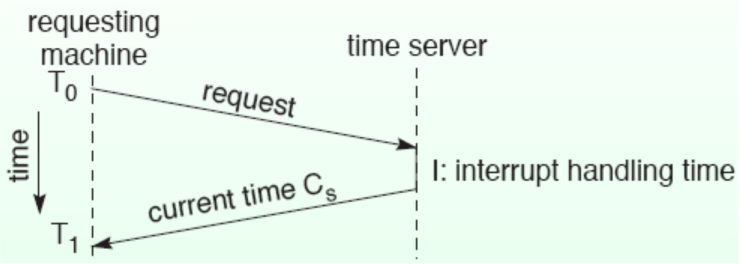


Figure 4.1: Cristian's algorithm for clock synchronisation.

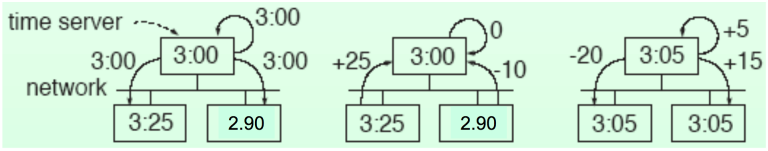


Figure 4.2: The Berkeley algorithm for clock synchronisation.

4.1.2 Berkeley Algorithm

The *Berkeley algorithm*, as developed by Gusella and Zatti, uses a master server that communicates with its slaves to determine the time offset to be applied to each slave. It is illustrated in Figure 4.2.

The complete algorithm generally follows these steps:

- A master server is chosen with a ring-based election algorithm (Chang and Roberts algorithm).
- The master polls the slaves who reply with their time in a similar way to Cristian's algorithm.
- The master observes the round-trip time (RTT) of the messages and estimates the time of each slave and its own.
- The master then averages the clock times, ignoring any values it receives far outside the values of the others.
- Instead of sending the updated current time back to the other process, the master then sends out the amount (positive or negative) that each slave must adjust its clock. This avoids further uncertainty due to the RTT of the slave processes.
- Everybody adjusts their times appropriately.

The use of an average cancels out the tendency of individual clocks to drift, addressing one of the major shortcomings of Cristian's algorithm.

However, special consideration must be given to avoiding direct application of a negative clock alteration. Computers generally rely on *monotonic time*, in which the clock value must either stay the same or go forwards; it must never go backwards.

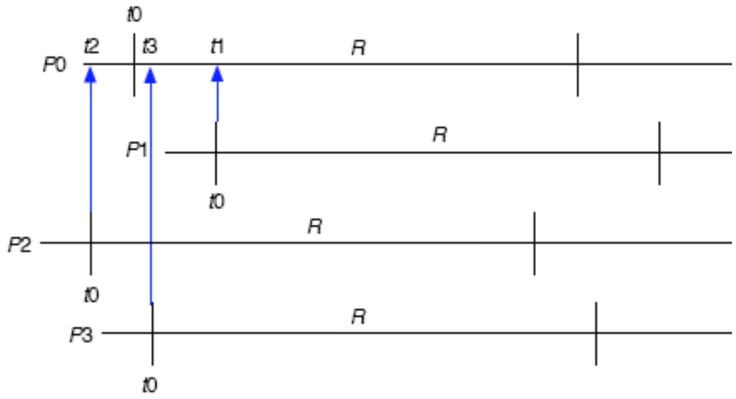


Figure 4.3: An example of the averaging algorithm for clock synchronisation.

The simple solution is to halt the clock in the event of a negative clock alteration, so that time naturally synchronises to the alteration required. However, this may cause issues of its own (as time is effectively stopped). For minor corrections, most systems will prefer to slow the clock (known as *clock skew*), so that the correction is distributed across a longer period of time.

4.1.3 Averaging Algorithm

Both Cristian's algorithm and the Berkeley algorithm are centralized algorithms with disadvantages including the existence of a central point of failure and high traffic volume around the server.

The *averaging algorithm* is a decentralized algorithm. It divides time into resynchronization intervals with a fixed length R. Every machine broadcasts the current time at the beginning of each interval according to its clock. A machine collects all other broadcasts for a certain interval and sets the local clock by the average of their arrival times.

Figure 4.3 depicts an example of the averaging algorithm for a system with four nodes. The clock on processor $P0$ should be advanced in accordance with the increment δt_0:

$$\delta t_0 = \frac{t_0 + t_1 + t_2 + t_3}{4} - t_0$$

4.1.4 Lamport's Synchronisation Algorithm

Leslie Lamport showed that clock synchronisation need not be absolute. If two processes do not interact, they do not need synchronised clocks. All that matters is that processes agree on the order in which events occur.

Based on this, it can be said that there are two types of clocks: *physical clocks* and *logical clocks*. Physical clocks agree on time within a certain time limit; the previous synchronisation algorithms attempt to synchronise physical clocks.

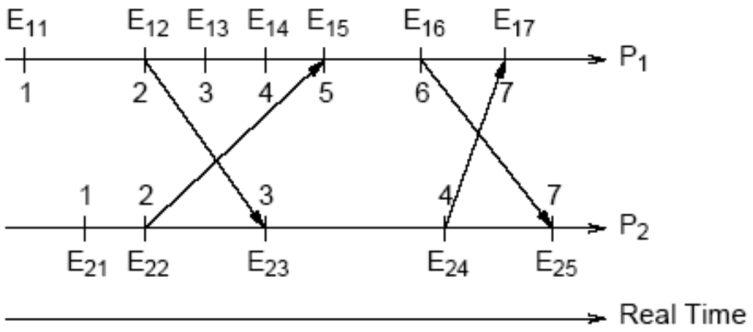


Figure 4.4: Illustration of the shortcoming with Lamport clocks.

Logical clocks are based on the assumption that the time itself is irrelevant for many purposes; all that matters is that all processes agree on some time value that can be used to determine the sequence in which events occur. This time value *does not need to be the real time.*

Lamport's algorithm is used to determine the ordering of events. This is done by establishing the happens-before relation (see subsubsection 2.5.1.3). Note that if two events X and Y occur in different processes that do not interact (even indirectly), then neither $X \to Y$ and $Y \to X$; these events occur *concurrently.*

The goal of the algorithm is to assign a time value $C(A)$ for which all processes agree on for any event A. This time value must obey $A \to B \implies C(A) < C(B)$ and that $C(A)$ *always* increases and can never decrease.

If event A happens before event B within the same process, $C(A) < C(B)$ is satisfied. If event A and event B represent the sending and receiving of a message, the clock of the receiving side is set so that $C(A) < C(B)$. For all events, the clock is increased at least by 1 between two events.

The largest shortcoming of the Lamport clock is that while $a \to b \implies C(A) < C(B)$, $C(A) < C(B) \not\iff a \to b$. This means that causal dependencies cannot be derived from time stamps.

The cause of this behaviour is that clocks advance independently or from messages, but there is no way of recovering the impetus for a clock advancement.

In Figure 4.4, $E_{12} \to E_{23}$ implies that $C_1(E_{12}) < C_2(E_{23})$; however, it is possible that $E_{13} \not\to E_{24}$ even though $C_1(E_{13}) < C_2(E_{24})$.

This results in a partial ordering on events, which is insufficient for situations in which a total ordering is required (e.g. distributed locks).

4.1.5 Vector Clock

Each process i maintains a vector clock $\mathbf{V}_i$ of size N, where N is the number of processes. $\mathbf{V}_i[j]$ refers to process i's knowledge of process j's clock. The vector $\mathbf{V}_i[j]$ is initialised with 0 for $i, j \in \{1, 2, \ldots, N\}$.

The clocks are advanced in accordance with these steps:

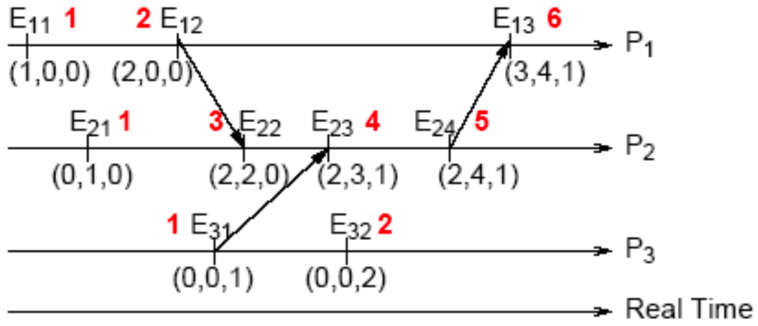


Figure 4.5: Vector clock compared to a Lamport clock.

1. Before process i timestamps an event, it executes $\mathbf{V}_i[i] = \mathbf{V}_i[i] + 1$; that is, it increments its timestamp for itself.
2. When a message m is sent from process i to process j:
   - Process i executes $\mathbf{V}_i[i] = \mathbf{V}_i[i] + 1$ and sends $\mathbf{V}_i$ with m.
   - Process j receives m and $\mathbf{V}_i$, and merges its own vector clock $\mathbf{V}_j$ with $\mathbf{V}_i$ using the following equation:

$$\mathbf{V}_j[k] = \begin{cases} \max(\mathbf{V}_j[k], \mathbf{V}_i[k]) + 1 & \text{if } j = k \text{ (as in scalar clocks)} \\ \max(\mathbf{V}_j[k], \mathbf{V}_i[k]) & \text{otherwise} \end{cases}$$

This guarantees that everything that occurs on process j after m is now causally related to everything that previously happened at process i.

This addresses the shortcoming of the Lamport clock as it can be used to establish whether a causal relationship exists or not. This is illustrated in Figure 4.5, where each triple is the vector clock value at that instant for that process, and the red value is the corresponding scalar Lamport clock.

Note that $C_1(E_{12}) = C_3(E_{32})$, but the vector clocks $(2, 0, 0)$ and $(0, 0, 2)$ do not agree; this implies that these events are not causally related. Additionally, $C_2(E_{24}) > C_3(E_{32})$, but $(2, 4, 1) \not> (0, 0, 2)$ and thus $E_{32} \not\to E_{24}$.

4.2 Mutual exclusion

When multiple processes access shared resources, the concept of *critical sections* is a relatively easy way to control access of the shared resources. Critical sections are sections in a program that access shared resources; these regions effectively act as gates that prevent other processes from accessing the resource until the process using the resource exits the region.

However, critical sections are implemented using semaphores and monitors in single-processor systems; these mechanisms do not generalise to distributed systems.

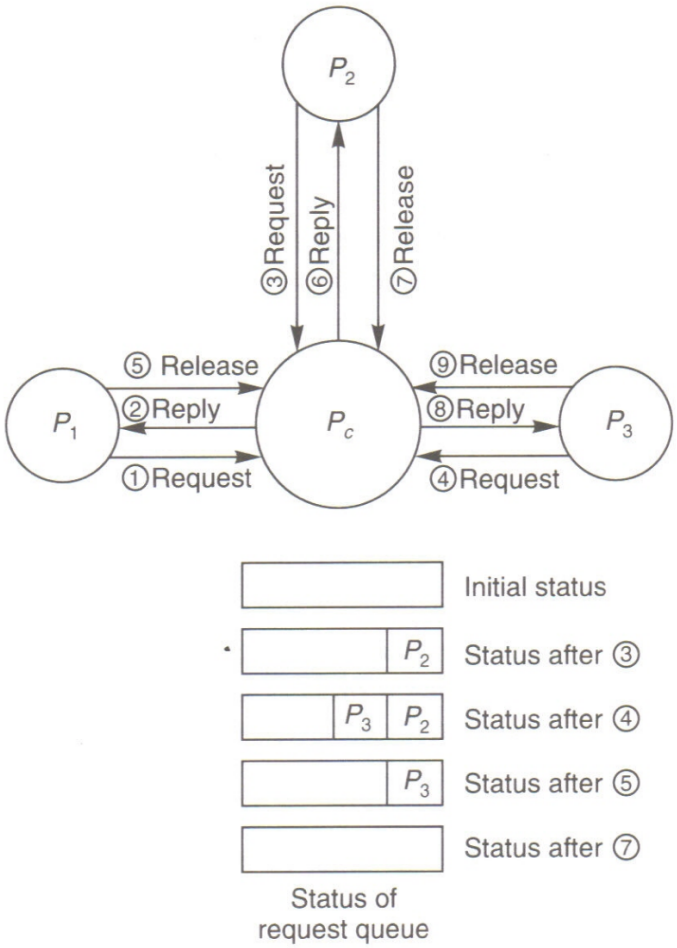


Figure 4.6: Centralised exclusion algorithm.

4.2.1 Centralised Exclusion

This algorithm, depicted in Figure 4.6, simulates the behaviour of mutual exclusion in single processor systems. One process is elected as the coordinator. When a process wants to enter a critical section, it sends a request to the coordinator stating which critical section it wants to enter.

If no other process is currently in that critical section, the coordinator returns a reply granting permission. If a different process is already in the critical section, the coordinator queues the request.

When the process exits the critical section, the process sends a message to the coordinator releasing its exclusive access. The coordinator takes the first item off the queue of deferred requests and sends that process a grant message.

The advantages of this system are:

- Since the service policy is first-come first-serve, it is fair and no process waits forever.
- It is easy to implement.
- It requires only three messages, request, grant, and release, per use of a critical section.

The disadvantages of this system are:

- If the coordinator crashes, the entire system may go down.
- Processes cannot distinguish a dead coordinator from permission denied.
- A single coordinator may become a performance bottleneck.

4.2.2 Distributed Algorithm

Ricart and Agrawala's distributed algorithm, illustrated in Figure 4.7, requires ordering of all events in the system, which can be provided using the Lamport clock.

When a process wants to enter a critical section, the process sends a request message to all other processes. The request message includes the name of the critical section, the process number, and the current time.

The other processes will then receive the request message. One of three things can happen:

- If the process is not in the requested critical section and also has not sent a request message for the same critical section, it returns an OK message to the requesting process.
- If the process is in the critical section, it does not return any response and puts the request to the end of a queue.
- If the process has sent out a request message for the same critical section, it compares the time stamps of the sent request message and the received message. If the time stamp of the received message is smaller than the one of the sent message, the process returns an OK message. If the time stamp of the received message is larger than the one of the sent message, the request message is put into the queue.

The requesting process waits until all processes return OK messages. When the requesting process receives all OK messages, the process enters the critical section.

When a process exits from a critical section, it returns OK messages to all requests in the queue corresponding to the critical section and removes the requests from the queue. Processes enter a critical section in time stamp order using this algorithm.

This algorithm will result in no deadlock or starvation, but is subject to a variety of disadvantages, listed here:

- $2(n-1)$ messages are required to enter a critical section, where n is the number of processes.
- If one of the processes fails, it will not respond to the request. No process will thus be able to enter the critical section.
- In a centralised system, the coordinator is the bottleneck. As all processes send requests to all other processes in this system, *all* processes are bottlenecks.

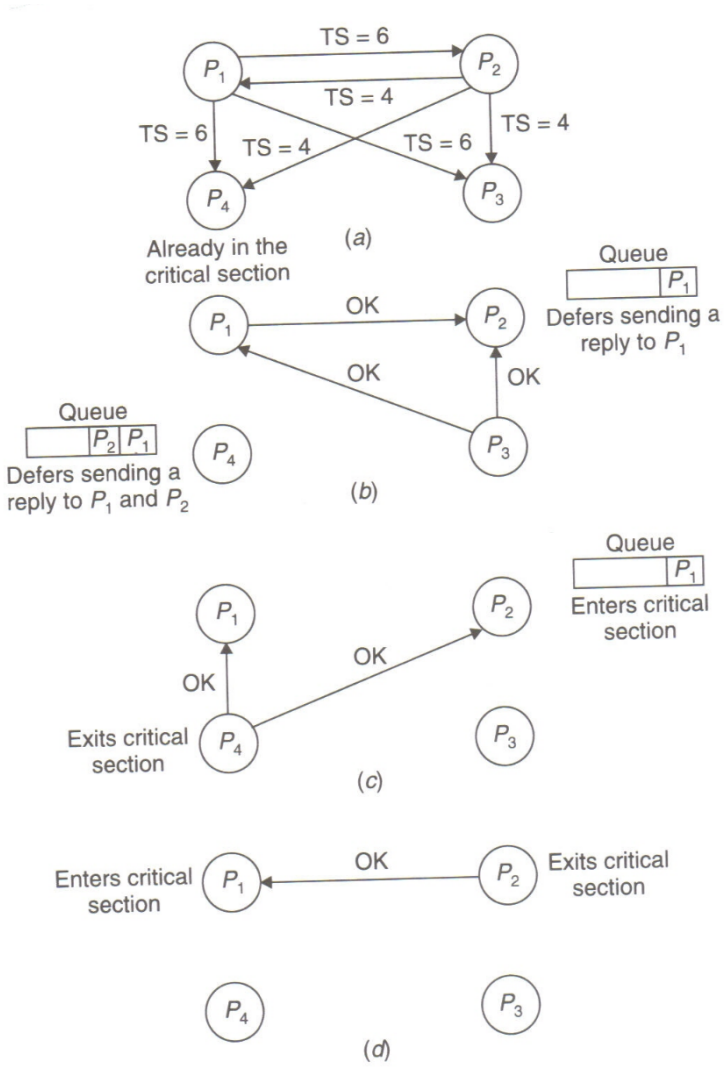


Figure 4.7: Distributed critical section algorithm.

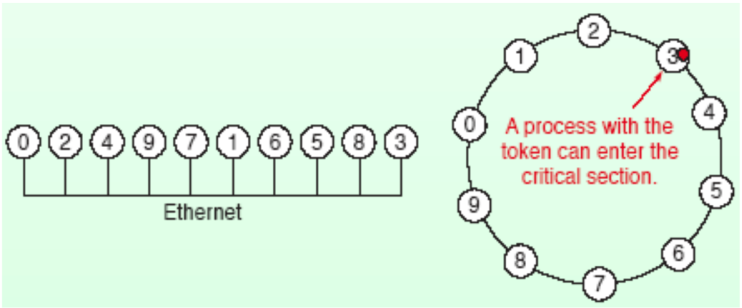


Figure 4.8: Token ring exclusion algorithm.

4.2.3 Token Ring Algorithm

The token ring algorithm, seen in Figure 4.8, constructs a virtual ring by assigning a sequence number to each process. This can be done even if the processes are not physically connected in a ring shape; process 0 receives a token when the ring is initialised.

The token is passed to the process with the next sequence number. A process can enter the critical section only if the process holds the corresponding token. The process passes the token to the next process when it is done. A process passes the received token if it does not need to enter the critical section upon receiving the token.

In this algorithm, the processes do not suffer from starvation. Before attempting to enter a critical section, a process's waiting is bounded by the duration required for all other processes to enter and exit the critical section.

However, if a token is lost for some reason, another token must be generated. Detecting a loss of token is difficult as there is no upper bound for the time required for a token to traverse the ring. If a process crashes, the ring must be reconstructed.

4.2.4 Mutual Exclusion Algorithm Comparison

Based on this, a comparison can be drawn between the three algorithms.

For each entry/exit of a critical section, the centralised algorithm requires 3 messages to be exchanged, the distributed algorithm requires $2(n-1)$ messages to be exchanged, and the ring algorithm requires 2 messages to be exchanged.

Additionally, all three algorithms have reliability concerns. For a centralised system, the coordinator can crash. For a distributed system, any process crashing will break the system. For a token ring system, the loss of the token or a process crashing will result in disruption of the system while the ring is rebuilt.

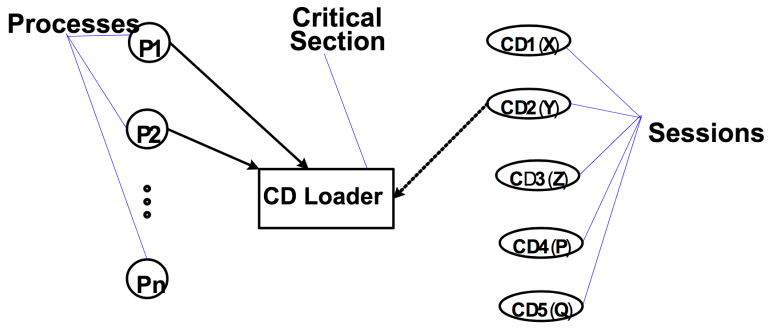


Figure 4.9: Group mutual exclusion problem.

4.2.5 Group Mutual Exclusion

In the group mutual exclusion problem, which generalizes mutual exclusion, processes choose a *session* when they want entry to the Critical Section (CS); processes are allowed to be in the CS simultaneously given that they request the same session. This is illustrated in Figure 4.9.

This problem is applicable to Computer Supported Cooperative Work (CSCW), wireless applications, improving the quality of services of an Internet server (i.e. by grouping requests for the same service), and more.

4.3 Deadlocks

A *deadlock* is a condition where a process cannot proceed because it needs to obtain a resource held by another process and it itself is holding a resource that the other process needs.

There are two types of deadlock to consider:

- **Communication deadlock** occurs when process A is trying to send a message to process B, which is trying to send a message to process C which is trying to send a message to A
- **Resource deadlock** occurs when processes are trying to get exclusive access to devices, files, locks, servers, or other resources.

These will not be differentiated between as a communication channel can be considered to be a resource without loss of generality, which allows a communication deadlock to be treated as a resource deadlock.

There are four conditions that must be met for deadlock to occur:

- **Mutual exclusion**: A resource can be held by at most one process.
- **Hold and wait**: Processes that already hold resources can wait for another resource.
- **Non-preemption**: A resource, once granted, cannot be taken away from a process.
- **Circular wait**: Two or more processes are waiting for resources held by one of the other processes.

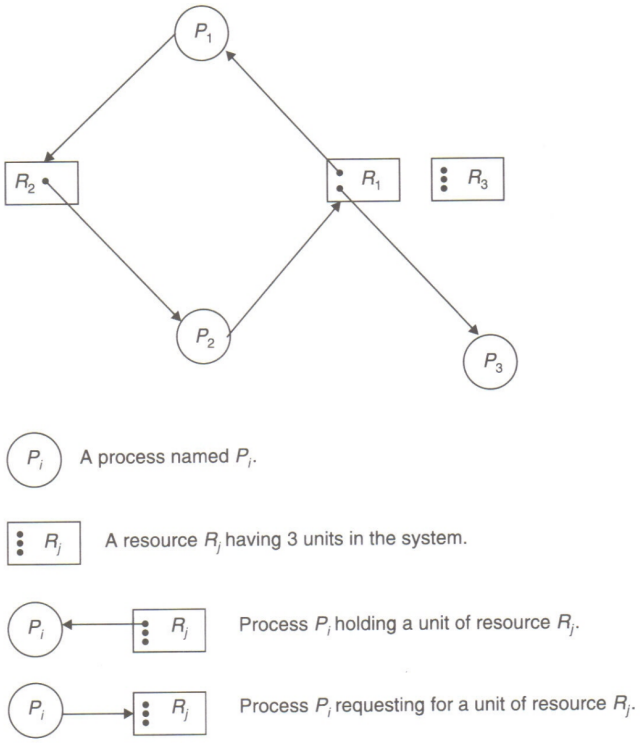


Figure 4.10: Resource allocation graph.

4.3.1 Modelling

A system with potential deadlocks can be modelled using a directed graph called a *resource allocation graph*. In this graph, both the sets of nodes and edges are partitioned into two types, creating the graph elements seen in Figure 4.10:

- **Process nodes**: P_1, P_2, P_3
- **Resource nodes**: R_1, R_2, R_3
- **Assignment edges**: (R_1, P_1), (R_1, P_3)
- **Request edges**: $(P_1, R_2), (P_2, R_1)$

A *cycle* in the resource allocation graph, assuming that there is only one copy of all resources, is a necessary condition for a deadlock to exist; this can be seen in Figure 4.11.

If there are multiple copies of some resources, then a *knot* is required for deadlock to exist. A knot in a directed graph is a collection of vertices and edges with the property that every vertex in the knot has outgoing edges, and all outgoing edges from vertices in the knot terminate at other vertices in the knot. This is illustrated in Figure 4.12.

When there is only a single unit of each type of resource in the resource allocation graph, the graph can be simplified into a wait-for graph. This is done by removing the resource nodes and collapsing the appropriate edges. This is shown in Figure 4.13.

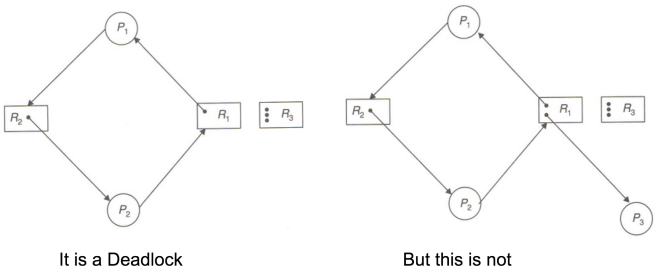


Figure 4.11: A cycle in the resource allocation graph.

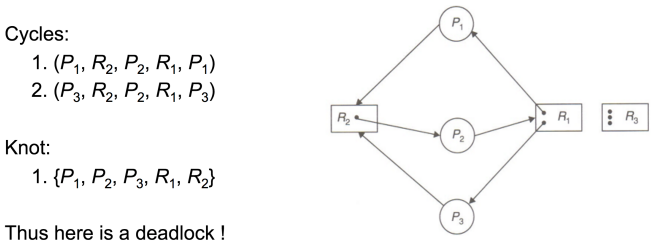


Figure 4.12: Deadlocks through cycles and knots.

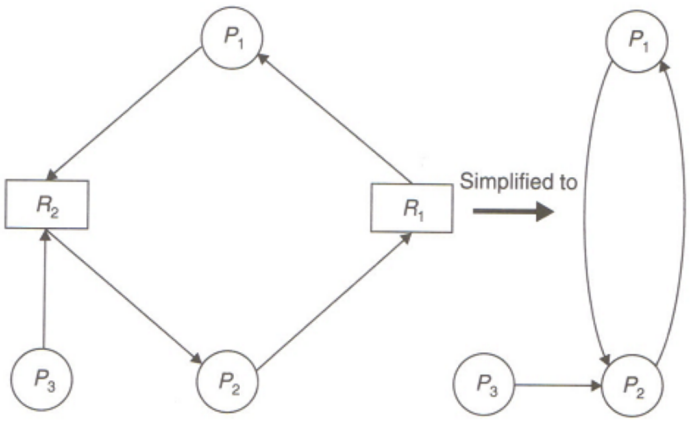


Figure 4.13: Resource allocation graph converted to wait-for graph.

4.4 Handling Deadlocks

4.4.1 Ostrich Algorithm

Ignore the deadlock problem, bury your head in the sand, and pretend that everything is fine. [2]

Figure 4.14: An ostrich.

[2] This doesn't actually work most of the time, y'know?

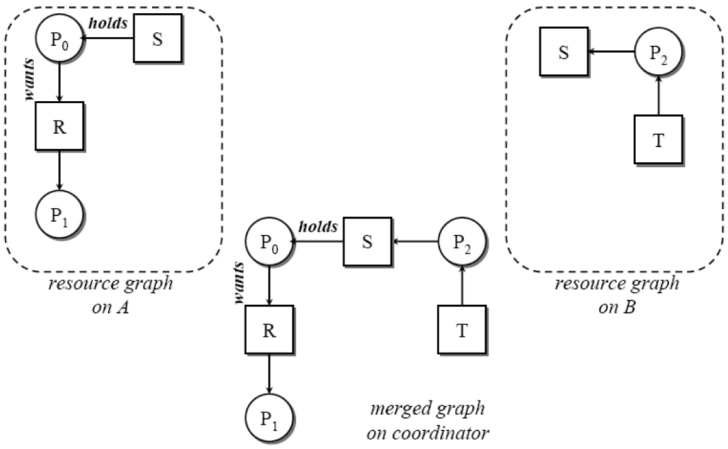


Figure 4.15: Diagram of the centralised deadlock algorithm.

4.4.2 Detection

Preventing or avoiding deadlocks can be difficult. Detecting them is easier. When a deadlock is detected, either kill off one or more processes (annoying users), or if the system is based on atomic transactions abort one or more transactions.

Transactions are generally designed to withstand being aborted. The system is restored to the state it was in before the transaction began, allowing the transaction to start a second time. As resource allocation in the system may be different, the transaction may succeed.

4.4.2.1 Centralised deadlock detection algorithm

The aim of the algorithm is to imitate the non-distributed algorithm through a coordinator, as seen in Figure 4.15. Each machine maintains a resource graph for its processes and resources. A central coordinator maintains a graph for the entire system, and a message can be sent to the coordinator each time an arc is added or deleted by a machine. The list of arc adds/deletes can be sent periodically.

However, this algorithm is susceptible to *false deadlock*. Examine Figure 4.16. If P_1 releases a resource R and then asks for a resource T from P_2, the coordinator may receive the message about waiting prior to the message about releasing. The coordinator will then end up construct a graph with a cycle, resulting in a detection of a deadlock.

This can be mitigated by enforcing global time ordering on all machines, or by using a reliable communication method to ask each machine whether it has any release messages prior to making a deadlock detection claim.

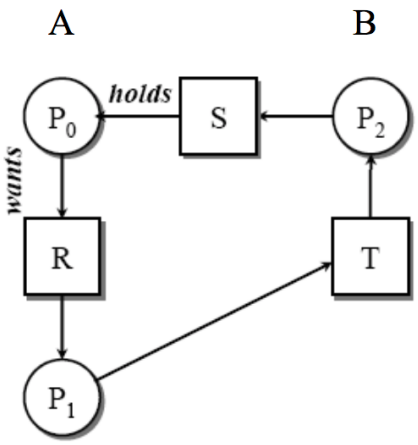


Figure 4.16: False deadlock in the centralised deadlock detection algorithm.

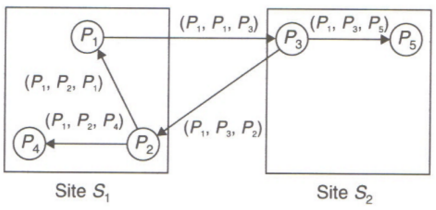
4.4.2.2 Chandy-Misra-Haas algorithm

The CMH algorithm is a distributed algorithm for deadlock detection, illustrated in Figure 4.17. Processes can request multiple resources at once. Some processes wait for local resources. Some processes wait for resources on other machines. The algorithm is invoked when a process has to wait for a resource.

When a process has to wait for a resource, a probe message is generated. This message contains 3 fields: process that was just blocked, the process sending the message, and the process to whom it is being sent.

When the probe message arrives, the recipient checks to see if it is waiting for any processes. If it is not, the message is ignored. If it is, the message is updated; the second field is replaced with its own process number, and the third field is replaced with the number of the process it is waiting for. The message is then sent to each process on which the recipient is blocked o.

If a message goes all the way around and comes back to the original sender, a cycle exists, and thus deadlock has occurred.



- Suppose,
  - P_1 gets blocked by P_3
  - P_3 gets blocked by P_2 and P_5
  - P_2 gets blocked by P_1 and P_4

Figure 4.17: Chandy-Misra-Haas algorithm diagram.

4.4.2.3 Recovery After Detection

After deadlock has been detected, several measures can be used. The operator can be asked to intervene, processes can be autonomously terminated, or process state can be rolled back. These each have their own advantages and disadvantages, especially with regards to performance and minimisation of frustration for the operator.

Additionally, recovery may involve other issues, such as minimisation of recovery cost and the prevention of starvation. [3]

[3] These are left as an exercise to the reader to contemplate.

Chapter 5

Distributed Coordination

5.1 Election Algorithms

Distributed algorithms often elect one process to act as a coordinator or a server. We assume each process is assigned a unique process number, that a process knows all process numbers, and a process does not know which process numbers are alive.

The election process proceeds as follows:

- Select the maximum process number among active processes.
- Appoint the process the coordinator.
- All the processes must agree upon this decision.

5.1.1 Bully Algorithm (Garcia-Molina)

When a process P notices that the coordinator is no longer responding to requests, it initiates an election.

P sends an *Election* message to all processes with higher numbers. If one of the higher numbered answers, that process becomes the new leader. If no one responds, P wins the election and becomes coordinator. Note that timeout overhead and delayed responses are a consideration.

At any moment, a process can get an *Election* message from one of its lower-numbered colleagues. The receiver sends an *OK* message back to the sender. The receiver then holds an election, unless it is already holding one.

Eventually, all processes give up but one, and that one is the new coordinator. It announces its victory by sending *Coordinator* messages to all processes. If a process that was previously down comes back up, it holds an election. If it happens to be the highest-numbered process currently running, it will win the election and take over the coordinator's job. The highest numbered process always wins, hence the name "bully algorithm."

5.1.2 Ring Algorithm

Processes form a logical ring. Each process knows which process comes after it. If a process finds the coordinator dead, it sends an *Election* message which contains its own process number to its successor in the ring. If the successor is down, the message is sent to the next process along the ring.

When receiving an *Election* message, each process appends its process number to the message and then forwards it to the successor. When the message returns to the process that originally sent the message, the process changes the message type from a *Election* message to a *Coordinator* message and circulates the message again.

The purpose of the *Coordinator* message is to inform all processes that the highest-numbered process is the new coordinator and the processes contained in the message are the members of the ring.

5.2 Distributed Transactions

5.2.1 Atomic Transactions

An atomic transaction provides a view where either a set of operations have all been completed or none of them have been completed.

First, a process declares to all other processes (involved in a transaction) that it is starting the transaction. The processes exchange information and perform their specific operations. The initiator announces that it wants all the other processes to *commit* to all the operations done up to that point.

If all processes agree to commit, the results are made permanent. If at least one of the processes refuse, all the states - usually opened files/databases (on all machines) - are reverted to the point prior to starting the transaction.

A set of assumptions must be made when working with distributed transactions. These are:

- A system consists of independent processes where each process may fail at random.
- No communication errors may be encountered.
- *Stable storage* is available; this guarantees no data is lost unless the storage is physically lost by disasters such as floods or earthquakes.

  Note: Data in RAM disappears when the power is turned off. Data in a disk becomes inaccessible if the head crashes. As a result, these are not stable storage.

Stable storage can be implemented with a pair of disks, as seen in Figure 5.1. Each block on drive 2 is a copy of the corresponding block on drive 1. When a block is updated, the block on drive 1 is first updated and verified. Then, the same block on drive 2 is updated and verified.

If the system crashes after the update of drive 1 but before the update of drive 2, the blocks on the two drives are compared and the blocks on drive 1 are copied to drive 2 after system reboot. If a checksum error is encountered because of a bad block, the block is copied from the disk without an error.

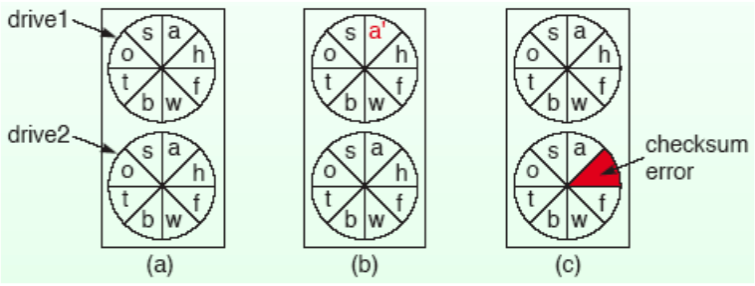


Figure 5.1: Stable storage using a pair of disks.

5.2.2 Primitives

Programming a transaction requires special primitives that must be supplied by the OS or by the programming language. These primitives are implemented as either system calls, library functions, or constructs of a programming language. Examples of the primitives include:

- **Begin Transaction**: Specifies the beginning of a transaction.
- **End Transaction**: Specifies the end of a transaction, and tries to commit the results.
- **Abort Transaction**: Abort the transaction and restore the old state.
- **Read**: Reads data from a file.
- **Write**: Writes data to a file.

Begin Transaction and End Transaction specify the scope of a transaction. The operations delimited by the two operations form the transaction.

5.2.3 Properties

Any transaction must abide by these four essential properties:

- **Atomicity**: A transaction happens indivisibly to the outside world. (Either all operations of the transaction are completed or none are completed.) The other process cannot observe the intermediate states of the transaction.
- **Consistency**: A transaction does not violate system invariants. For example, an invariant in a banking system may be the law of conversation regarding the total amount of money - that is, the sum of money in the source and destination accounts after any transfer is the same as the sum before the transfer (although conservation may be violated during the transfer).
- **Isolation**: Even if more than one transaction is running simultaneously, the final result is the same as the result of running the transactions sequentially in some order. This property is also called *serializable*.
- **Durability**: Once a transaction commits, changes are permanent. No failure after the commit can undo the results or cause these to be lost.

5.2.4 Nested Transactions

Transactions may contain their own sub-transactions; these are referred to as *nested transactions*. They are subject to a potential problem:

- Assume a transaction starts several sub-transactions in parallel.
- One of these sub-transactions commits and makes its results visible to the parent transaction.
- For some reason, the parent transaction is aborted and the entire system needs to be restored to its original state.
- Consequently, the results of the sub-transaction that committed must be undone.
- Undo-ing a committed transaction is a violation of transaction laws.

To alleviate this, we define characteristics for the nested transactions:

- Permanence of a sub-transaction is only valid within the world of its direct parent and is otherwise invalid/irrelevant in the context of further ancestors.
- Permanence of a transaction is only valid for the outermost transaction (that is, the top-level transaction that is itself not nested in another transaction).

To facilitate this, one method is to have each sub-transaction make a private copy of all objects in the system. If a sub-transaction commits, its private copy of the mutated resources replaces its parent's copy. This limits any potential mutation to the scope of the parent transaction, so that the outside world is not affected until the parent transaction commits.

5.2.5 Implementation of atomic transactions

Implementing atomic transactions involves the hiding of operations within a transaction from outer processes, making the results visible only after commit and restoring the old state if the transaction is aborted.

There are two ways to restore the old state: *private workspaces*, and *write-ahead logs*. These are discussed in more detail below.

5.2.5.1 Private Workspace

When a process starts a transaction, it is given a private workspace containing all the objects including data and files. Read and write operations are performed within the private workspace. The data within the private workspace is written back when the transaction commits.

The problem with this technique is that the cost of copying every object to a private workspace is high. The following optimisation is possible: if a process only reads an object, there is no need for a private copy; if a process updates an object, a copy of the object is made in the private workspace.

If we use indices to objects, we can reduce the number of copy operations. When an object is to be updated, a copy of the index of each object is made in the private workspace. When an

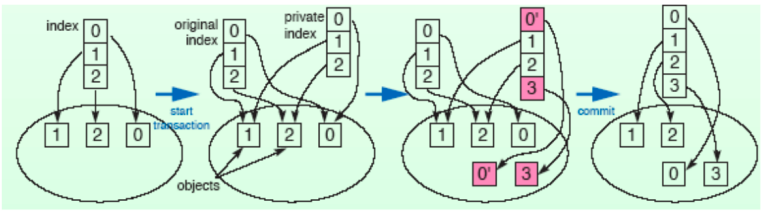


Figure 5.2: Optimising the private workspace technique using indices.

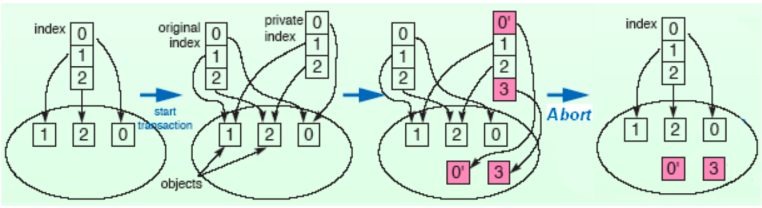


Figure 5.3: Aborted optimised private workspace tranction.

object is mutated, its private index is updated. On commit, the private index is copied to the parent's index. This is illustrated in Figure 5.2.

If the transaction aborts, the private copies of the objects and indices are discarded, as shown in Figure 5.3.

5.2.5.2 Write-ahead Log

Before any changes to objects are made, a record is written to a *write-ahead log* on stable storage. The log contains the following information: which transaction is making the change, which object is being changed, and what the old and new values are.

The change is made to the object only after the log has been written successfully. If the transaction committed, a commit record is written to the log (data has already been written). If the transaction aborts, the log can be used to rollback to the original state. The log can be used for recovering from crashes.

5.2.5.3 Two-Phase Commit Protocol

The action of committing a transaction must be done instantaneously and indivisibly. In a distributed system, the commit requires the cooperation of multiple processes that may be on different machines. Two-phase commit protocol is the most widely used protocol to implement an atomic commit.

One process functions as the coordinator. The other participating processes are subordinates (Cohorts). This is shown in Figure 5.4.

The coordinator writes an entry *prepare* in the log on stable storage and sends the other processes involved (the subordinates) a message telling them to prepare. When a subordinate receives the

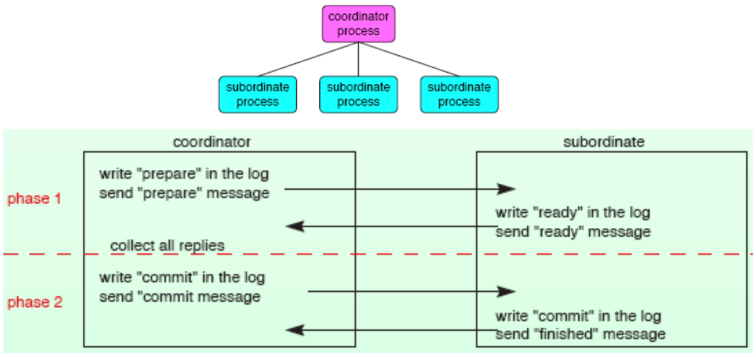


Figure 5.4: Two-phase commit protocol.

message, it checks to see if it is ready to commit, makes a log entry, and sends back its decision.

After collecting all the responses, if all the processes are ready to commit, the transaction is committed. Otherwise, the transaction is aborted. The coordinator writes a log entry and then sends a message to each subordinate informing it of the decision. The action of the coordinator writing the commit to the log is equivalent to committing the transaction; no rolling back can occur afterwards, regardless of what may happen.

This protocol is highly resilient in the face of crashes because it uses stable storage. If the coordinator crashes after writing a *prepare* or *commit* message, it can still resume from its previous state.

If a subordinate crashes after writing a *ready* or *commit* log entry, it can resume from a *ready* or *finished* message upon restarting.

5.2.5.4 Three-Phase Commit Protocol

This protocol assumes that each site uses the write-ahead log protocol, and that at most one site can fail during the execution of the transaction.

Before the commit protocol begins, all the sites are in state q. If the coordinator fails while in state $q1$, all the cohorts perform the *timeout transition*, thus aborting the transition. Upon recovery, the coordinator performs the *failure transition*. [1]

5.3 Concurrency Control

When several transactions run simultaneously in different processes, we must guarantee the final result is the same as the result of running the transactions sequentially in some order. That is, we need a mechanism to guarantee isolation or serializability. This mechanism is *concurrency control*.

There are three typical algorithms: using locks, using timestamps, and being optimistic.

[1] No, I'm not entirely sure what this protocol is or what it has to offer, either.

Table 5.1: Compatibility between lock modes.

	Locking Mode		
Locking Mode	Unlocked	Shared	Exclusive
Shared	✓	✓	✗
Exclusive	✓	✗	✗

5.3.1 Locks

This is the oldest and most widely used concurrency control algorithm. When a process needs to access shared resource as part of a transaction, it first locks the resource. If the resource is locked by another process, the requesting process waits until the lock is released.

The basic scheme is overly restrictive. The algorithm can be improved by distinguishing read locks from write locks. Data that only needs to be read (referenced) is locked in shared mode. Data that needs to be updated (modified) is locked in exclusive mode.

Lock modes have the following compatibility:

- If a resource is not locked or locked in shared mode, a transaction can lock the resource in shared mode.
- A transaction can lock a resource in exclusive mode only if the resource is not locked.

This is shown in Table 5.1.

If two transactions are trying to lock the same resource in incompatible modes, the two transactions are in *conflict*. A conflict relationship can be a shared-exclusive conflict (or read-write conflict) or a exclusive-exclusive conflict (or write-write conflict).

5.3.1.1 Two-Phase Locking

Using locks does not necessarily guarantee serializability. Only transactions executed concurrently in accordance with the following steps will be serializable:

1. A transaction locks a shared resource before accessing the resource and releases the lock after using it.
2. Locking is granted according to the compatibility constraint.
3. A transaction does not acquire any locks after releasing a lock.

The use of locks in a manner that satisfies the above conditions is referred to as the *two-phase locking protocol*.

A transaction acquires all necessary locks in the first phase; the *growing phase*. The transaction releases all the locks in the second phase; the *shrinking phase*. Two-phase locking is sufficient to guarantee serializability.

Suppose a transaction aborts after releasing some of the locks. If other transactions have used resources protected by these released locks, these transactions must also be aborted; this is referred to as a *cascading abort*, as seen in Figure 5.5.

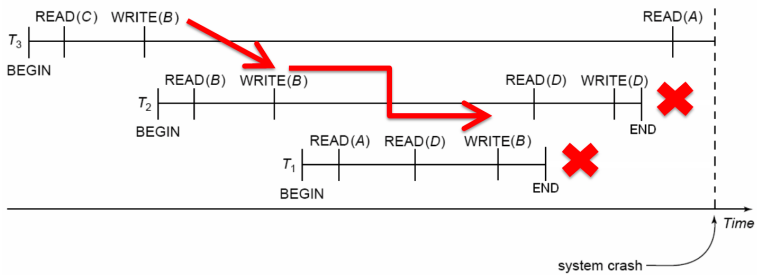


Figure 5.5: Cascading abort in two-phase locking.

To avoid cascading abort, many systems use *strict two-phase locking* in which transactions do not release any locks until the transaction is fully committed. The sequence for strict two-phase locking follows:

1. Begin transaction
2. Acquire locks before reading or writing resources
3. Commit changes
4. Release all locks
5. End transaction

Strict two-phase locking has the following advantages:

- A transaction always reads a value written by a committed transaction. A transaction never has to be aborted - the cascading abort never happens.
- All lock acquisitions and releases can be handled by the system without the transaction being aware. Locks are acquired whenever data is accessed and released when the transaction has finished.

However, two-phase locking can cause deadlocks to occur. This will happen when two processes try to acquire the same pair of locks but in reverse order to each other.

5.3.1.2 Granularity

The *lock granularity* refers to the size of resources that can be locked by a single lock. Finer granularity allows for more concurrent processing, but requires a greater quantity of locks.

For example, consider locking by file and locking by record contained in a file. Suppose that two transactions access different records in the same file. If the record is the unit of lock, two transactions can access the data simultaneously. On the other hand, if the file is the unit of lock, the two transactions cannot access the data simultaneously.

5.3.2 Timestamps

Assign each transaction a timestamp when the transaction begins. These timestamps must be unique; this can be ensured using Lamport's algorithm.

Each resource has a *read timestamp* and a *write timestamp*. A read timestamp is the timestamp of the transaction which read the resource most recently. A write timestamp is the timestamp of the transaction which updated the resource most recently. However, note that read timestamps and write timestamps are *not* the actual time values for when the data items were read or written.

Let $\texttt{TRD}(x)$ be the read timestamp of resource x. Let $\texttt{TWR}(x)$ be the write timestamp of resource x.

When a transaction with timestamp T tries to read resource x,

- $T < \texttt{TWR}(x)$ implies that the transaction is attempting to read a x that has already been overwritten. As a result, the request is rejected and the transaction is aborted.
- $T \geq \texttt{TWR}(x)$ implies that the transaction is attempting to read a value after its last write, so the transaction is allowed to read the resource.

When a transaction with timestamp T tries to update resource x,

- $T < \texttt{TWR}(x)$ implies that the transaction is attempting to write an old x. As a result, the request is rejected and the transaction is aborted.
- $T < \texttt{TRD}(x)$ implies that the transaction is attempting to write a x that was previously needed, but is no longer needed. As a result, the request is rejected and the transaction is aborted.
- Otherwise, the transaction is allowed to update the resource.

The timestamp-ordering protocol guarantees serializability since all the edges in the precedence graph originate from a transaction with a smaller timestamp and terminate at a transaction with a larger timestamp. This prevents cycles in the precedence graph, ensuring freedom from deadlock as no transaction will ever wait on a resource. However, the execution schedule may not be recoverable.

5.3.3 Optimistic Concurrency Control

The crux of this algorithm is to run transactions and to be optimistic about transaction conflicts. At the end of a transaction (i.e. during a commit), the system determines whether any conflicts exist. If no conflicts are detected, the local copies are written to the real resource. Otherwise, the transaction is aborted. This requires a three-phase transaction; the first phase is *read & private write*, the second phase is *validate*, and the third phase is *write*.

The conflict check is completed by examining whether the resources used by the committing transaction were updated by other transactions during the execution of the committing transaction. To make this check possible, the system must keep track of the resources used by all transactions.

Each transaction T_i is assigned a timestamp $TS(T_i)$ at the beginning of the validation phase. Each pair of transactions T_i and T_j such that $TS(T_i) < TS(T_j)$ is then checked for one of the following validation conditions:

1. T_i completes all three phases prior to T_j beginning.
2. T_i completes before T_j starts its write phase, and T_i does not write any object read by T_j.
3. T_i completes its read phase before T_j completes its read phase, and T_i does not write any objects that are read or written by T_j.

Optimistic concurrency control is designed with the assumption that conflicts are rare and we do not need to abort transactions often. Updates are performed on local copies, which fits well with the private workspaces technique. It is advantageous as it allows maximum parallel execution since transactions never have to wait and deadlocks never occur.

However, the disadvantage of the system is that a transaction may be aborted at the very end since the conflict check is done at the commit point. All operations of the aborted transaction must be done again from the start. If the initial assumption with regards to low conflict rates is incorrect, optimistic concurrency control is a poor fit for the problem.

Chapter 6

Failure, Distributed Consensus

6.1 Failure

A dependable system must satisfy four properties:

- **Availability**: The system is ready to be used immediately at any point in time.
- **Reliability**: The system can run continuously without failure.
- **Safety**: If the system (temporarily) fails to operate correctly, nothing catastrophic will happen.
- **Maintainability**: The system can be repaired relatively easily.

Building a dependable system comes down to controlling failure and faults.

First, it is important to define what failure *is*:

- **Failure**: a system fails when it fails to meet its promises or cannot provide its services in the specified manner
- **Error**: part of the system state that leads to failure (i.e., it differs from its intended value)
- **Fault**: the cause of an error (results from design errors, manufacturing faults, deterioration, or external disturbance)

Note that a failure can be recursive:

- Failure may be initiated by a mechanical fault
- Manufacturing fault leads to disk failure
- Disk failure is a fault that leads to database failure
- Database failure is a fault that leads to email service failure

In a *total failure*, all components in a system fail; this is typical for a non-distributed system. In a *partial failure*, one or more (but not all) components in a distributed system fail - that is, some components are affected, but other components are completely unaffected; this is treated as a fault for the whole system.

6.1.1 Classifications

Distributed systems are designed at the process level, so we consider failures that are visible at the process level first.

6.1.1.1 Crash Failure

A process undergoes crash failure when it permanently ceases to execute its actions. This is an irreversible change.

In an asynchronous model, crash failures cannot be detected with total certainty, since there is no lower bound of the speed at which a process can execute its actions.

In a synchronous system where processor speed and channel delays are bounded, crash failures can be detected using timeouts.

6.1.1.2 Omission Failure

If the receiver does not receive one or more of the messages sent by the transmitter, an omission failure occurs. For wireless networks, this can occur when a collision occurs in the MAC layer or when a receiving node moves out of range.

6.1.1.3 Transient Failure

A transient failure can disturb the state of processes in an arbitrary way. The agent inducing this problem may be momentarily active but it can have a lasting effect on the global state. Examples include a power surge, mechanical shock, or lightning.

6.1.1.4 Byzantine Failure

Byzantine failures represent the weakest of all of the failure models. It allows every conceivable form of erroneous behaviour. The term alludes to uncertainty and was first proposed by Pease et al.

Assume that process i forwards the value x of a local variable to each of its neighbours. The following inconsistencies may occur:

- two distinct neighbours j and k receive values x and y, where $x \neq y$.
- one or more neighbours do not receive any data from i.
- every neighbour receives a value z where $z \neq x$.

Possible causes of Byzantine failures include:

- total or partial breakdown of a link joining i with one of its neighbours.
- software problems in process i.

- hardware synchronization problems assume that every neighbour is connected to the same bus, and is attempting to read the same copy sent out by i. However, since the clocks are not perfectly synchronized, they may attempt to read the value of x at the same local time but at different absolute times. If the value of x varies with time, then each neighbour of i may read different values of x.

6.1.1.5 Software Failure

Software failure is failure originating from within the software. Note that many of the other types of failure, such as crash, omission, transient and Byzantine failures, originate from software issues. For example, a poorly designed loop that does not terminate can appear as a crash failure to external observers. Similarly, an inadequate policy in routing software can cause packets to drop, resulting in omission failure.

Causes of software failure include:

- **Coding or human error**: As an example, the program may have used the wrong physical parameters. On September 23, 1999, NASA lost the Mars Climate Orbiter spacecraft (valued at 327.6 million USD) to this class of error; one team used metric units while another team used imperial units, resulting in the spacecraft entering the Mars atmosphere at a much lower altitude than expected and promptly disintegrating. This trajectory discrepancy is illustrated in Figure 6.1.
- **Software design error**: The software was designed in such a way that errors can arise in its functionality. The Mars Pathfinder mission landed without issues on the Martian surface on July 4, 1997. However, its ability to communicate failed due to a design flaw in the real-time embedded software kernel VxWorks; this flaw was later determined to have originated from a priority inversion scenario:
  - A low priority task `LP` locks file `F`.
  - A high priority task `HP` is scheduled next, which also needs to lock file `F`.
  - A medium priority `MP` task (with high CPU requirement) becomes ready to run.
  - `MP` is the highest priority unblocked task; as a result, it is allowed to run, and consumes all CPU.
  - `LP` has no CPU available to it and thus idles indefinitely. However, even though `HP` is higher priority than `MP`, `MP` is still being run, and `HP` cannot be scheduled as `LP` still has the lock on `F`. As a result, *priority inversion* occurs: `HP` is indirectly prevented from running by `MP`.
- **Memory leaks**: Processes fail to fully free up the physical memory that has been allocated to them. This effectively reduces the size of available physical memory over time. When the available memory falls below the minimum requirement by the system, a crash becomes inevitable.
- **Inadequacy of specification**: An issue with the specification may have long-lasting ramifications that are only apparent when the bounds of the specification are exceeded. The canonical example is the Y2K problem, in which years in dates were only stored with two digits; this became problematic with the turn of the century where the "year" would roll over from 99 to 00.

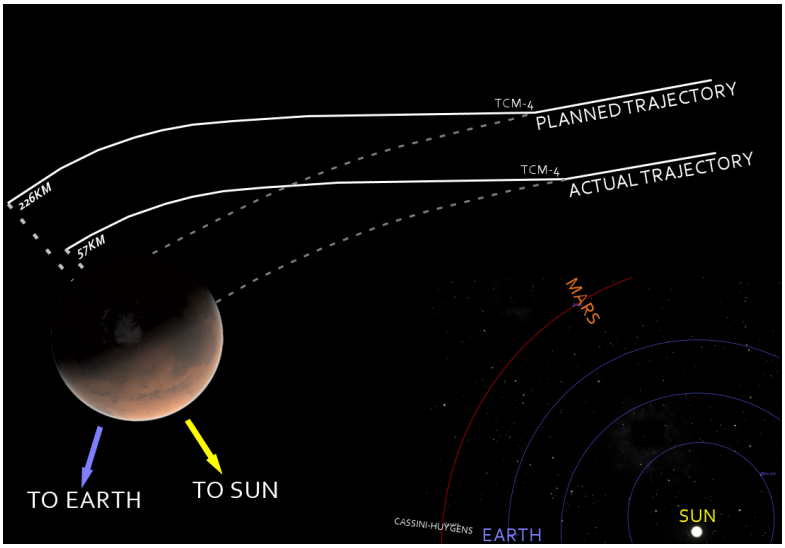


Figure 6.1: The incorrect trajectory calculated for the Mars Climate Orbiter. Courtesy of Wikipedia.

6.1.1.6 Temporal Failure

Real-time systems require actions to be completed within a specific time frame. When this time limit is not met, a temporal failure occurs.

6.1.1.7 Security Failure

Viruses and other malicious software may lead to unexpected behaviour, which may manifest as a system fault.

6.1.1.8 Human Error

Human errors can also play a large part in system failure. Examples include:

- In November 1988, a significant portion of the long distance service along the East Coast of the United States of America was disrupted when a construction crew accidentally detached a major fibre optic cable in New Jersey. As a result, 3,500,000 call attempts were blocked.
- On September 17, 1991 AT&T technicians in New York failed to respond to an activated alarm for six hours as they were attending a seminar on warning systems. The resulting power failure blocked nearly 5 million domestic and international calls and paralysed air travel throughout the Northeast, causing nearly 1,170 flights to be cancelled or delayed.

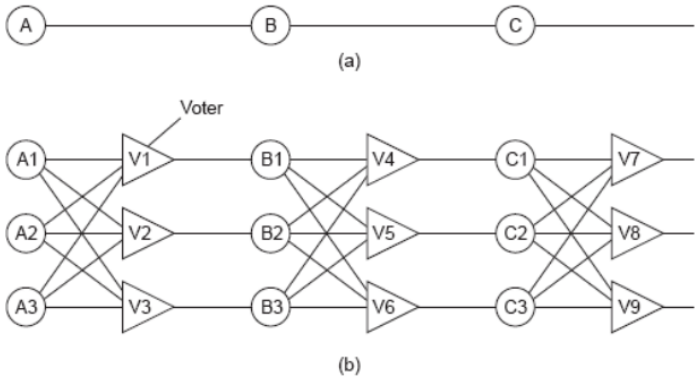


Figure 6.2: Failure masking comparison.

6.1.2 Fault-Tolerant Systems

We designate a system that does not tolerate failures as a fault-tolerant system. In such systems, the occurrence of a fault violates *liveness* and *safety* properties. Safety properties specify that "something bad never happens"; doing nothing easily fulfils a safety property as this will never lead to a "bad" situation. Safety properties are complemented by liveness properties, which assert that "something good" will eventually happen.

The first known fault-tolerant computer was SAPO, built in 1951 in Czechoslovakia by Antonin Svoboda. Most of the development in Long Life, No Maintenance (LLNM) computing was undertaken by NASA during the 1960s in preparation for Project Apollo and other research prospects. NASA's first machine went into a space observatory, and their second attempt, the JSTAR computer, was used in Voyager. This computer had a backup of memory arrays to facilitate the use of memory recovery methods; this led to the name of the JPL Self-Testing-And-Repairing computer. It could detect its own errors and fix them or bring up redundant modules as needed.

6.1.2.1 Masking Tolerance

Let P be the set of configurations for the fault-tolerant system. Given a set of fault actions F, the fault span Q corresponds to the largest set of configurations that the system can support. In a failure-masking system, when a fault F is masked, its occurrence has no impact on the application (i.e. $P = Q$).

Masking tolerance is important in many safety-critical applications where the failure can endanger human life or cause massive loss of properties; an aircraft must be able to fly even if one of its engines malfunctions. Masking tolerance preserve both safety and liveness properties of the original system.

To implement failure masking, redundancy must be introduced. This includes information redundancy, time redundancy, and physical redundancy. This is illustrated in Figure 6.2.

6.1.3 Non-Masking Tolerance

In non-masking fault tolerance, faults may temporarily affect and violate the safety property (i.e. $P \subset Q$). However, liveness is not compromised, and eventually normal behaviour is restored. As an example, consider watching a movie where the backend server crashes; the system can automatically restore the service by switching to a standby proxy server.

Checkpointing and stabilisation represent two opposing scenarios in non-masking tolerance.

- *Checkpointing* relies on history; recovery is achieved by retrieving lost computation.
- *Stabilisation* is history-insensitive and does not care about lost computation as long as eventual recovery is guaranteed.

6.1.4 Fail-Safe Tolerance

Certain faulty configurations do not affect the application in an adverse way and are therefore considered harmless. A fail-safe system relaxes the tolerance requirement by aiming to avoid the subset of faulty configurations that will have catastrophic consequences (not withstanding the failure), and allowing the remaining faulty configurations to occur.

As an example, at a four-way traffic crossing, collision is possible if the lights are green in both directions. However, if the lights are red, traffic may stall but there will be no catastrophic side effects.

6.1.5 Graceful Degradation

There are systems that neither mask nor fully recover from the effect of failures, but exhibit a degraded behaviour that falls short of normal behaviour, but is still considered acceptable. The notion of acceptability is highly subjective and entirely dependent on the user running the application.

Examples include:

- While routing a message between two points in a network, a program computes the shortest path. In the presence of a failure, the program returning a valid path that is not the shortest possible may be considered acceptable.
- An operating system may switch to a safe mode where users cannot create or modify files, but can read the files that already exist.

6.2 Distributed Consensus

Distributed consensus - that is, consensus amongst a distributed system - is a necessary requirement for many scenarios; it is easier to achieve in the absence of failures. Some of these scenarios are explored below:

1. The leader election problem in a network of processes. Each process begins with an initial proposal for leadership. At the end, one of the candidates is elected as a leader; every process must agree on this.

2. Fund transfer
3. Clock synchronisation

The distributed consensus problem is formulated as such: a distributed system contains n processes, and each process has an initial value in a mutually agreed domain. The challenge is to design a failure-resilient algorithm that allows processes to reach an irrevocable decision that fulfills the following conditions:

- **Termination**: Every (non-faulty) process must eventually come to a decision.
- **Agreement**: The final decision of every (non-faulty) process must be identical.
- **Validity**: If every (non-faulty) process begins with the same initial value v, their final decision must be v.

If there is no failure, then reaching an agreement is trivial. Reaching consensus, however, becomes surprisingly difficult when one or more members fail to execute actions. In the formulation of the problem, we assume that at most k members ($k > 0$) can fail; an important finding by Fischer et al. is that in a fully asynchronous system, it is impossible to reach consensus even if $k = 1$.

In discussion of distributed consensus, it is important to discuss *bivalence* and *univalence*. A decision state is bivalent if there exist at least two distinct executions leading to two distinct decision values (e.g. 0 or 1). On the other hand, a state from which only one decision value can be reached is called a univalent state. Univalent state states can be either 0-valent or 1-valent.

Consider a best-of-five-sets tennis match between A and B. If the score is 6-3, 6-4 in favour of A, the decision state is bivalent, since anyone can win at this point. However, if the score becomes 6-3, 6-4, 7-6 in favour of A, then the state becomes univalent.

6.2.1 The Byzantine Generals' Problem

The Byzantine Generals' Problem, as described by Wikipedia:

> [...] a group of generals, each commanding a portion of the Byzantine army, encircle a city. These generals wish to formulate a plan for attacking the city. In its simplest form, the generals must decide only whether to attack or retreat. Some generals may prefer to attack, while others prefer to retreat. The important thing is that every general agree on a common decision, for a halfhearted attack by a few generals would become a rout, and would be worse than either a coordinated attack or a coordinated retreat.
>
> The problem is complicated by the presence of treacherous generals who may not only cast a vote for a suboptimal strategy, they may do so selectively. For instance, if nine generals are voting, four of whom support attacking while four others are in favor of retreat, the ninth general may send a vote of retreat to those generals in favor of retreat, and a vote of attack to the rest. Those who received a retreat vote from the ninth general will retreat, while the rest will attack (which may not go well for the attackers). The problem is complicated further by the generals being physically separated and having to send their votes via messengers who may fail to deliver votes or may forge false votes.

Leslie Lamport showed by proof that for a system of $n + 1$ nodes, there cannot be more than $\frac{n}{3}$ faulty nodes if distributed consensus is desired. This can be reformulated in the terms of the

Byzantine Generals' Problem: there must be more than $3m$ troops in an army with up to m traitors to launch a concerted attack.

Chapter 7

Parallel Computing

Parallel computing is a form of computation in which many instructions are carried out simultaneously (see Figure 7.1). It operates on the principle that large problems can often be divided into smaller ones, which are then solved concurrently. There are several different forms of parallel computing: bit-level parallelism, instruction-level parallelism, data parallelism, and task parallelism.

Contemporary computer applications require the processing of large amounts of data in sophisticated ways. Examples include:

- parallel databases, data mining
- oil exploration
- web search engines, web based business services
- computer-aided diagnosis in medicine
- management of national and multi-national corporations
- advanced graphics and virtual reality, particularly in the entertainment industry
- networked video and multi-media technologies
- collaborative work environments

Ultimately, parallel computing is an attempt to minimise computational time and effort.

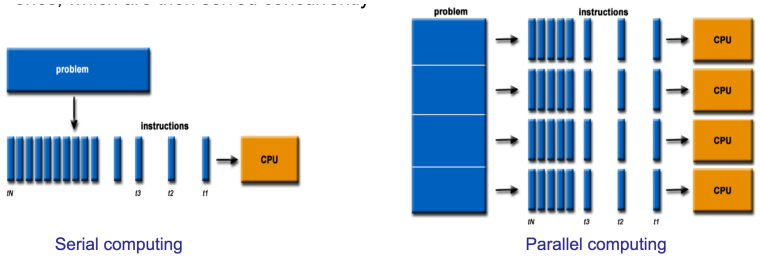


Figure 7.1: Serial computing vs parallel computing.

There are different ways to classify parallel computers. One of the more widely used classifications, in use since 1966, is Flynn's taxonomy. Flynn's taxonomy distinguishes multi-processor computer architectures according to two independent dimensions of Instruction and Data. Each of these dimensions can have only one of two possible states: Single or Multiple. This lends itself to the four possible classifications, shown in Figure 7.2:

- **Single Instruction, Single Data (SISD)**: A serial (non-parallel) computer; executes deterministically. This is the oldest and until recently, the most prevalent form of computer. Examples: most PCs, single CPU workstations and mainframes.

  Single instruction: Only one instruction stream is being acted on by the CPU during any one clock cycle.

  Single data: Only one data stream is being used as input during any one clock cycle.

- **Single Instruction, Multiple Data (SIMD)**: A type of parallel computer best suited for specialized problems characterized by a high degree of regularity, such as image processing; it features synchronous (lockstep) and deterministic execution. This type of machine typically has an instruction dispatcher, a very high-bandwidth internal network, and a very large array of very small-capacity instruction units.

  Two varieties: Processor Arrays and Vector Pipelines. Examples for processor arrays: Connection Machine CM-2, Maspar MP-1, MP-2. Examples for vector pipelines: IBM 9000, Cray C90, Fujitsu VP, NEC SX-2, Hitachi S820.

  Single instruction: All processing units execute the same instruction at any given clock cycle.

  Multiple data: Each processing unit can operate on a different data element.

- **Multiple Instruction, Single Data (MISD)**: A single data stream is fed into multiple processing units. Each processing unit operates on data independently via independent instruction streams.

  Multiple instruction: Every processor may be executing a different instruction stream.

  Single data: Only one data stream is being used as input during any one clock cycle.

  Few actual examples of this class of parallel computer have ever existed. One is the experimental Carnegie-Mellon computer. Some conceivable uses might be:

  - multiple frequency filters operating on a single signal stream
  - multiple cryptography algorithms attempting to crack a single coded message.

- **Multiple Instruction, Multiple Data (MIMD)**: Currently, the most common type of parallel computer. Most modern computers fall into this category. Execution can be synchronous or asynchronous, and deterministic or non-deterministic. Examples: most current supercomputers, networked parallel computer "grids" and multi-processor SMP computers - including some types of PCs.

  Multiple instruction: Every processor may be executing a different instruction stream.

  Multiple data: Each processing unit can operate on a different data element.

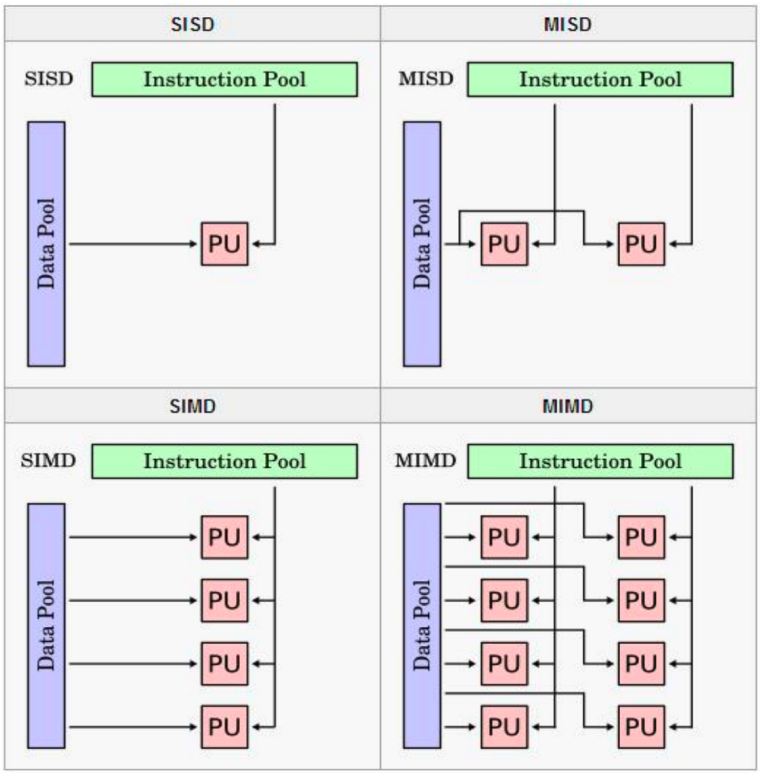


Figure 7.2: Flynn's classifications.

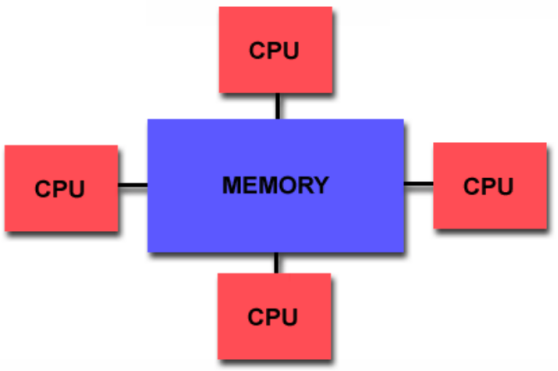


Figure 7.3: Shared memory architecture.

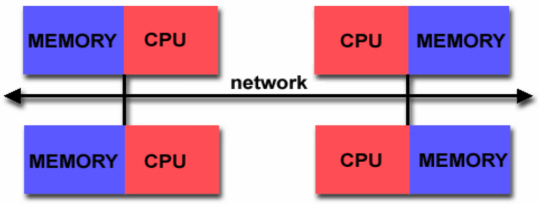


Figure 7.4: Distributed memory architecture.

7.1 Memory Architectures

Memory architectures for a parallel computer are divided into three major categories.

7.1.1 Shared Memory

Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space, as seen in Figure 7.3. Multiple processors can operate independently but share the same memory resources. Changes in a memory location actioned by one processor are visible to all other processors. Shared memory machines can be divided into two main classes based upon memory access times: UMA and NUMA.

7.1.2 Distributed Memory

Distributed memory systems require a communication network to connect inter-processor memory, shown in Figure 7.4. Processors have their own local memory; that is, there is no global address space across all processors. As each processor has its own local memory, they operate independently - changes a processor makes to its local memory have no effect on the memory of other processors. As a result, issues with cache coherency do not generally apply [1]

[1] You may still have cache coherency issues if you have intra-processor parallelism (i.e. multiple cores). This is closer to a hybrid memory architecture discussed in subsection 7.1.3.

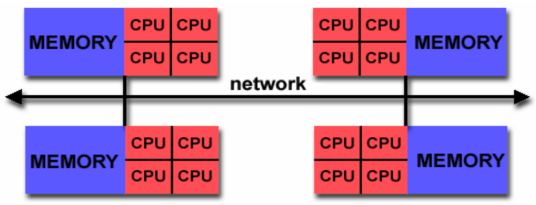


Figure 7.5: Hybrid memory.

When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated. Synchronization between tasks is likewise the programmer's responsibility.

The network "fabric" used for data transfer varies widely, though it can be as simple as Ethernet.

7.1.3 Hybrid Memory

The largest and fastest computers in the world today employ both shared and distributed memory architectures. The shared memory component is usually a cache coherent SMP machine. Processors on a given SMP can address that machine's memory as global.

The distributed memory component is the networking of multiple SMPs. SMPs know only about their own memory - not the memory on another SMP. Therefore, network communications are required to move data from one SMP to another.

Current trends seem to indicate that this type of memory architecture will continue to prevail and increase at the high end of computing for the foreseeable future.

The advantages and disadvantages of the hybrid model correspond to the common advantages and disadvantages of both shared and distributed memory architectures.

7.2 Parallel Programming

There are several parallel programming models in common use:

- shared memory
- threads
- message passing
- data parallelism
- hybrid

These models exist as an abstraction above hardware and memory architectures. These models are *not* specific to a particular type of machine or memory architecture; they can theoretically be implemented on any underlying hardware. This conclusion is not immediately obvious.

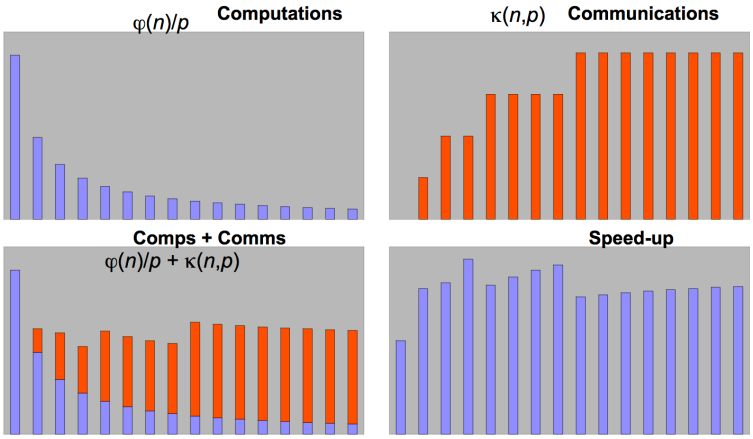


Figure 7.6: The effect of varying values in the speed-up formula.

7.2.1 Performance

The general speed-up possible for parallel computing can be quantified using the following formula:

$$\text{Speedup} = \frac{\text{Sequential execution time}}{\text{Parallel execution time}}$$

This can be used to determine a upper bound for parallel speedup where inherently sequential computations are $\sigma(n)$, potentially parallel computations are $\phi(n)$, and communication operations are $\kappa(n,p)$, n is the problem size, and p is the number of processors. Each component is investigated in Figure 7.6.

$$\psi(n,p) \leq \frac{\sigma(n) + \phi(n)}{\sigma(n) + \frac{\phi(n)}{p} + \kappa(n,p)}$$

Put simply:

$$\text{ParallelSpeedup}(n,p) \leq \frac{\text{Sequential}(n) + \text{PotentiallyParallel}(n)}{\text{Sequential}(n) + \frac{\text{PotentiallyParallel}(n)}{p} + \text{Communication}(n,p)}$$

7.2.1.1 Amdahl's Law

Amdahl's law states that a small portion of the program which cannot be parallelized will limit the overall speed-up available from parallelization.

Any large mathematical or engineering problem will typically consist of several parallelizable parts and several non-parallelizable (sequential) parts. This relationship is given by the equation

$$S = \frac{1}{1 - P}$$

where S is the speedup of the program (as a factor of its original sequential runtime), and P is the fraction that is parallelisable.

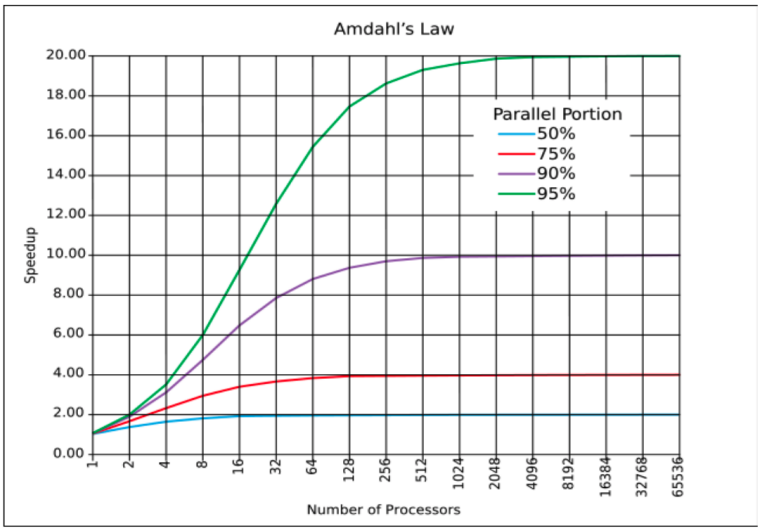


Figure 7.7: Consequences of Amdahl's law.

This leads to the consequence that if the sequential portion of a program is 10% of the runtime, we can get no more than a 10x speed-up, regardless of how many processors are added. This puts an upper limit on the usefulness of adding more parallel execution units. This is shown in Figure 7.7.

Amdahl's law can be derived by altering the general speedup formula:

$$\psi(n,p) \leq \frac{\sigma(n)+\phi(n)}{\sigma(n)+\frac{\phi(n)}{p}+\kappa(n,p)}$$

Remove the communication overhead (this will result in a larger upper bound):

$$\psi(n,p) \leq \frac{\sigma(n)+\phi(n)}{\sigma(n)+\frac{\phi(n)}{p}}$$

Let $f = \frac{\sigma(n)}{\sigma(n)+\phi(n)}$; that is, f is the fraction of the code which is inherently sequential. This leads to the complete formulation, which is a simplification of the general speedup formula:

$$\psi \leq \frac{1}{f+\frac{1-f}{p}}$$

Examples of using Amdahl's Law:

- 95% of a programs execution time occurs inside a loop that can be executed in parallel. What is the maximum speedup we should expect from a parallel version of the program executing on 8 CPUs?

$$\psi \leq \frac{1}{f+\frac{1-f}{p}} = \psi \leq \frac{1}{0.05+\frac{0.95}{8}} = 5.9$$

- 20% of a programs execution time is spent within inherently sequential code. What is the limit to the speedup achievable by a parallel version of the program?

$$\lim_{p \to \infty} \frac{1}{0.2 + \frac{1-0.2}{p}} = \frac{1}{0.2} = 5$$

Amdahl's law ignores the communication overhead, so it overestimates the possible speedup. It also assumes that f is constant, which may lead to the achievable speedup being underestimated.

The Amdahl effect refers to the potential speedup increasing with a larger n. This is due to the communication overhead $\kappa(n, p)$ typically having lower complexity than $\frac{\phi(n)}{p}$; as n increases, $\frac{\phi(n)}{p} \gg \kappa(n, p)$, resulting in the speedup increasing and f decreasing.

7.2.1.2 Gustafson's Law

Gustafson's Law (also known as Gustafson-Barsis' law, 1988) states that any sufficiently large problem can be efficiently parallelized. Gustafson's Law is closely related to Amdahl's law, which gives a limit to the degree to which a program can be sped up due to parallelization:

$$S(P) = P - \alpha * (P - 1)$$

where P is the number of processors, S is the speedup, and α the non-parallelizable part of the process.

Gustafson's law addresses the shortcomings of Amdahl's law, which cannot scale to match availability of computing power as the machine size increases.

It also removes the fixed problem size or fixed computation load on the parallel processors: instead, he proposes a fixed time concept which leads to scaled speed up. Amdahl's law is based on fixed workload or fixed problem size. It implies that the sequential part of a program does not change with respect to machine size (i.e, the number of processors). However, the parallel part is evenly distributed by n processors.

7.2.2 Examples

7.2.2.1 Independent 2D grid

This example demonstrates calculations on 2-dimensional array elements, with the computation on each array element being independent from other array elements (see Figure 7.8). As the calculations are independent of one another, it is an *embarrassingly parallel situation*. For best use of resources, the problem should be computationally intensive.

The serial program calculates one element at a time in sequential order. Pseudocode for this is shown in Algorithm 3.

For the parallel solution, array elements are distributed so that each processor owns a portion of the array (subarray); see Figure 7.9. Independent calculation of array elements ensures there is no need for communication between tasks. Distribution scheme is chosen by other criteria, e.g. unit stride (stride of 1) through the subarrays. Unit stride maximizes cache/memory usage.

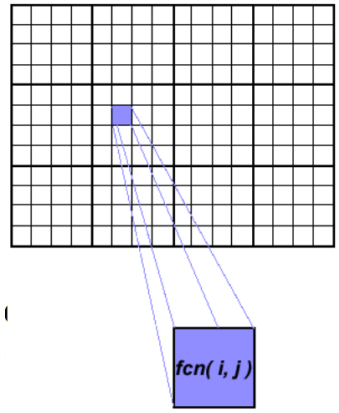


Figure 7.8: Embarrassingly-parallel 2D grid parallel programming problem.

Algorithm 3 Serial algorithm for computing elements in a 2D grid.

for j = 1, n **do**
  for i = 1, m **do**
    $\texttt{a}(i, j) \leftarrow \texttt{fcn}(i, j)$
  end for
end for

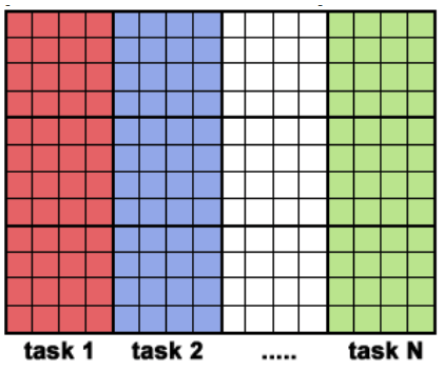


Figure 7.9: Partitioning of the 2D grid problem for each task.

Algorithm 4 Parallel algorithm for computing elements in a 2D grid.

for j = mystart, myend **do**
  for i = 1, m **do**
    $\texttt{a}(i, j) \leftarrow \texttt{fcn}(i, j)$
  end for
end for

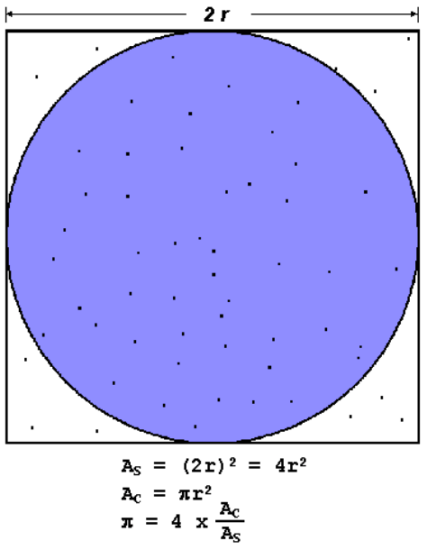


Figure 7.10: Calculating π through Monte Carlo sampling of a circle inside a square.

After the array is distributed, each task executes the portion of the loop corresponding to the data it owns. This is shown in Algorithm 4. Note that only the outer loop variables are different from the serial solution.

Pseudocode for each node is shown in Algorithm 5.

Algorithm 5 Complete parallel algorithm for computing elements in a 2D grid.

```
if am i the master? then
    initialize array
    send each worker info on part of array it owns
    send each worker its portion of initial array
    receive results from each worker
else
    receive information on which part of the array this worker owns
    receive the part of the array this worker owns
    for j = mystart, myend do
        for i = 1, m do
            a(i, j) ← fcn(i, j)
        end for
    end for
    send results to master
end if
```

7.2.2.2 Pi calculation

The value of π can be calculated through a variety of means. The algorithm chosen here is simple (depicted in Figure 7.10):

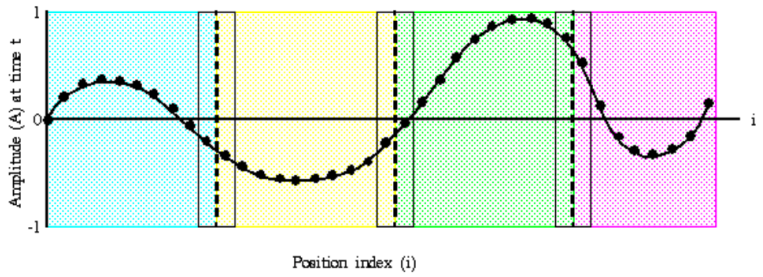


Figure 7.11: Parallel 1-D wave equation.

- Inscribe a circle in a square.
- Randomly generate points in the square.
- Determine the number of points in the square that area that are also in the circle.
- Let r be the number of points in the circle divided by the number of points in the square.
- Finally, $\pi \approx 4r$. The approximation is improved by increasing the number of points generated.

The serial pseudocode is shown in Algorithm 6. From this pseudocode, it can be seen that the

Algorithm 6 Serial algorithm for computing π.

$points_{total} \leftarrow 10000$
$points_{circle} \leftarrow 0$
for j $= 1$, $points_{total}$ **do**
  $(x, y) =$ RANDOM$(0, 1)$
  if INSIDE_CIRCLE(x, y) **then**
    $points_{circle} \leftarrow points_{circle} + 1$
  end if
end for
$\pi \approx 4 \times \frac{points_{circle}}{points_{total}}$

majority of the time spent running this program is spent in the loop, and that the algorithm itself is embarrassingly parallel. It is computationally intensive and requires little communication or I/O.

The parallel solution for this problem modifies the serial pseudocode by breaking the loop into portions that can be executed by each task. Each task can execute its portion of the loop a number of times. Each task can do its work without requiring any information from the other tasks (there are no data dependencies). This is implemented using the Single Process, Multiple Data (SPMD) model, which is a subset of MIMD; one task acts as the master and collects the results from the other processes. The pseudocode is shown in Algorithm 7.

Algorithm 7 Parallel algorithm for computing π. Each worker task calculates a number of samples that were inside the circle; these are then sent to the master, which adds them together and computes a final estimate of π.

$points_{total} \leftarrow 10000$
$points_{circle} \leftarrow 0$
$p \leftarrow$ number of tasks
$num \leftarrow \frac{points_{total}}{p}$
for j = 1, num **do**
  $(x, y) =$ RANDOM$(0, 1)$
  if INSIDE_CIRCLE(x, y) **then**
    $points_{circle} \leftarrow points_{circle} + 1$
  end if
end for
if am i the master? **then**
  $points_{circle} \leftarrow points_{circle} + \sum_p$ RECEIVE(p)
  $\pi \approx 4 \times \frac{points_{circle}}{points_{total}}$
else
  SEND(master, $points_{circle}$)
end if

7.2.2.3 1-D Wave Equation

Implemented with the SPMD model. The entire amplitude array is partitioned and distributed as sub-arrays to all tasks, as seen in Figure 7.11. Each task owns a portion of the total array. All points require equal work, so the points should be divided equally. A block decomposition would have the work partitioned into the number of tasks as chunks, allowing each task to own mostly contiguous data points.

Communication only needs to occur on data borders. The larger the block size, the less the communication. The equation to be solved is the one-dimensional wave equation:

$$A(i, t+1) = (2.0 \times A(i, t)) - A(i, t-1) + (c \times (A(i-1, t) - (2.0 \times A(i, t)) + A(i+1, t)))$$

where c is a constant.

Note that the amplitude will depend on previous timesteps $(t, t-1)$ and neighboring points $(i-1, i+1)$. This data dependence will mean that a parallel solution will require communication. This parallel solution is depicted in Algorithm 8.

[2] I didn't want to transcribe this to pseudocode properly. Sorry!

Algorithm 8 Parallel algorithm for computing the 1-D wave equation. [2]

```
find out number of tasks and task identities

#Identify left and right neighbors
left_neighbor = mytaskid - 1; right_neighbor = mytaskid +1
if mytaskid = first then left_neigbor = last
if mytaskid = last then right_neighbor = first
find out if I am MASTER or WORKER
if I am MASTER
     initialize array ; send each WORKER starting info and subarray
else if I am WORKER
     receive starting info and subarray from MASTER
endif
#Update values for each point along string
#In this example the master participates in calculations
do t = 1, nsteps
     send left endpoint to left neighbor ; receive left endpoint from right neighbor
     send right endpoint to right neighbor ; receive right endpoint from left neighbor
#Update points along line
    do i = 1, npoints
        newval(i) = (2.0 * values(i)) - oldval(i) + (sqtau * (values(i-1) - (2.0 *
        values(i)) + values(i+1)))
    end do
end do
#Collect results and write to file
if I am MASTER
     receive results from each WORKER write results to file
else if I am WORKER
     send results to MASTER
endif
```

Chapter 8

Parallel Computing Alternatives

8.1 Parallel Virtual Machine

PVM is based on a dynamic computing model, where nodes can be added/deleted on the fly and parallel tasks can be spawned/killed during the computation. This provides fault tolerance and adaptability, but it is not as rich in message passing features as MPI. However, as it is a virtual machine, it provides features ideal for creating dynamic parallel programs.

Each host participating in the virtual machine executes the `pvmd` daemon. The `pvmd`s of all the hosts combine to form the virtual machine. Applications with PVM primitives can contact their local `pvmd` to interact and/or execute actions across the virtual machine. This is shown in Figure 8.1.

PVM provides a monitoring and notification system where any or all tasks in an application can be asked to be notified of specific events. These include exiting of a task within the application, or failure/deletion of a node in the cluster. Tasks can watch their neighbouring tasks in a logical ring. The monitoring is actually done by PVM, but the logical ring reduces the number of messages.

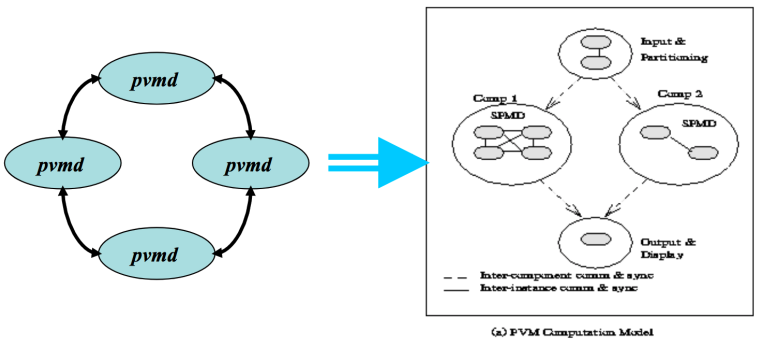


Figure 8.1: PVM computational model.

Cluster programs can be made to adapt to the cluster they are running on. A parallel application may dynamically adapt to the size of the virtual machine. They can add and release nodes based on the computational needs of the application.

8.2 LINDA

LINDA is a concurrent programming model that was evolved from a Yale University research project. The primary concept is Tuple-Space, which is an abstraction by which cooperating processes communicate. It is an alternative paradigm to the two traditional methods of parallel processing: shared memory and message-passing.

Tuple-Space is an abstraction of distributed shared memory with some differences; Tuple-Spaces are associative, and destructive and nondestructive reads and different coherency semantics are possible.

Applications use the LINDA model by embedding explicitly, within cooperating sequential programs, constructs that manipulate (insert/retrieve tuples) the tuple space.

From the application's point of view, LINDA is a set of programming language extensions for facilitating parallel programming. It provides a shared-memory abstraction for process communication without requiring the underlying hardware to physically share memory.

The LINDA system usually refers to a specific software implementation that supports the LINDA programming model. System software is provided that establishes and maintains tuple spaces; this is used in conjunction with libraries that appropriately interpret and execute LINDA primitives.

Depending on the environment (shared-memory multiprocessors, message-passing parallel computers, networks of workstations, etc.), the tuple space mechanism is implemented using different techniques and with varying degrees of efficiency. For instance, Piranha proposes a proactive approach to concurrent computing: computational resources (viewed as active agents) seize computational tasks from a well-known location based on availability and suitability.

8.3 OpenMP

OpenMP is a set of open specifications for Multi-Processing defined by collaborative work between interested parties from the hardware and software industry, government and academia. It is jointly defined and endorsed by a group of major computer hardware and software vendors.

It is an Application Program Interface (API) to explicitly direct multi-threaded, shared memory parallelism. It is comprised of three primary API components: compiler directives, runtime library routines, and environment variables. The API is relatively portable; it is specified for C/C++ and FORTRAN, and there is support for most major platforms, including UNIX, Linux and Windows.

In the early 90s, vendors of shared-memory machines supplied similar, directive-based, FORTRAN programming extensions: The user would augment a serial FORTRAN program with directives specifying which loops were to be parallelized. The compiler would be responsible

for automatically parallelizing such loops across the SMP processors. Implementations were all functionally similar, but were prone to divergence (as is typically the case with standards).

The first attempt at a standard was the draft for ANSI X3H5 in 1994. It was never adopted, largely due to waning interest as distributed memory machines became popular. The OpenMP standard specification started in the spring of 1997, taking over where ANSI X3H5 had left off as newer shared memory machine architectures started to become prevalent again.

Goals of the OpenMP project include:

- **Standardization**: Providing a standard among a variety of shared memory architectures/-platforms.
- **Lean and Mean**: Establishing a simple and limited set of directives for programming shared memory machines. Significant parallelism can be implemented by using just 3 or 4 directives.
- **Ease of Use**:
  - Providing the capability for incrementally parallelizing a serial program. This is unlike message-passing libraries, which typically require an all-or-nothing approach.
  - Providing the capability to implement both coarse-grained and fine-grained parallelism with ease.
- **Portability**:
  - Supporting Fortran (77, 90, and 95), C, and C++
  - Maintaining a public forum for API and membership, so that interested parties can join

However, OpenMP is not:

- Meant for distributed memory parallel systems (by itself)
- Necessarily implemented identically by all vendors
- Guaranteed to make the most efficient use of shared memory
- Required to check for data dependencies, data conflicts, race conditions, or deadlocks
- Required to check for code sequences that cause a program to be classified as non-conforming
- Meant to cover compiler-generated automatic parallelization and directives to the compiler to assist such parallelization
- Designed to guarantee that input or output to the same file is synchronous when executed in parallel. The programmer is responsible for synchronizing input and output.

8.3.1 Programming Model

OpenMP is based around shared memory, thread-based parallelism. It is also an explicit (not automatic) programming model, offering the programmer full control over parallelization. It uses the fork-join model of parallel execution, as depicted in Figure 8.2.

All OpenMP programs begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered.

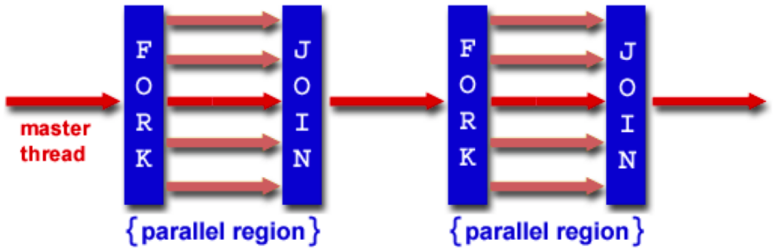


Figure 8.2: Fork-join parallel execution, used by OpenMP.

The master thread then creates a team of parallel threads; this is the *fork* step.

The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads.

When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread; this is the *join* step.

Most OpenMP parallelism is specified through the use of compiler directives which are embedded in C/C++ or Fortran source code. Additionally, there are optional features that implementations are not required to support:

- The API provides for the placement of parallel constructs inside of other parallel constructs - that is, nested parallelism.
- The API provides for dynamically altering the number of threads which may be used to execute different parallel regions.

8.3.1.1 I/O

OpenMP specifies nothing about parallel I/O; this is particularly important if multiple threads attempt to write/read from the same file. If every thread conducts I/O to a different file, the issues are not as significant. It is entirely up to the programmer to ensure that I/O is conducted correctly within the context of a multi-threaded program.

OpenMP provides a *relaxed-consistency* and *temporary* view of thread memory (in their words). In other words, threads can cache their data and are not required to maintain exact consistency with real memory all of the time. When it is critical that all threads view a shared variable identically, the programmer is responsible for ensuring that the variable has been *flushed* by all threads as appropriate (that is, the state of the variable is synchronised with real memory for all threads).

8.3.2 Examples

Algorithm 9 demonstrates the use of OpenMP in C. Note how parallelism is delivered by compiler directives (the `#pragmas`). To compile the example, use `gcc -fopenmp omp_hello.c -o hello`. To execute the example, use `./hello` (Linux, Unix, Mac OS X) or `./hello.exe` (Windows). By default, OpenMP will set the number of threads to equal the number of available CPUs. The number of threads to use can be set by changing the `OMP_NUM_THREADS` environment variable.

Algorithm 9 OpenMP example by Blaise Barney.

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    int nthreads, tid;

/* Fork a team of threads giving them their own copies of ↩
   variables */
#pragma omp parallel private(nthreads, tid) {
    /* Obtain thread number */
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);
    /* Only master thread does this */
    if (tid == 0) {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);
    }
}
/* All threads join master thread and disband */

}
```

Your output will look similar to the following; the actual order of output strings may vary:

```
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 3
Hello World from thread = 1
Hello World from thread = 2
```

Algorithm 10 shows the use of OpenMP for worksharing. Note the use of additional `#pragma`s, as well as the use of chunking.

Algorithm 10 OpenMP workshare example by Blaise Barney.

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define CHUNKSIZE 10
#define N 100

int main(int argc, char* argv[])
{
    int nthreads, tid, i, chunk;
    float a[N], b[N], c[N];
    /* Some initializations */
    for (i = 0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;

#pragma omp parallel shared(a, b, c, nthreads, chunk) private(i,↩
    tid) {
      tid = omp_get_thread_num();
      if (tid == 0) {
          nthreads = omp_get_num_threads();
          printf("Number of threads = %d\n", nthreads);
      }
      printf("Thread %d starting...\n", tid);

      #pragma omp for schedule(dynamic, chunk)
      for (i = 0; i < N; i++) {
          c[i] = a[i] + b[i];
          printf("Thread %d: c[%d]= %f\n", tid, i, c[i]);
      }
  }
  /* end of parallel section */
}
```

On a dual-core CPU, this produces output similar to the following:

```
Thread 1 starting...
Number of threads = 2
Thread 1: c[0]= 0.000000
Thread 0 starting...
Thread 1: c[1]= 2.000000
Thread 0: c[10]= 20.000000
Thread 1: c[2]= 4.000000
Thread 0: c[11]= 22.000000
Thread 1: c[3]= 6.000000
Thread 0: c[12]= 24.000000
```

```
Thread 1: c[4]= 8.000000
Thread 1: c[5]= 10.000000
...
```

8.4 GPGPU

Traditional CPU design is suited to sequential processing. Graphics require data parallelism, which forms the design basis of the *graphics processing unit*, or GPU. The GPU can be extended for use with non-graphic data - *general-purpose GPU*, or *GPGPU* - which allows for very high performance virtually free of cost.

However, only certain types of applications can benefit from this approach. Additionally, GPGPU programming requires the use of special libraries; however, pre-existing software and wrappers are available in some cases.

GPGPU was made possible with the addition of programmable stages and higher precision arithmetic to the rendering pipelines; these pipelines were originally used to facilitate rasterization-based rendering. With this functionality, stream processing on non-graphics data was made possible.

Stream processing is a computing model based on the SIMD paradigm. Some applications can be easily parallelised within the limits of this style of processing; they can utilise the multiple floating point units on the GPU. Allocation, synchronisation and communication, which are usually required in SIMD, are assumed to be automatically managed by the units in this model.

The model simplifies parallel software and hardware by restricting the parallel computation that can be performed. Computation is performed with two components:

- A **data set** constitutes a (data) stream.
- A **kernel function** is a series of operations that is applied to each element in the stream. It is commonly defined as a series of nested loops with no data specification.

Several features of the programming model facilitate high-performance programming. *Uniform streaming*, where the same kernel function is applied to all elements in the stream, is common. Kernel functions are typically *pipelined*, and local on-chip memory is reused to minimise external memory bandwidth. As the kernel and stream abstractions expose data dependencies by design, compilers can fully automate and optimise on-chip management tasks to maximise performance.

A kernel requires that stream data abide by two characteristics: being independent and local [1]. The kernel defines the basic unit of data for both input and output, allowing the hardware to better allocate resources and schedule I/O; this definition is usually explicit in the kernel, which is expected to have well-defined structures for I/O.

Compute blocks that are well-defined and independent allow scheduling of bulk I/O operations; this scheduling is undertaken in such a way to leverage the underlying hardware cache and memory bus in the most efficient manner possible. Additionally, values associated with a single kernel invocation (that is, values local to that kernel) are treated as local variables and thus use fast kernel-local registers.

[1] Not explained in the slides

Generally, the following type of applications will benefit from stream computing:

- Compute intensive applications where the number of arithmetic operations per each I/O or global memory access operation is large, such that performance is bound by computation and not I/O.
- Applications with data parallelism where the same kernel function can be applied to records of an input stream, and a number of records can be processed simultaneously without waiting for results from previous records.
- Applications with data locality where data is produced once, read once or twice later in the application, and never read again. Intermediate streams passed between kernels, as well as intermediate data within kernel functions, can capture this locality directly using the stream processing programming model.

Notable stream processing languages include the open standards of ACOTES (based on OpenMP) and OpenCL, and the vendor-specific solutions of BROOK+ from AMD and Compute Unified Device Architecture (CUDA) from NVIDIA.

8.4.1 OpenCL

OpenCL is a set of open specifications for a stream processing framework; it is designed for number crunching and parallel computing tasks. Each vendor offers their own implementation, but these implementations must be compliant with the specifications. It was originally proposed with Apple, with NVIDIA and Intel acting as supporting parties.

The specifications were developed by a number of companies, and then handed over to Khronos Group to be maintained. Khronos offer the contents for free without a license free, and are also responsible for maintaining the OpenGL specifications.

OpenCL codifies a shift from clock-speed (based around sequential processing) to multicore processing (parallel programming, data sharing). Its API provides a good framework for utilising the underlying (parallel) hardware in a portable fashion. It aims to abstract over hardware heterogeneity, and offers CPU and GPU support.

Applications for OpenCL include:

- Image, video and audio processing
- Gaming, scientific calculations and simulations. It has interoperability with OpenGL for highly visual data representations.
- Financial modelling
- Computationally intensive data-parallel applications

8.4.2 Programming Techniques

There are several programming techniques used for GPGPU processing. These are discussed below.

8.4.2.1 Map

The map operation simply applies the kernel (the user specified function) to every element in the stream. A simple example is multiplying each value in the stream by a constant (e.g. increasing the brightness of an image). It is simple to implement on the GPU.

8.4.2.2 Reduce

Some computations require calculating a smaller stream (possibly a stream of only 1 element) from a larger stream. This is called a reduction of the stream. Generally, a reduction can be accomplished in multiple steps:

- The results from the previous step are used as the input for the current step.
- The range over which the operation is applied is reduced until only one stream element remains.

8.4.2.3 Scatter

The scatter operation is equivalent to `a[i] = x` in C. The scatter operation is best defined on the vertex processor, as the vertex processor is able to adjust the position of the vertex, which allows the programmer to control where information is deposited on the grid. By comparison, the fragment processor cannot perform a direct scatter operation because the location of each fragment on the grid is fixed at the time of the fragment's creation and cannot be altered by the programmer.

A scatter implementation would first emit both an output value and an output address followed by a gather operation.

8.4.2.4 Gather

The gather operation is the opposite of the scatter operation; it is equivalent to `x = a[i]` in C. This is analogous to image processing in the GPU, where the fragment processor is able to read textures in a random-access fashion. A GPGPU can gather information from any grid cell, or multiple grid cells, as need be.

8.4.2.5 Filter / Sort

Stream filtering is essentially a non-uniform reduction. Filtering involves removing items from the stream based on some criteria.

The sort operation transforms an unordered set of elements into an ordered set of elements. The most common implementation on GPUs uses sorting networks.

8.4.2.6 Search

The search operation allows the programmer to find a particular element within the stream, or possibly find neighbors of a specified element. The GPU is used to run multiple searches in parallel.

Chapter 9

Instruction-Level Parallelism

Increasing processor performance has been provided by increased Instruction Level Parallelism (ILP). Processors have progressed from non-pipelined processors to pipelined processors to multiple pipeline processors to VLIW processors to superscalar processors; each step offers additional parallelism in the execution process.

Consider the following code fragment:

```
mul r3, r4 -> r8
add r1, r2 -> r7
load r6, (r1)
add #5, r5
```

None of these instructions have dependencies on each other; that is, they are all independent. This means they can be executed in any order, and the CPU can rearrange them for the most performant execution order. Why does execution order matter? ILP allows you to execute multiple instructions at once, either through pipelining or through VLIW/superscalar approaches.

9.1 Pipelining

As seen in Figure 9.1, operations can be split into multiple stages: fetch, decode, execute, and writeback. These are the typical stages, but more are possible; the Intel NetBurst architecture was infamous for having 31 stages in its pipeline. When an instruction is executed, it is in one of the stages; the other stages are free for use.

This allows multiple instructions to be run simultaneously if each instruction is using a different stage of the pipeline. If instruction 1 is currently in the write-back stage, then instructions 2-4 after it can use the execute, decode and fetch stages. This is illustrated in Figure 9.2.

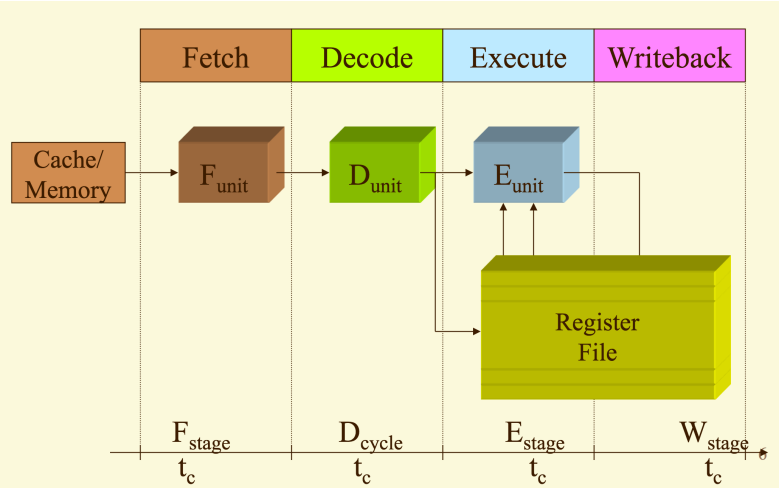


Figure 9.1: Pipelining: splitting one operation into multiple stages.

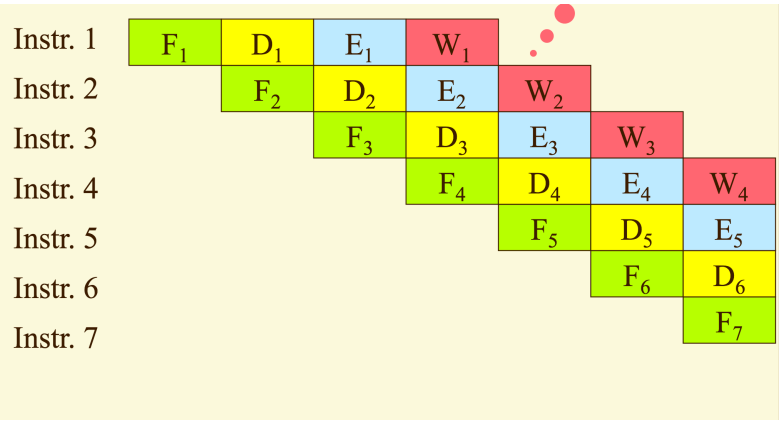


Figure 9.2: Multiple instructions through pipelining.

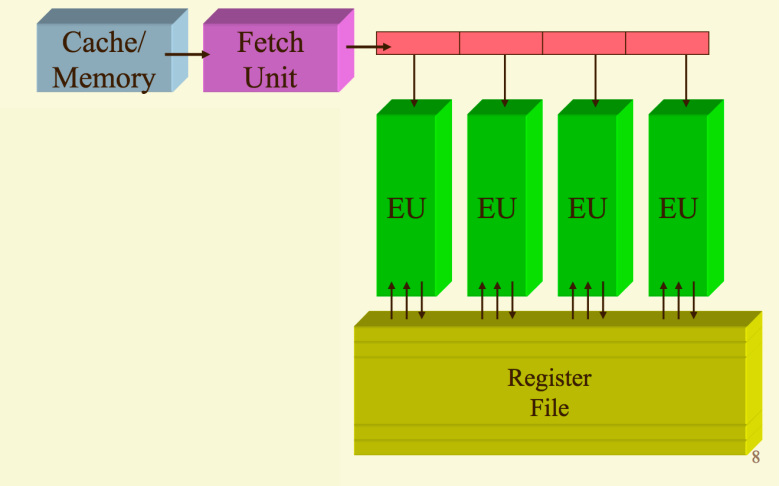


Figure 9.3: VLIW approach. Note how the four execution units are fed with the four segments of the one instruction word - you might even say the instruction word is *very long*. The compiler decides which instructions form the instruction word at compile-time.

9.2 VLIW/Superscalar

Very Long Instruction Word (VLIW) and Superscalar approaches are largely similar. Essentially, there are multiple *execution units* (EUs) dedicated to executing multiple instructions simultaneously. In VLIW, the instructions to be run across the units are scheduled in software (i.e. they are sequenced as part of the compiled machine code; see Figure 9.3), while the superscalar approach is scheduled at runtime by the hardware (see Figure 9.4).

9.3 Dependencies

However, in all instruction-level parallelism approaches, dependencies are a crucial factor. Dependencies are responsible for determining which instructions can be scheduled where, and whether they can be run in parallel or must be serialised.

There are multiple types of dependencies; these are discussed below.

9.3.1 Data Dependencies

Future instructions depend on the results of prior instructions; this means that future instructions have to wait on registers and memory to be resolved. These can be further broken down into two types of data dependencies:

- **Read-after-write**: Data must be read, but is currently being written to. That is, if $r1$ is written to in instruction 1 and used in instruction 2, instruction 2 cannot be rearranged prior to instruction 1: it depends on the value written by instruction 1.

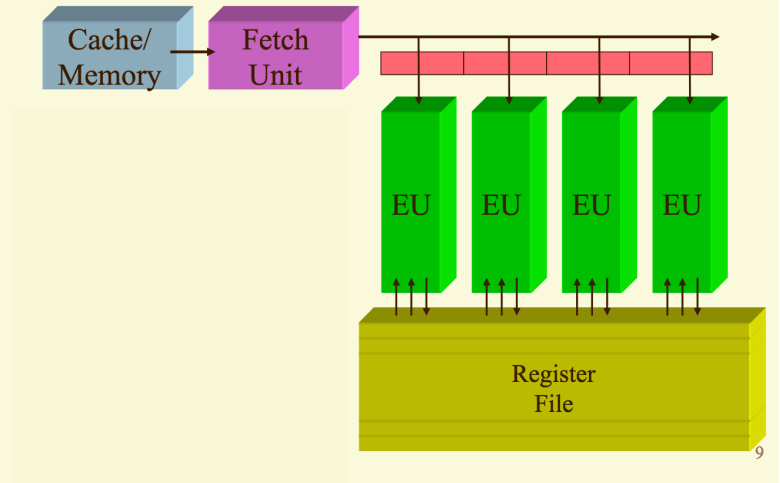


Figure 9.4: Superscalar approach. There are four individual instructions that are concurrently fed to the execution units; the hardware is responsible for determining *what* these four instructions are based on the instruction stream at runtime.

A load-use dependency is where a register is loaded from memory, and then used. A define-use dependency is where a register is assigned to by an instruction, and then used by another instruction.

- **Write-after-read/write**: This is a *false dependency*; it will result in the correct results as long as the instructions are executed in order. That is, a register that has been read/written to can be rewritten as long as the rewriting instruction occurs after the original read/write instruction. This can be broken by pipelining, but this can be fixed by using register renaming to temporarily change the registers being altered.

Not all data dependencies are obvious by examining the source code. As an example, a loop iteration that depends on the value calculated by the previous iteration has a data dependency; the compiler needs to be able to analyse loops to detect and handle these scenarios.

9.3.2 Control Dependencies

Future instructions cannot be evaluated until an upcoming branch in the instruction flow is evaluated - that is, code in an `if` branch can't normally be evaluated until the condition of the `if` has been evaluated.

This will normally stall a pipeline, unless special measures are taken (such as branch prediction). In general, for 25% of code one in four instructions is a branch; as a result, control dependencies cannot be ignored. This is especially problematic for superscalar/VLIW architectures, where multiple EUs cannot be used to handle instructions that are blocked on a control dependency.

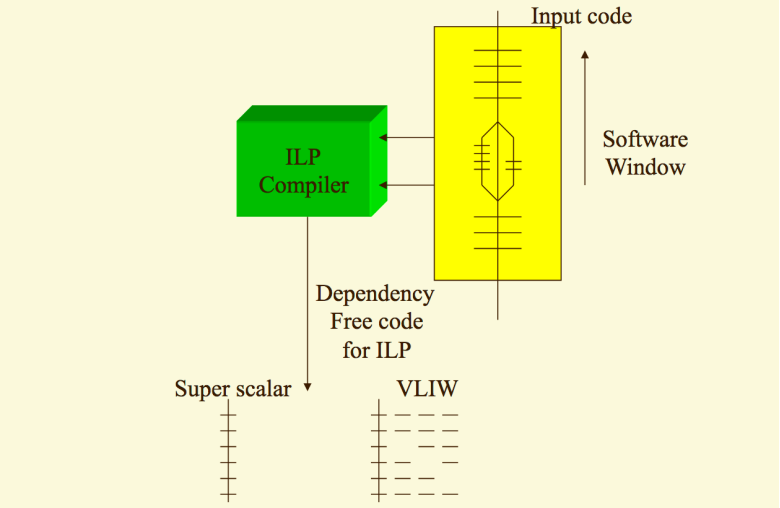


Figure 9.5: Static scheduling.

9.3.3 Resource Dependencies

The CPU has a limit to the number of Arithmetic Logic Units (ALUs), memory ports, and other resources at any given time; parallelism is fundamentally limited by the number of free resources.

9.4 Scheduling

There are three primary methods of instruction scheduling, briefly discussed prior.

9.4.1 Static

In static scheduling (depicted in Figure 9.5), the compiler is responsible for generating optimal code that maximises instruction-level parallelism. The processor should receive dependency-free and optimised code for parallel execution. This is typical for VLIW architectures and a few pipelined processors (e.g. MIPS).

9.4.2 Dynamic

In dynamic scheduling (depicted in Figure 9.6), the hardware is responsible for determining the sequence of instructions in the incoming instruction stream that maximises instruction-level parallelism. It is performed entirely by the processor, which expects no effort on the behalf of the compiler to ease the scheduling problem. This was the approach favoured by early ILP processors (e.g. CDC 6600, IBM 360/91, and more).

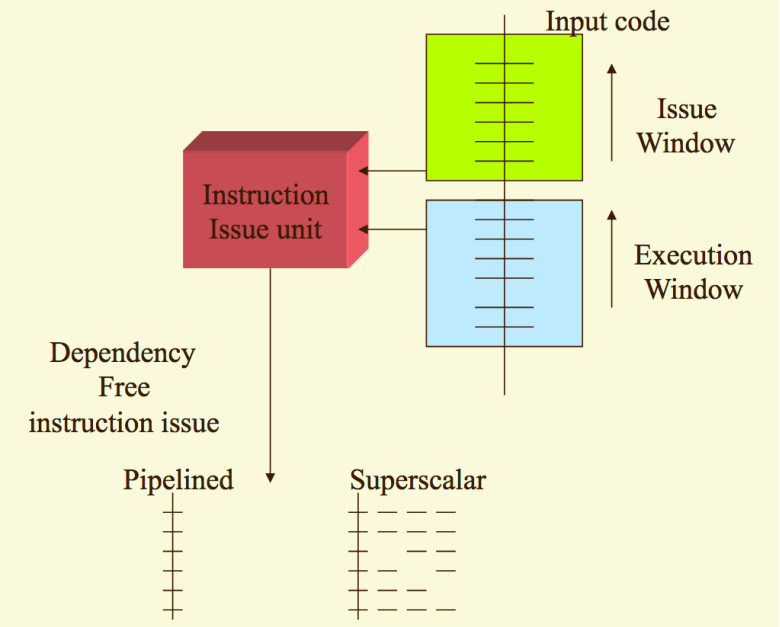


Figure 9.6: Dynamic scheduling.

9.4.3 Hybrid

In hybrid scheduling, both the compiler and the hardware work together to find optimal scheduling for instructions. The compiler makes a best-effort attempt to determine an appropriate scheduling; the hardware will then receive this, attempt to determine dependencies itself, and further optimise the instruction stream prior to execution. This is now commonplace, being the usual practice for modern pipelined superscalar processors (e.g. RS/6000, PL.8/XL, i960, QTC SF960, and notably modern x86 processors).

9.5 Sequential Consistency

When instructions are executed in parallel, the processor must be careful to preserve sequential consistency of operations. Consider the following code:

```
div r1, r2 -> r3
add r5, r6 -> r7
jz somewhere
```

The `div` and `add` set condition codes implicitly (in this case, whether or not the resulting value was zero), which are used by the `jz` instruction (which jumps when the if-zero condition code is set) to determine its next course of action. Division typically takes longer than addition; as a result, it is possible that the if-zero condition code of `div` may override that of `add`, despite `add` being evaluated "after" the `div`. Sequential consistency must be maintained to ensure that this is handled appropriately.

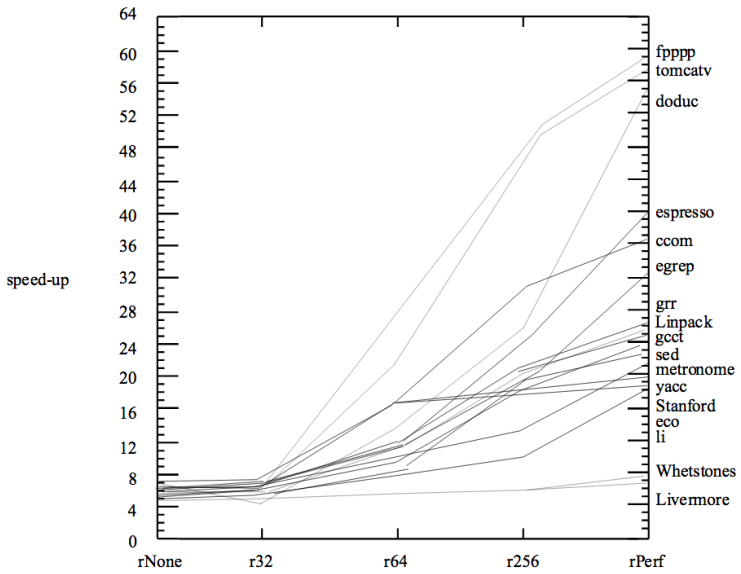


Figure 9.7: Effects of register renaming on performance.

There are two kinds of consistency: *strong consistency*, which preserves the actual execution order, and *weak consistency*, which may execute out of order as long as the result is still correct.

9.6 Performance

The performance of instruction-level parallelism is fundamentally limited by

- the underlying algorithm (which may have dependencies that cannot be removed)
- compiled code (the compiler may introduce false dependencies and code that is difficult to pipeline)
- actual hardware (the hardware may not have the resources to execute a given sequence of code in parallel)

All three of these factors must be considered when the maximum speedup is evaluated. Given this, previous studies have found potential speedups from the single-digits (i.e. 1.2 times faster) to hundreds of thousands of times faster (i.e. 64,500 times faster) for particular classes of applications on particular systems.

Additionally, register renaming may have a significant impact. The hardware can safely *rename* a register to remove a false dependency, allowing it to increase instruction-level parallelism. Algorithms for doing this have improved over the decades. This can be seen in Figure 9.7, which illustrates the performance gains from the use of register renaming.

Chapter 10

Vector Architectures

The basic principle underpinning a vector processor is the combination of two vectors to produce an output vector. If A, B and C are vectors of N elements, a vector processor can perform the operation

$$C = B + A$$

which is equivalent to

$$C(i) = B(i) + A(i) \text{ where } 0 \leq i \leq N - 1$$

The memory subsystem for a vector processor needs to be able to support 2 reads per cycle and 1 write per cycle. This is illustrated in Figure 10.1.

A vector processor is interesting as it can be used to dramatically speed up computation for operations over vectors. Consider unvectorised computation in Figure 10.2, and vectorised computation in Figure 10.3. The former is naive and must reload all state at the start of each computation; as a result, it must spend $N(t_1 + t_2 + t_3)$ time (where t_1, t_2 and t_3 are the pre-computation, computation, and post-computation times respectively) to evaluate N results. The latter can compute multiple results simultaneously, allowing it to reduce the time to process N elements to $t_1 + Nt_2 + t_3$ (i.e. setup, computation for N elements, followed by teardown).

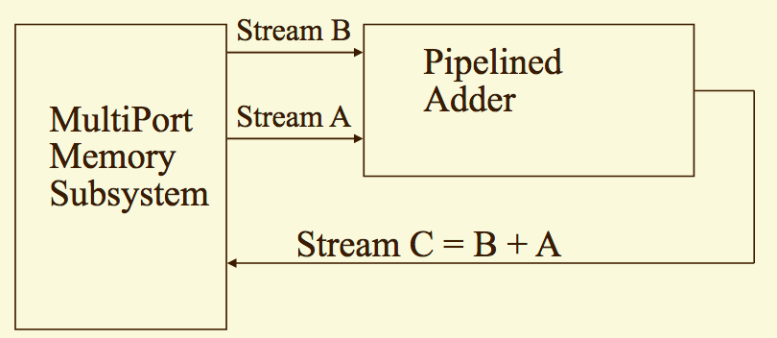


Figure 10.1: Memory system for a vector processor.

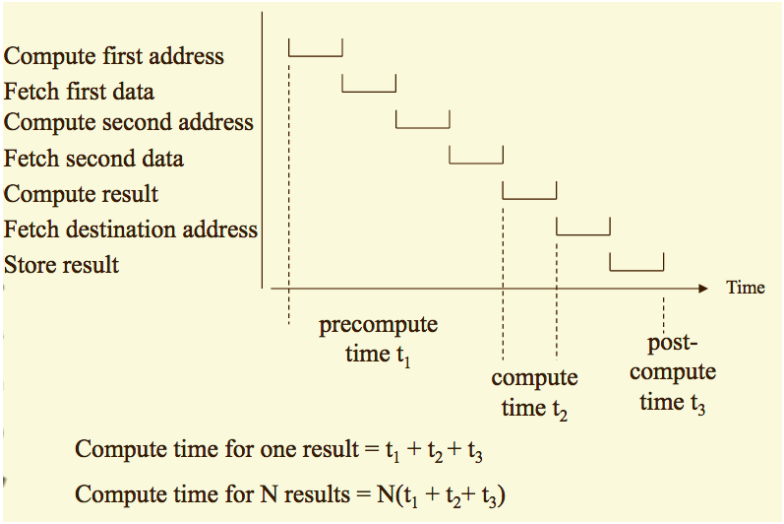


Figure 10.2: Unvectorised computation.

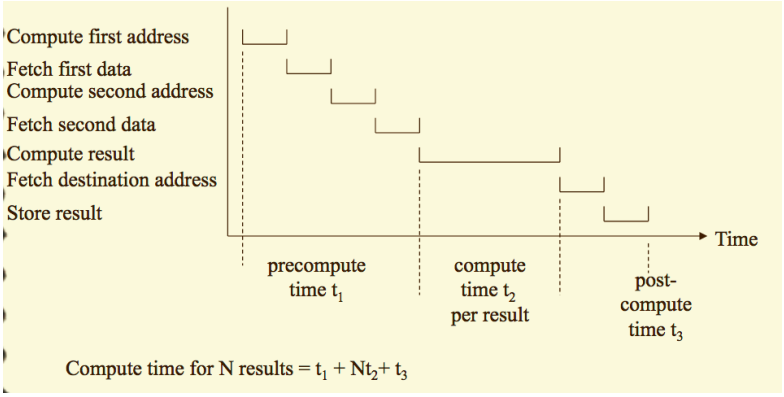


Figure 10.3: Vectorised computation.

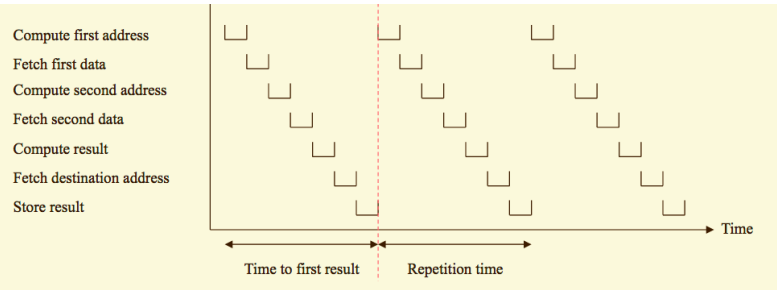


Figure 10.4: Non-pipelined computation.

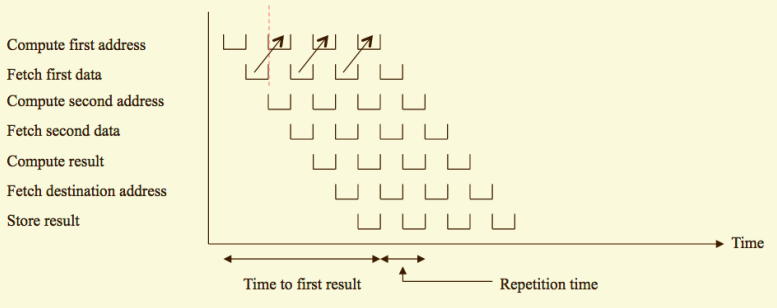


Figure 10.5: Pipelined computation.

This is made possible through the existence of pipelining, discussed in chapter 9. This is shown in Figure 10.4 and Figure 10.5, which showcase non-pipelined (i.e. serial) and pipelined (i.e. vectorised) computation respectively. Note that the execution time of a pipeline will always be bounded by its slowest component; this can be improved by increasing the granularity of the pipeline by increasing the number of stages. However, increasing granularity has its drawbacks, especially with regards to dependencies (discussed in section 9.3).

10.1 Interleaving

For vector pipelining to work, it must be possible to fetch instructions from memory quickly. This is typically achieved through the use of cache memories (which caches repeatedly-accessed and upcoming instructions in CPU-local memory for rapid access), and interleaving (which will be discussed).

Conventional memory, seen in Figure 10.6, consists of a set of storage locations accessed with some sort of address decoder. However, this means that there can be only one execution unit accessing the memory at any given time, preventing access from other EUs.

An interleaved memory system, seen in Figure 10.7, uses a number of banks which correspond to a certain range of addresses. This effectively partitions the memory into multiple sections which can be concurrently accessed. Banks are usually selected using some of the low-order bits of the address (i.e. the least-significant bits) to ensure that sequential access will access different banks.

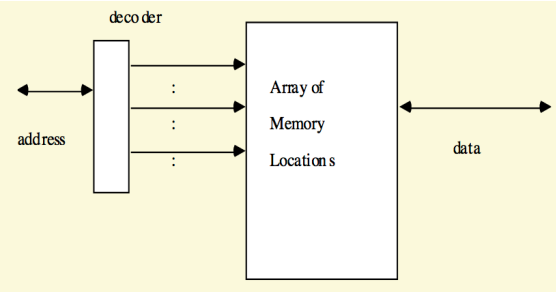


Figure 10.6: Conventional memory.

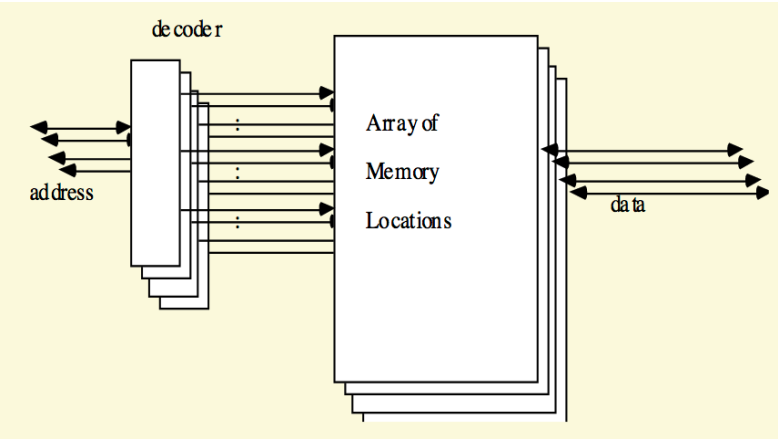


Figure 10.7: Interleaved memory.

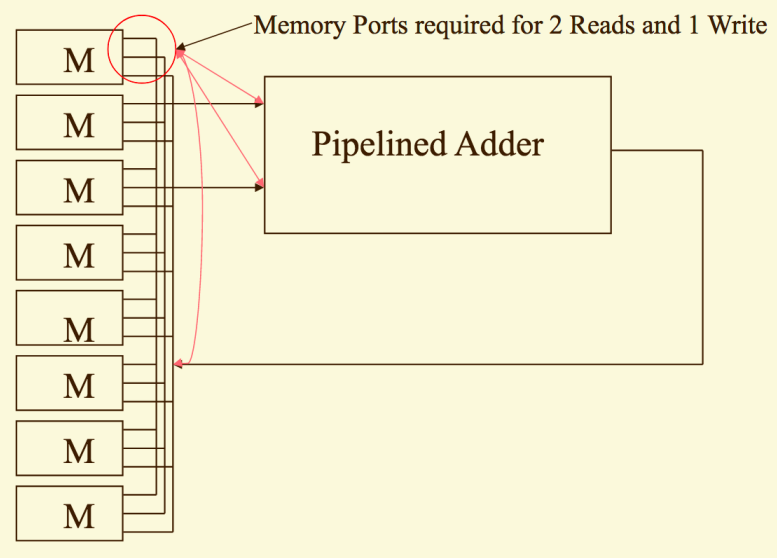


Figure 10.8: Pipelined adder fed by an interleaved memory system.

A pipelined machine can be kept fed with instructions even though the main memory may be quite slow. An interleaved memory system slows down when subsequent accesses are for the same bank of memory. However, this is rare when prefetching instructions, because they tend to be sequential; as mentioned previously, banks are chosen to minimise the impact of sequential access. Additionally, it is possible to access two locations at the same time if they reside in different banks.

The use of multiple banks can be seen in context in Figure 10.8, which uses two memory banks for its input and one memory bank for its output. This is further detailed in Figure 10.9 and Figure 10.10, which depict the memory layout used for the adder and the utilisation of CPU pipelines respectively.

However, it is possible for memory contention to occur, as depicted in Figure 10.11. To alleviate this, a delay path can be added to the pipelined adder (seen in Figure 10.12) to mitigate this. The impact of doing this is seen in Figure 10.13.

10.2 CRAY-1

The CRAY-1 has facilities for vector processing that work on a 64-word basis per vector. The compilers are vectorising, and can translate loops of serial operations into vector arithmetic. Additionally, the floating-point functional units can accept vector registers. This can be seen in Figure 10.14.

The CRAY-1 was capable of achieving even faster vector operations by using chaining. The result vector was not only sent to the destination vector register, but also directly to another functional unit. Data could be made to chain from one functional unit to another, potentially without any intermediate storage.

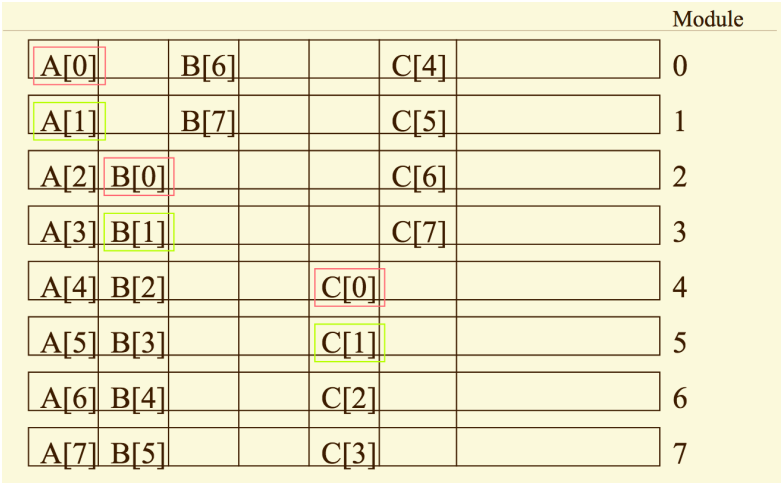


Figure 10.9: Memory layout of an interleaved vector processing system.

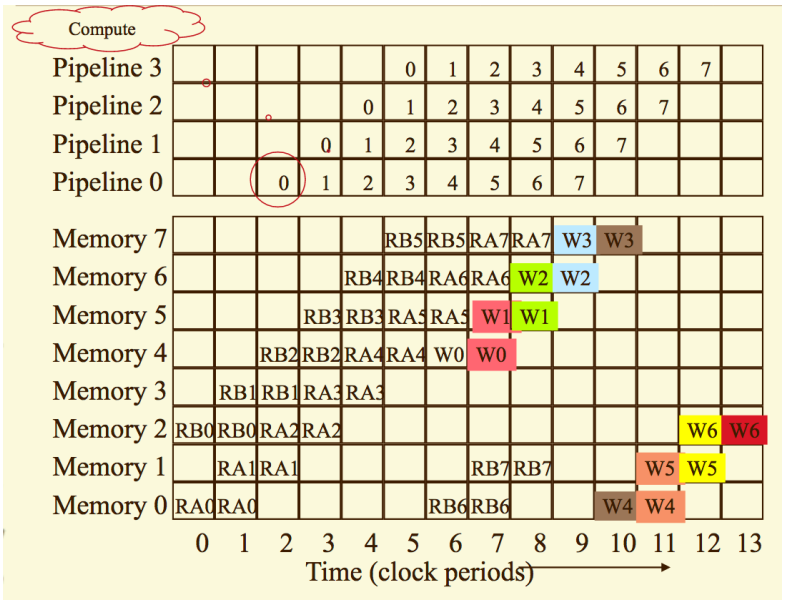


Figure 10.10: Pipeline utilisation of an interleaved vector processing system.

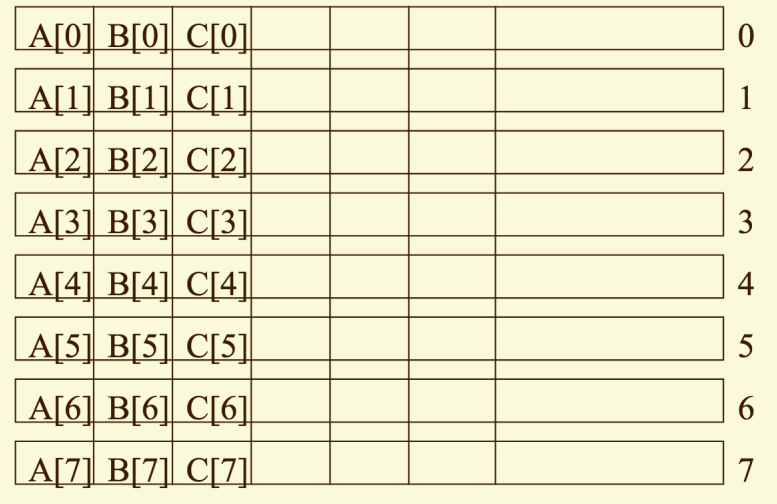


Figure 10.11: Memory contention; these arrays are arranged in such a way that all accesses are on the same bank, dramatically slowing down processing.

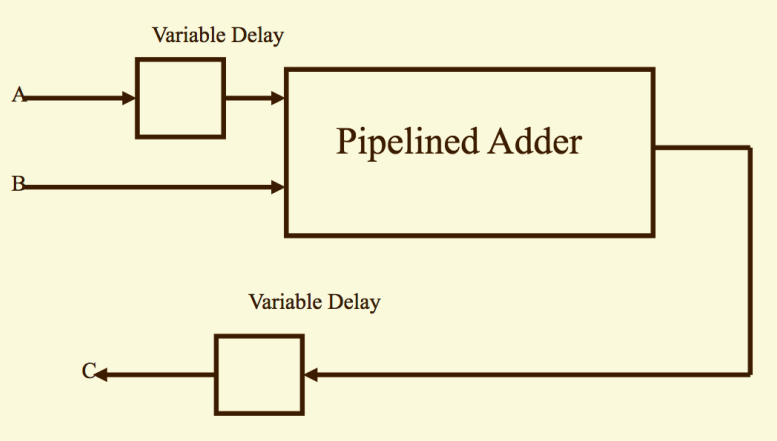


Figure 10.12: Adding artificial delay paths to the pipelined adder.

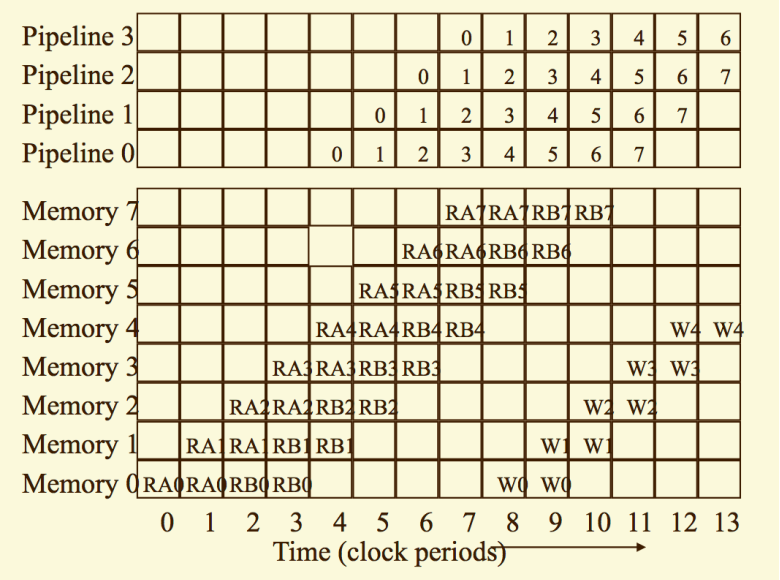


Figure 10.13: Pipelining after the delay path has been added. Note that the pipelines can run concurrently as the data they require has been made available by the delay.

Vector instructions may be issued at the rate of one instruction per clock period; if there is no contention, they will generally be issued at this rate. The first result will appear after some delay, and then each word of the vector will arrive at the rate of one word per clock period. Vectors longer than 64 words are broken into 64 word chunks.

10.3 Stride

Most interleaving schemes take the bottom bits of the address to select the memory bank to use. This is excellent for sequential address patterns of a stride of 1 (that is, each access is one address apart), and acceptable for random access.

However, when the stride is n, where n is a multiple of the number of memory banks (that is, each access is n addresses apart), performance can be extremely poor. This is due to each access using the same memory bank; this effectively cripples any vectorisation capability, resulting in serialisation and thus a dramatic performance drop.

There has been significant research into mitigating the performance impact of $n \propto m$-stride, where m is the number of memory banks. The two primary approaches are to arrange the data to match the stride (software), or to make the hardware insensitive to stride.

10.3.1 Software

Consider a 8x8 matrix; it can be placed in memory in either row or column order. If the program only needs to access the matrix in either row or column order, the order of the matrix can be chosen to guarantee conflict-free access.

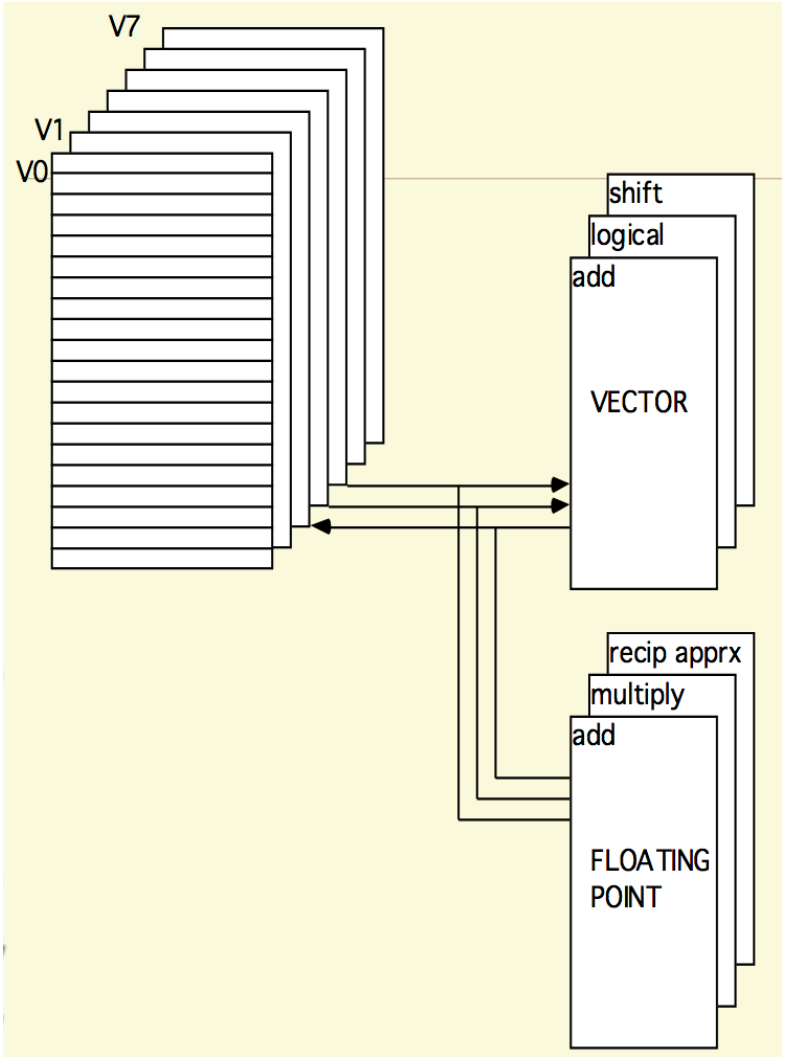


Figure 10.14: CRAY-1 architecture.

Additionally, the matrix can be skewed so that each row starts in a different memory bank. This allows access by row or column order without memory contention. However, the software will need to ensure it accesses the matrix correctly.

10.3.2 Hardware

The hardware can use another function for determining which memory bank to use. This function can be optimised to provide stride-free access for many different strides. Additionally, there are schemes that give optimal packing and do not waste any space.

10.3.3 Other

However, row and column access are not necessarily the only methods of access. Other common modes of access are matrix diagonals and square subarrays of matrices.

The stride to access a diagonal element from the current element is equal to the column stride + 1. However, if m, the number of memory banks, is equal to a power of 2, both column stride and column stride + 1 cannot both be efficient to access; this follows from the fact that both column stride and column stride + 1 cannot be relatively prime to m.

10.4 Vector Algorithms

10.4.1 Gaussian Elimination

Consider the solution of the linear equations given by $Ax = b$, where A is a $N \times N$ matrix and x, b are $N \times 1$ column vectors.

Gaussian elimination is an efficient algorithm for producing upper and lower diagonal matrices L and U such that $A = LU$. [1] Given L and U, it is possible to write $Ly = b$ and $Ux = y$; this can then be used to solve for x using backsubstitution.

This can be implemented as a vectorised algorithm [2]:

```
for i := 1 to N do begin
    imax := index of Max(abs(A[i..N,i]));
    Swap(A[i,i..N],A[imax,i..N]);
    if A[i,i] = 0 then Singular Matrix;
    A[I+1..N,i] := A[I+1..N,i]/A[i,j];
    for k := i+1 to N do
        A[k,i+1..N] := A[k,i+1..N] - A[k,i]*A[i,i+1..N];
end;
```

The breakdown of A can be seen in Figure 10.15.

[1] Well *actually*, this is LU decomposition.
[2] Nope, sorry, didn't want to rewrite this one as pseudocode either.
[3] Yep, still LU decomposition

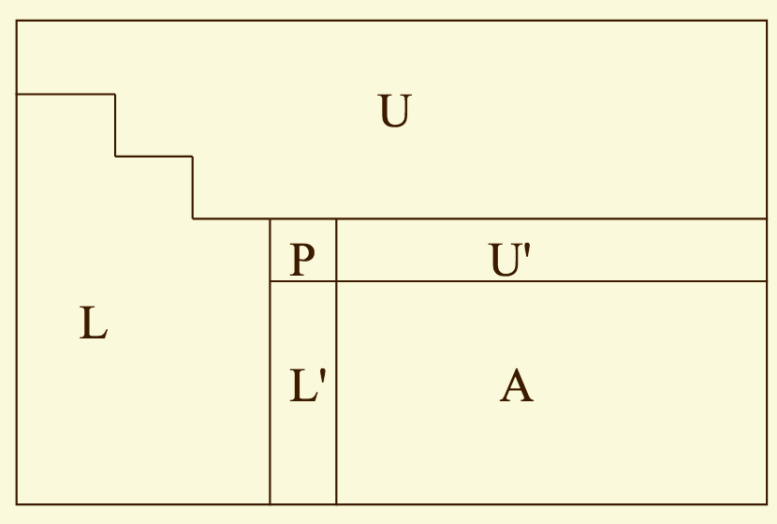


Figure 10.15: Vectorised Gaussian elimination [3]breakdown.

The algorithm as expressed accesses both rows and columns. The majority of the vector operations have either two vector operands, or a scalar and a vector operand, and they produce a vector result. The `Max` operation on a vector returns the index of the maximum element, not the value of the maximum element. The length of the vector items accessed decreases by 1 for each successive iteration.

It is worth noting that row and column access should be equally efficient (see section 10.3). Additionally, the vector pipeline should be able to handle a scalar on an input, and the `Min`, `Max` and `Sum` operators should accept one or more vectors and return a scalar. Finally, vectors may start large, but will get smaller. This may affect code generation due to the cost of initialising the vector units.

10.4.2 Sparse Matrices

In many engineering computations, there may be very large matrices. These are often sparely occupied, with many zero elements; this class of matrices are called *sparse matrices*. Storing sparse matrices traditionally would occupy far too much memory, and be very slow to process.

Many software packages solve this problem by using a software data representation that *does not* store elements with a zero value. They typically provide at least random and row/column sequential access to the sparse matrices.

Matrix multiplication will then consist of adjusting the pointers to the real data and multiplying elements if they share the same index values. In future, hardware may implement a matrix data representation suitable for sparse matrices.

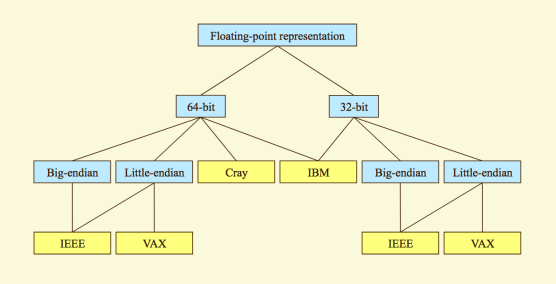


Figure 10.16: The design space for floating-point precision.

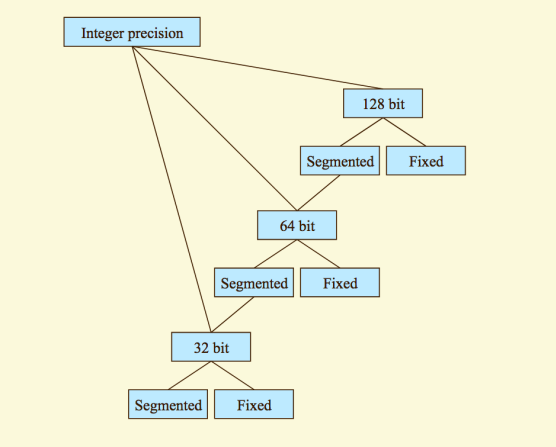


Figure 10.17: The design space for integer precision.

10.5 Random Figures

These figures were at the end of the slides.

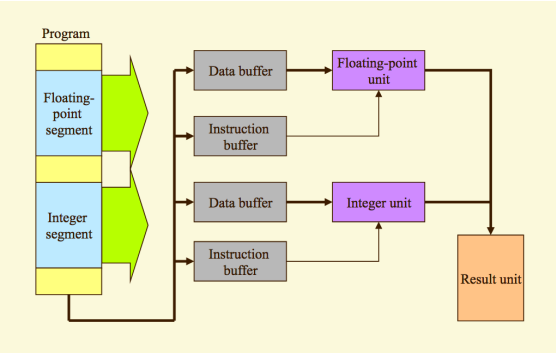


Figure 10.18: Parallel computation of floating-point and integer results.

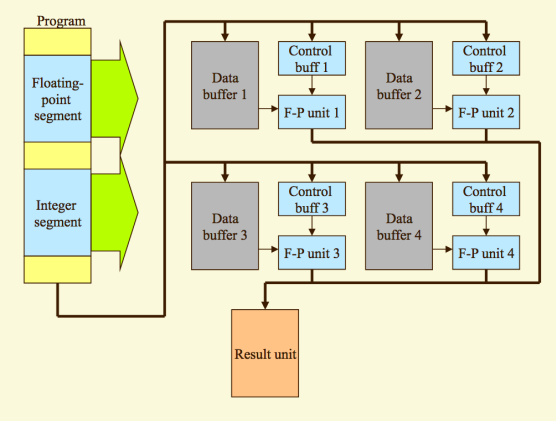


Figure 10.19: Mixed function and data parallelism.

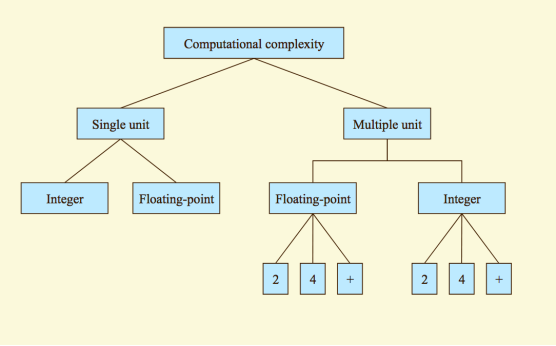


Figure 10.20: The design space for parallel computational functionality.

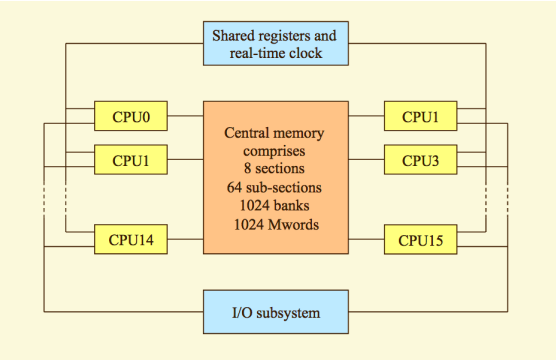


Figure 10.21: Communication between CPUs and memory.

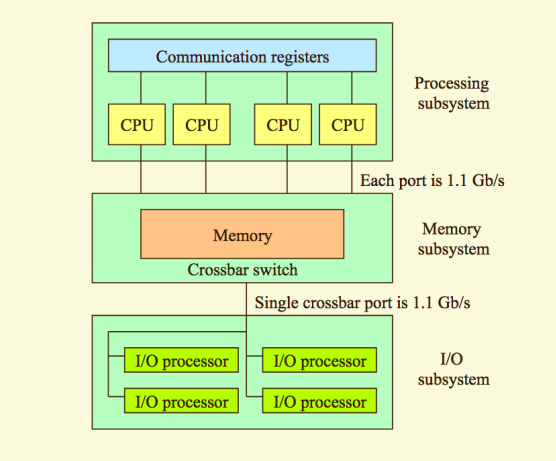


Figure 10.22: The overall architecture of the Convex C4/XA system.

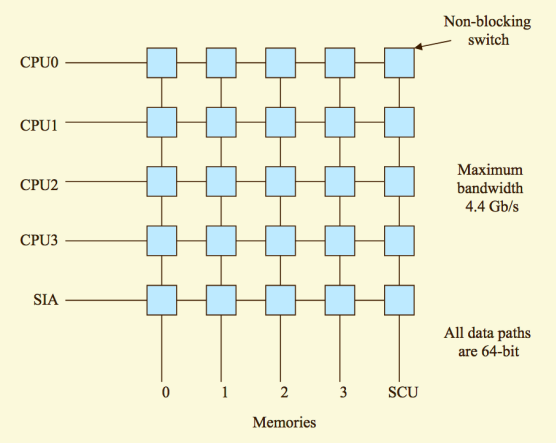


Figure 10.23: The configuration of the crossbar switch.

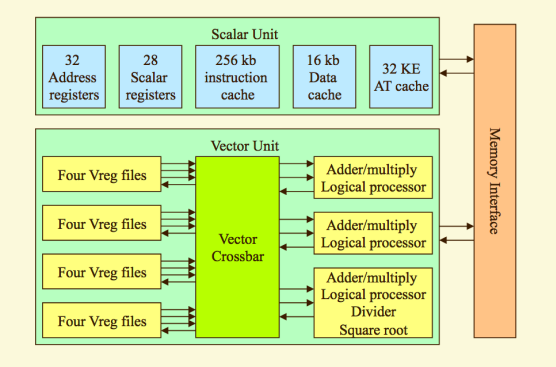


Figure 10.24: The processor configuration. (Some processor, anyway.)

Boy, that was a lot of figures. Wish I knew what they were for!

Chapter 11

Data-Parallel Architectures

Data-parallel architectures are parallel over individual records of data. This can be each cell in a matrix, each pixel of an image, every record of a database, or more.

11.1 Connectivity

We want to support basic computations required at the cell level. As an example:

$$A[i,j] = (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1])/4$$

To achieve this, individual processing units [1] can be connected in a variety of ways:

- **Nearest Neighbours**: Mapping spatially-coherent data (like images) onto SIMD systems. It is common to connect to the cardinal directions, but diagonal connections have also been implemented. This has been applied to massively parallel systems, is scalable, and simple to implement.
- **Trees and Graphs**: Problems expressed as graphs (such as database searching, model matching, expert systems, etc). These have no mathematically regular structure. As a result, the architecture will require reconfigurability. Binary and quadtrees are common. Data bottlenecks are possible when traversing the roots of subtrees.
- **Pyramid**: The pyramid structure is a combination of a mesh and tree, depicted in Figure 11.1. It supports nearest neighbour and quadtree communication; local communication is done with the mesh, while global communication is done with the tree. This is useful for e.g. moving data from one corner to another. It is useful for data stored at multiple resolutions, such as images.
- **Hypercube**: Consists of 2^N processors, each of which has N links; depicted in Figure 11.2. It is fault tolerant, and has shorter pathways than a mesh arrangement.
- **Multistage**
- **Reconfigurable**

[1] or cells, or nodes, or whatever your preferred terminology may be

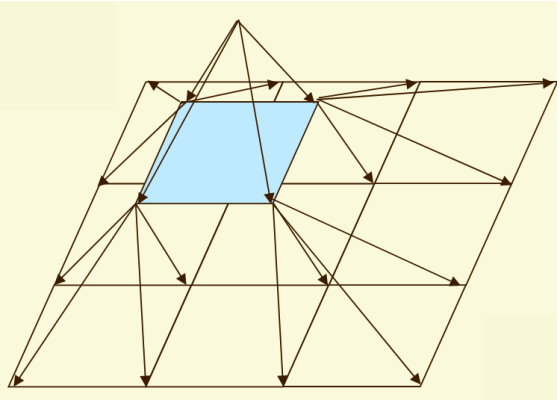


Figure 11.1: Pyramid connectivity scheme.

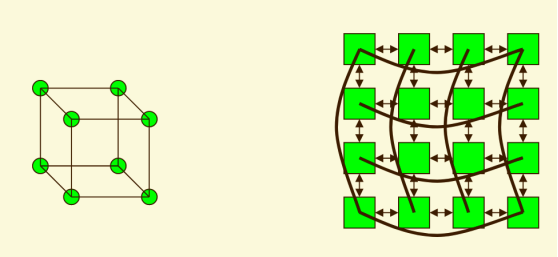


Figure 11.2: Hypercube connectivity scheme.

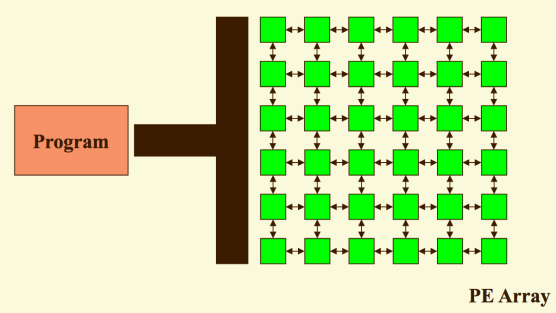


Figure 11.3: SIMD data-parallel architecture.

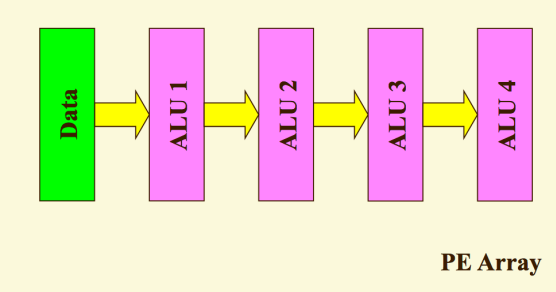


Figure 11.4: Systolic/pipelined architecture.

- **Crossbar**
- **Bus**

11.2 Architectures

There are four primary classes of data parallel architectures: SIMD (Figure 11.3), systolic/pipelined (Figure 11.4), vectorising (Figure 11.5), and associative/neural (Figure 11.6). These are compared in Table 11.1.

Table 11.1: A comparison of the principal characteristics of data-parallel systems.

Property	SIMD	Systolic	Pipeline	Vectorizing	Neural	Associative
Programmability	Good	Fixed	Fixed	Good	Poor	Good
Availability	Good	Poor	Poor	Good	Poor	Poor
Scalability	Good	Fixed	Fixed	Fixed	Fixed	Good
Applicability	Wide	Narrow	Narrow	Wide	Narrow	Wide

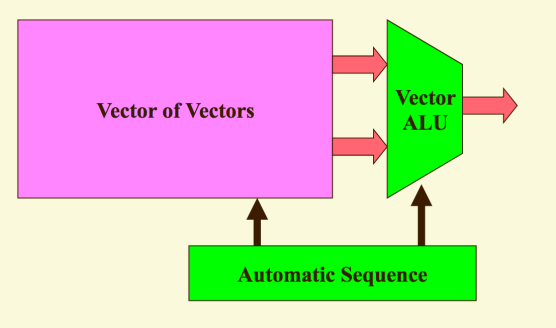


Figure 11.5: Vectorising architecture.

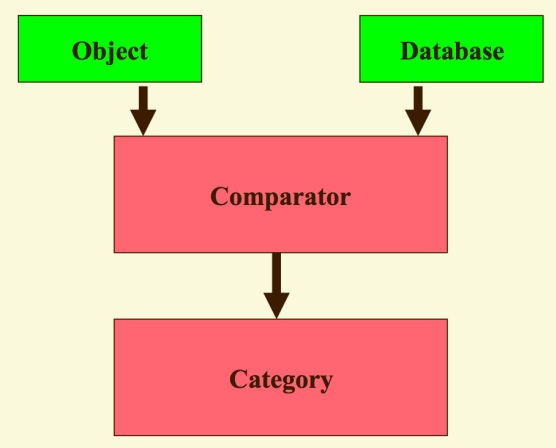


Figure 11.6: Associative/neural architecture.

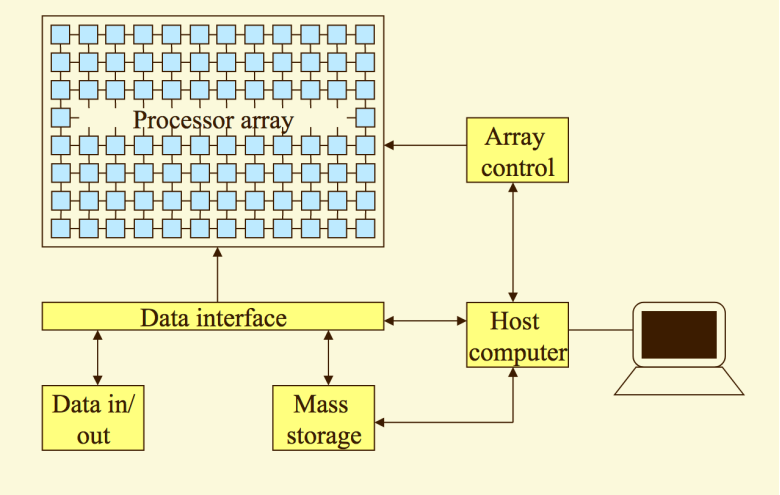


Figure 11.7: The archetypal SIMD system.

11.3 SIMD

SIMD systems offer simplicity of programming, regularity of structure, scalability (both in size and performance), and wide applicability. The archetypal SIMD system is depicted in Figure 11.7. The basic ideas, as formulated in 1958, are as follows:

- Two dimensional array of processing elements
- All processors execute the same instruction simultaneously
- Each processor incorporates local memory
- Processors are programmable
- Data can propagate through the array

When designing a SIMD system to solve a problem, it is important to consider granularity. Fine-grained systems attempt to map one data element to one PE as closely as possible, while coarse-grained systems relax this and allow one PE to handle multiple data elements. This is depicted in Figure 11.8.

Additionally, the connectivity of the system should be explored, as discussed in section 11.1. Potential choices are depicted in Figure 11.9.

The granularity of the system also applies to the data type used for individual data elements, as shown in Figure 11.10. PEs may also have a degree of local autonomy, as explored in Figure 11.11.

11.3.1 Example

The SIMD architecture can be illustrated using the ILLIAC IV computer, which has only one control unit and 64 processing elements (Figure 11.12). Each processing element has a 2K word memory attached. Each processing element performs the same operation, but they may be active or inactive.

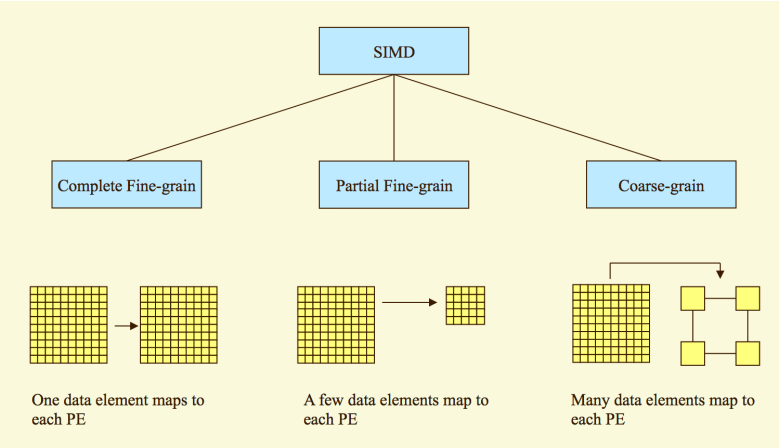


Figure 11.8: Design space for granularity in a SIMD system.

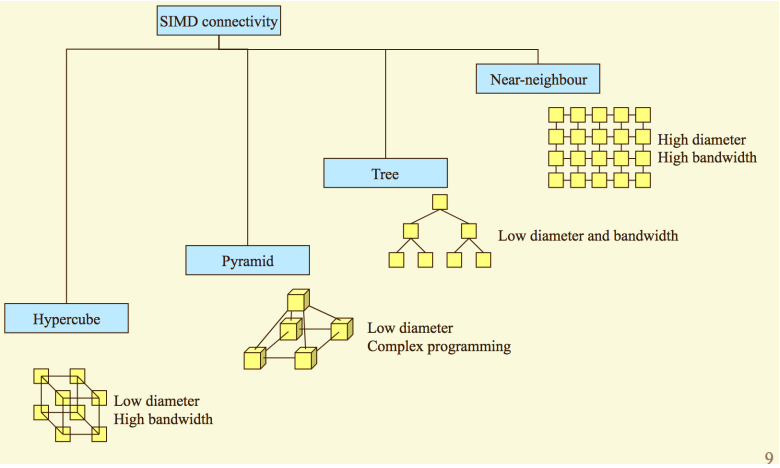


Figure 11.9: Design space for SIMD connectivity.

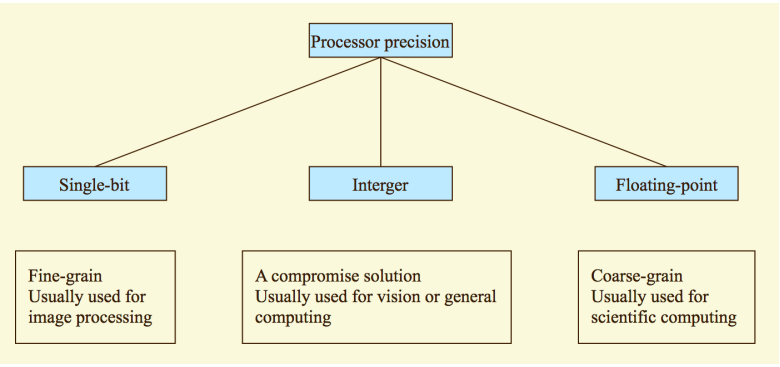


Figure 11.10: Design space for processor complexity in a SIMD system.

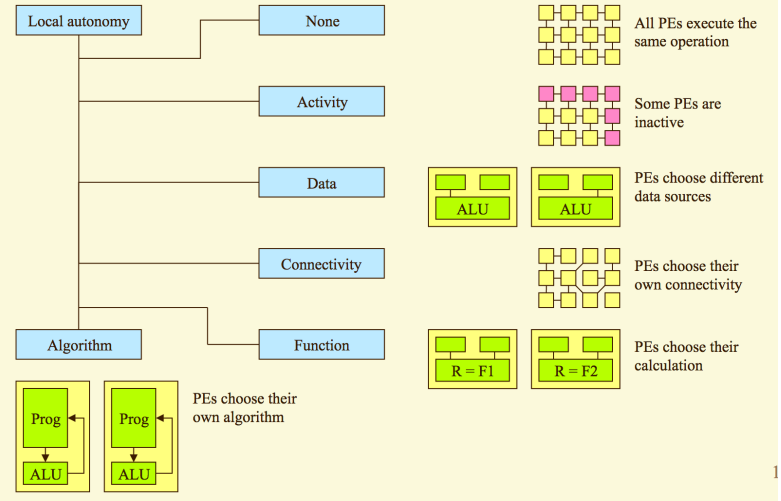


Figure 11.11: Design space for local autonomy.

If they are inactive then they ignore the instruction. In this way the instruction may be applied to certain data elements and not others. For example, there is an instruction which disables all PE's active status if their accumulator is greater than 0.

Each processing element can transfer data to 4 other processing elements using a routing network; each element is also a full ALU capable of executing a wide range of arithmetic and logic functions.

Each element has 6 registers:

- **A**: accumulator
- **B**: second operand
- **R**: routing register
- **S**: temporary storage
- **X**: index register
- **D**: mode register (active/inactive)

Each PEM contains 2048 64 bit words of data. PEMi can only be addressed from PEi. Thus a PE can only change data in its own PEM. Data can be passed from PE to PE via the routing network, shown in Figure 11.13. The control unit bus allows instructions to be fetched from PEMs. Data can be broadcast to all PEs using a broadcast bus. Data is passed between processing elements using the routing network.

Array addition $A(i) = B(i) + C(i)$ for $1 < i \leq N = 64$ can be handled in a vectorised fashion, offering a speedup of N times. The instruction stream is simple: load from location 2, add to location 1, store in location 0. If $N < 64$, then some of the processing elements must be disabled. If $N > 64$, the code must loop until N is exhausted.

Consider $A(i) = B(i) + A(i-1)$ for $2 < i \leq 64$. This is a serial loop, and therefore cannot be run concurrently. Expanding out the loop gives:

- $A(2) = B(2) + A(1)$

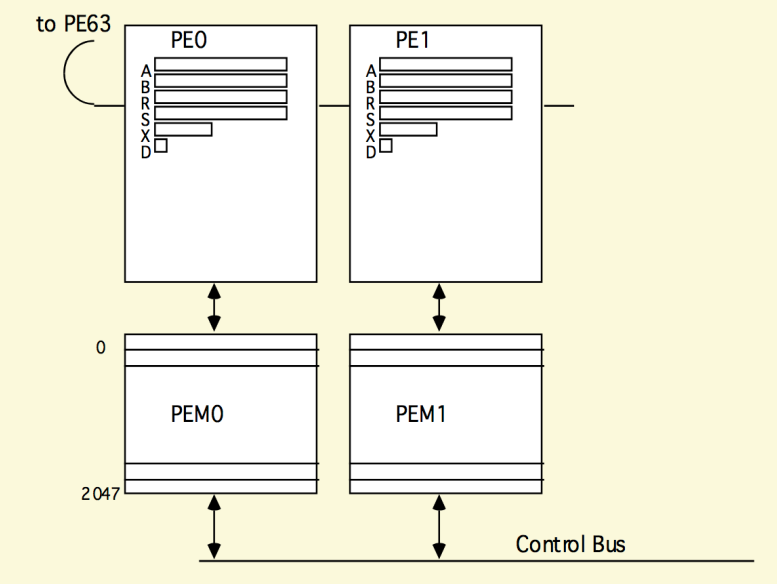


Figure 11.12: ILLIAC IV machine structure.

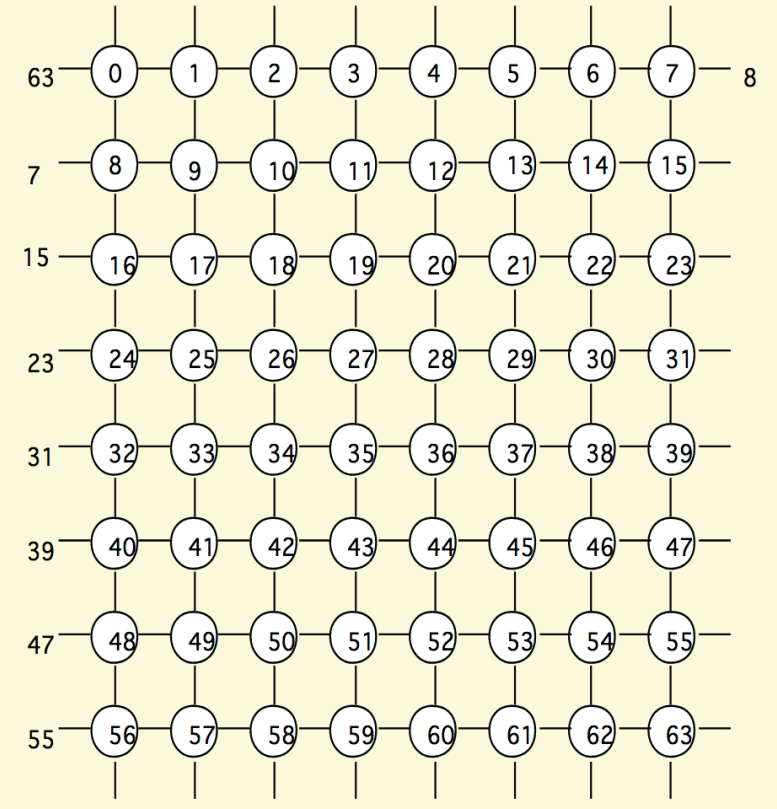


Figure 11.13: ILLIAC IV connectivity.

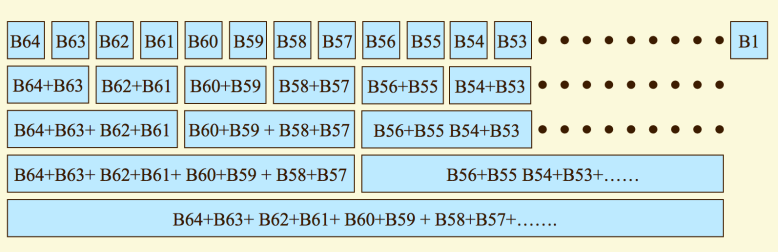


Figure 11.14: Recursive doubling algorithm.

- $A(3) = B(3) + A(2)$
- ...
- $A(64) = B(64) + A(63)$

However, this is a recurrence relationship, so it can be rewritten as such:

- $A(2) = B(2) + A(1)$
- $A(3) = B(3) + B(2) + A(1)$
- $A(4) = B(4) + B(3) + B(2) + A(1)$
- ...

This rearrangement allows calculation of A independently from the loop:

$S \leftarrow A(1)$
for N = 2, 64 **do**
  $S \leftarrow S + B(N)$
end for
$A(N) = S$

This reformulation is still a sequential algorithm, however. *Recursive doubling* can be used to execute in log time, as shown in Figure 11.14.

This algorithm is shown below [2]:

```
1. Enable all PE's (Turn on all PE's)
2. All PE's load RGA from location B
3. i = 0
4. All PEs load RGR from their RGA
5. All PEs ROUTE their RGR contents 2i to the right.
6. j = 2i -1
7. Disable all PE's number 0 through j
8. All enabled PE's ADD RGA to RGR
9. i = i + 1
10. if i < 6 goto 4
11. Enable all PEs
12. All PEs store RGA to A
```

[2] I made the executive decision to not rewrite any algorithms longer than 6 lines. Maybe for version 2!

Chapter 12

MIMD Architectures

In a distributed memory MIMD machine, the processor/memory pairs are replicated and are then connected via an interconnection network. In a shared memory MIMD machine, the processors and memories are replicated independently and are then connected via an interconnection network.

MIMD machines can be classified into a hierarchy, as can be seen in Figure 12.1.

12.1 Distributed Memory

In a distributed memory system, each processor has its own local memory that is not shared with other processors (Figure 12.2). The access to this local memory module is much faster than remote memory. The hardware accesses remote memory through load/store primitives and a message passing layer. Cache memory is also used for local memory traffic. Messages can be memory-memory or cache-cache.

Distributed memory offers the following advantages:

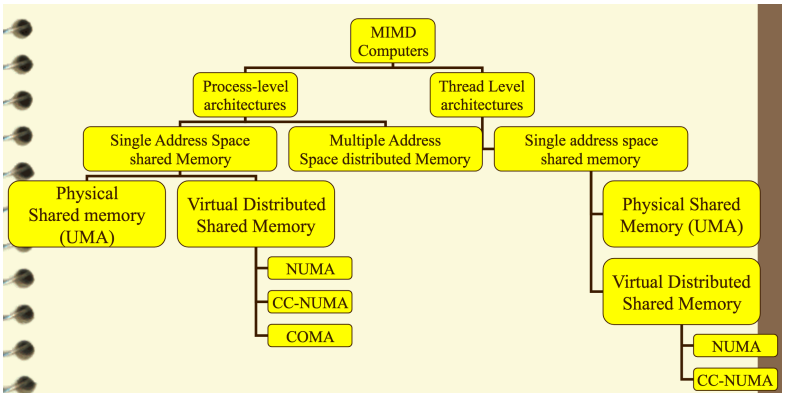


Figure 12.1: Classification of MIMD computers.

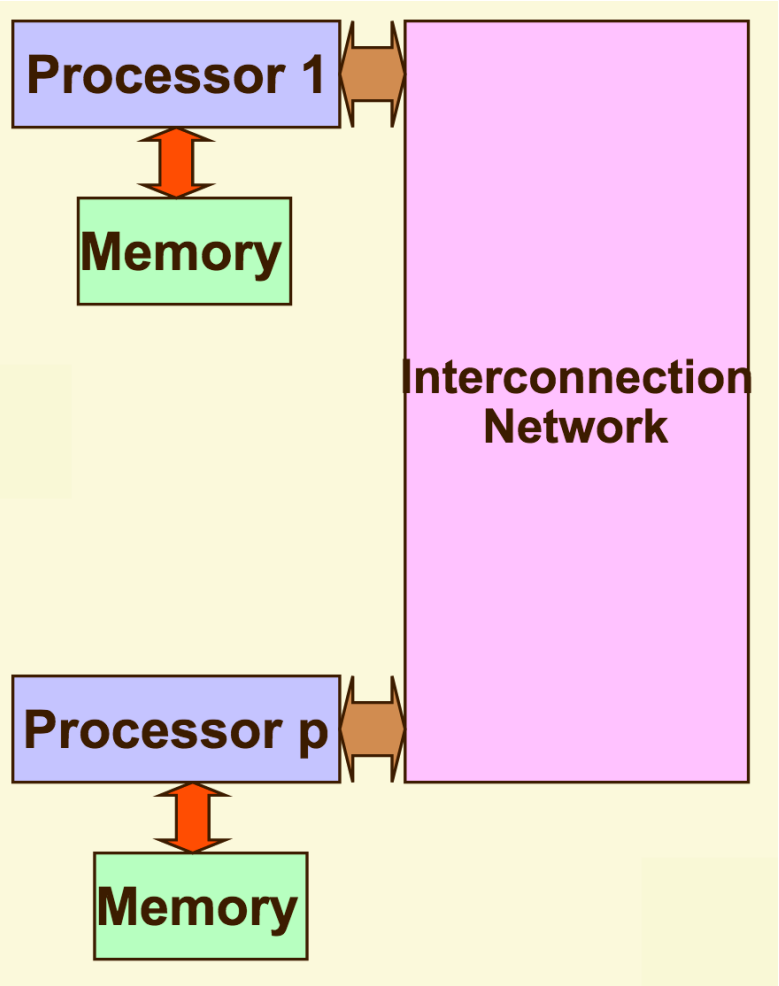


Figure 12.2: Distributed memory system.

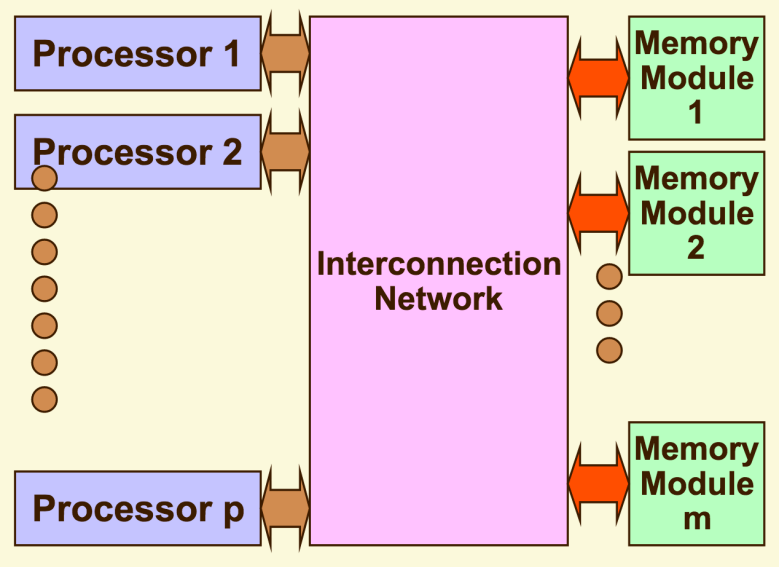


Figure 12.3: Shared memory system.

- Local memory traffic offers less contention than in shared memory.
- Highly scalable.
- Sophisticated synchronization features like monitors or semaphores are not needed. Message passing serves the dual purposes of data transmission and synchronisation.

However, it is subject to the following disadvantages:

- Load balancing is difficult.
- Message passing can lead to synchronisation failures, including deadlock (i.e. the sequence of BlockingSend → BlockingReceive, BlockingReceive → BlockingSend can occur).
- Entire data structures need to be copied, which can be an intensive process.
- The overhead for small messages is high relative to the size of the message.

12.2 Shared Memory

In a shared memory system, all processors have equal access to shared memory modules (Figure 12.3). Local caches reduce memory traffic, network traffic, and memory access time. As memory is synchronised between the processes, load/store is indivisible.

Shared memory offers the following advantages:

- There is no need to partition code or data; the system handles this automatically.
- There is no need to move data explicitly.
- Existing programming languages and/or compilers can be used.

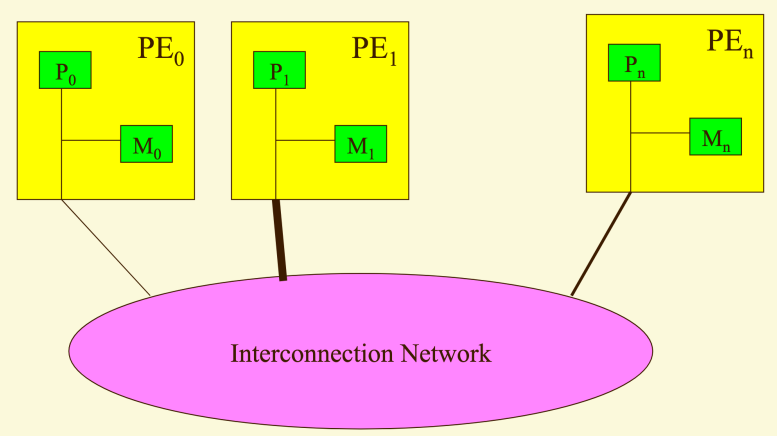


Figure 12.4: NUMA.

However, it is subject to the following disadvantages:

- Synchronisation of resources is difficult.
- Lack of scalability; IPC [1] becomes a bottleneck. This can be addressed by using a high-throughput and low-latency network, using cache memory (but this will lead to cache coherence issues), or using a distributed shared memory architecture.

12.3 Distributed Shared Memory

There are three choices for design that can be made:

- Non-uniform memory access (NUMA, Figure 12.4): Cray T3D
- Cache-coherent non-uniform memory access (CC-NUMA, Figure 12.5): Convex SPP, Stanford DASH
- Cache-only memory access (COMA, Figure 12.6): KSR-1

12.4 Problems of Scalable Computers

A scalable computer needs to be able to tolerate and hide the latency of remote loads, as shown in Figure 12.7; this is made worse if the output of one computation relies on another to complete.

It also needs to be able to tolerate and hide idling brought about by synchronisation among processors.

To tolerate latency, a variety of techniques can be used:

[1] I assume this is interprocess communication, but it could also be instructions-per-clock. Overloaded acronyms are *fun!*

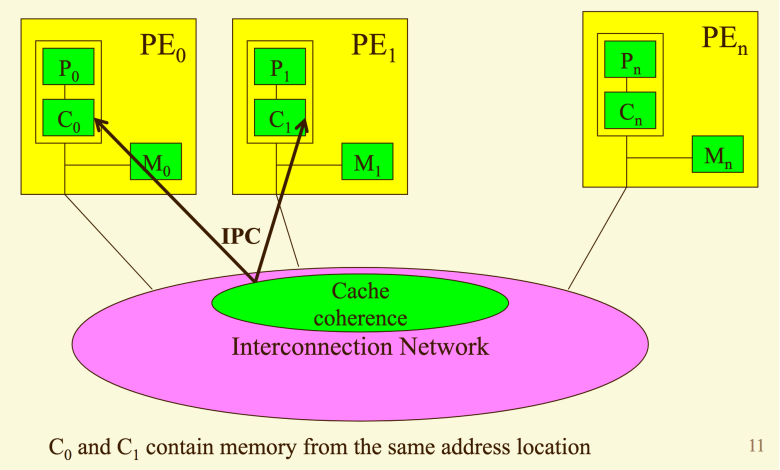


Figure 12.5: CC-NUMA.

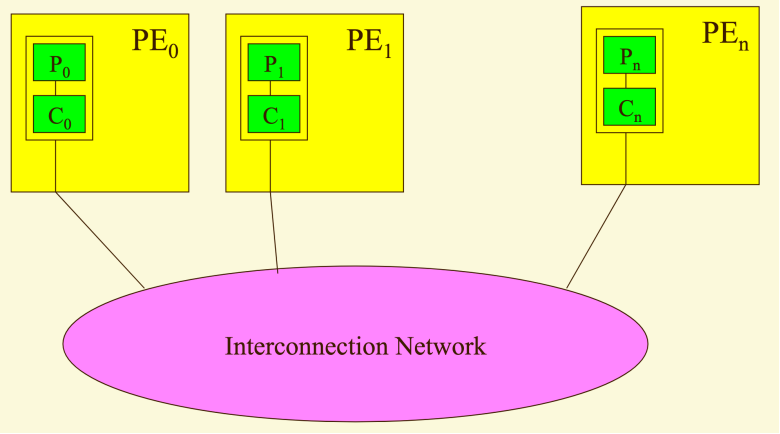


Figure 12.6: COMA.

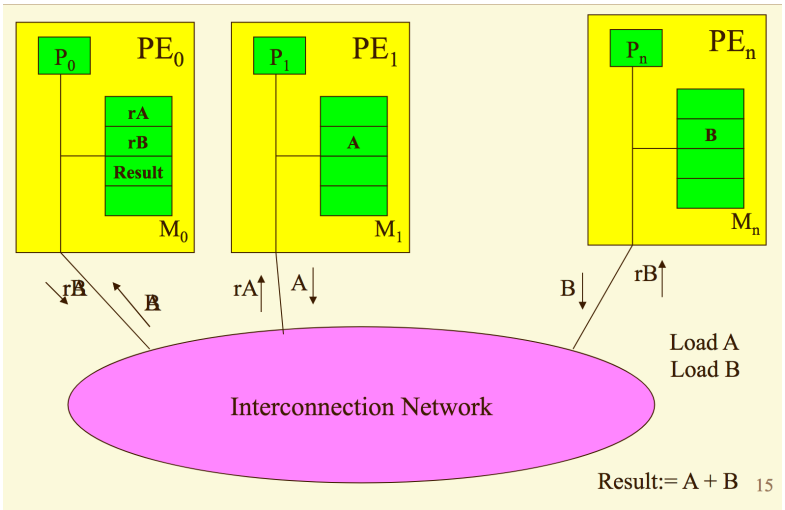


Figure 12.7: Tolerating remote loads.

- **Cache memory**: Maintaining local copies of remote memory that is cheap to access. This lowers the cost of remote access, but introduces the cache coherence problem.
- **Prefetching**: Prefetching relevant data before it is needed. This is already present to some extent, so the cost of implementation is low. However, it will increase network load.
- **Threads + fast context switching**: Using local multithreading to minimise the time spent idling. This is acceptance that a remote action will take a long time, and using this time effectively to cover the overhead.

However, these solutions do not solve synchronisation issues; latency-tolerant algorithms must be used fo rthat.

12.5 Design issues of scalable MIMD

Scalable MIMD is subject to several design issues that need to be considered.

The processor design must consider pipelining and issues that arise from running instructions in parallel. In doing so, concerns with atomic data access, prefetching, cache memory, message passing and more may be raised.

The interconnection network design must be designed to be scalable, high-bandwidth and low-latency.

Memory in the system may become a complex subject if a shared memory design is used. For performance, the shared memory will need local caches on each processor. However, maintaining these caches and keeping them up to date with the remote memory results in the *cache coherency* problem, in which the cache may desynchronise and cause issue.

Finally, the I/O subsystem needs to be designed to handle parallel I/O. A parallel system that cannot actually receive or submit data quickly enough will be bottlenecked.

Chapter 13

Distributed Memory MIMD

A distributed memory MIMD system works on the process level instead of threads (which can be modelled as large grain threads). Messages are passed between processes to facilitate synchronisation and data movement. The important questions are "how to best partition the parallel program?" and "how to best map processes onto the processors to minimise IPC [1]?"

The design space revolves around reducing the time and frequency of thread switches, which are higher in MIMD machines than in multithreaded architectures. The frequency of computation, as well as the processor time spent on communication, are important factors. The complete design space is depicted in Figure 13.1.

The design focus is on the organisation of the communication subsystem (especially with regards to the network) and hardware support for message passing. This is often dependent on the algorithms and software in use.

Communication can be conducted over links or channels; the communication graph is mapped onto the underlying network. Nodes consist of a computational processor + private memory, a communication processor, and a router/switch unit.

[1] Well, it's definitely not instructions-per-clock, but I don't think it's interprocess communication either. Maybe interprocessor communication? As mentioned, acronyms are ***fun!***

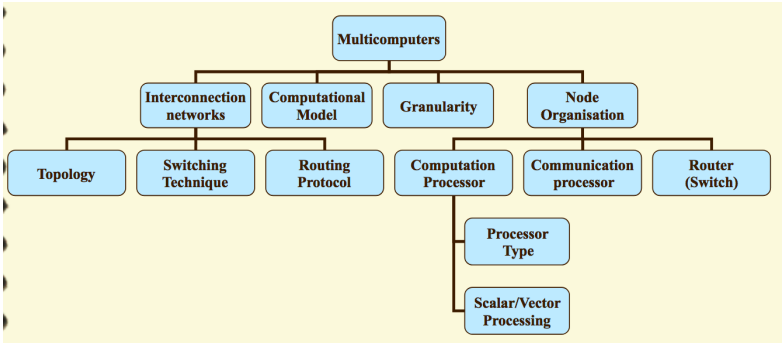


Figure 13.1: Complete design space for a distributed memory MIMD system.

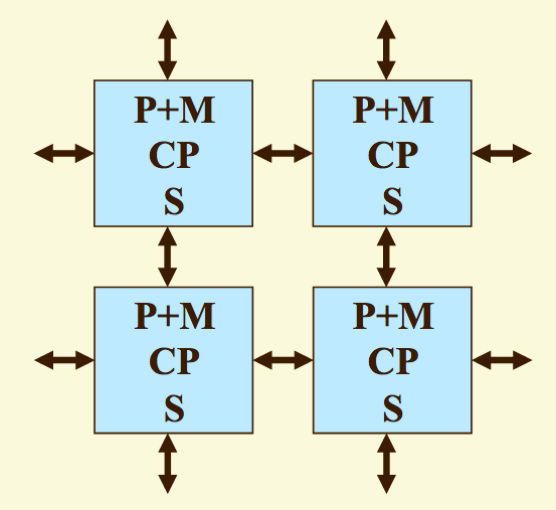


Figure 13.2: Node organisation: first generation.

A multiprocessor can be classified with regards to the following criteria:

- How the interconnection network is organised.
  - Topology
    - Switching technique
    - Routing Protocols
- How the three components of the nodes are composed and integrated:
  - First generation: No communications processor (Figure 13.2).
  - Second generation: Separate comms switching units (Figure 13.3, Figure 13.4).
  - Third generation: Separate processors (Figure 13.5).
- The message passing computational model supported by the multicomputer:
  - Library calls (MPI, PVM, etc)
  - CSP support
  - Push, pull or active messages
- The optimal grain of computation:
  - Coarse-grained: Traditional languages.
  - Medium-grained: CSP.
  - Fine-grained: Dataflow.

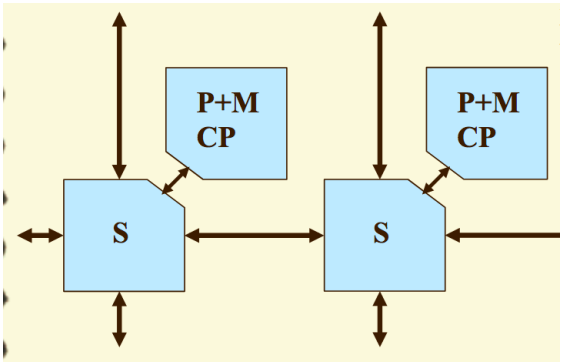


Figure 13.3: Node organisation: second generation variant 1.

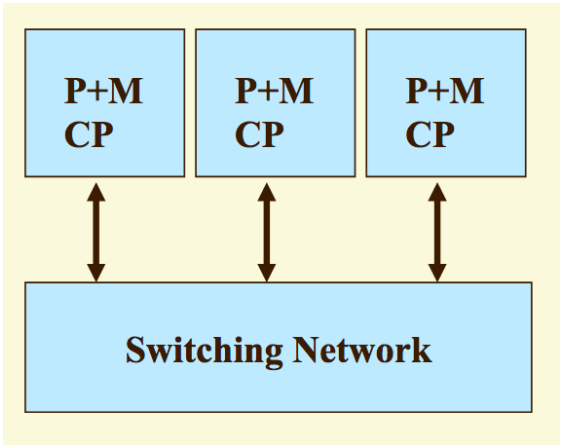


Figure 13.4: Node organisation: second generation variant 2.

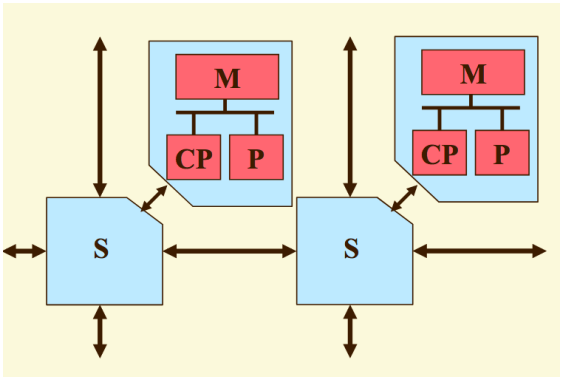


Figure 13.5: Node organisation: third generation.

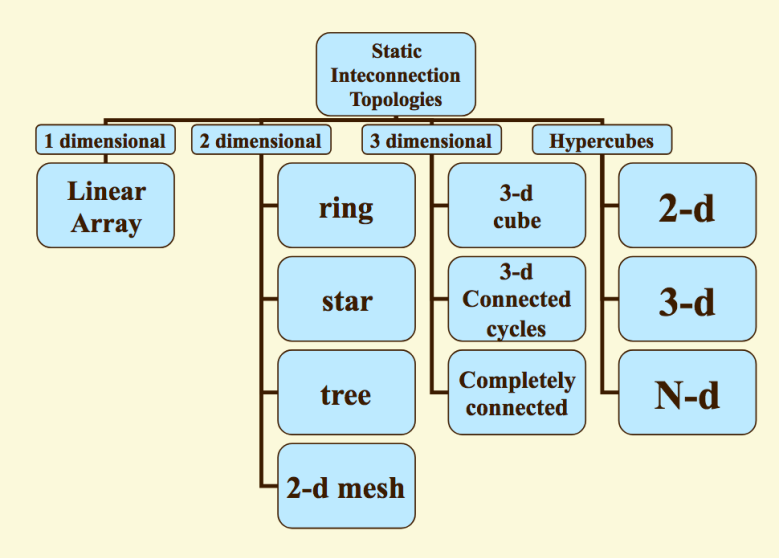


Figure 13.6: Interconnection topologies.

13.1 Interconnection Network

The design of the interconnection network is critical to the performance of MIMD machines. Bandwidth and latency are both critical parameters. These networks are characterised by the following:

- **Network Size (N)**: Number of nodes in the network.
- **Node Degree (d)**: Number of IO links per node.
- **Network diameter (D)**: Maximum number of hops to get from any node to any other.
- **Bisection Width (B)**: The minimum number of links broken to break network in 2 equal halves; related to wiring density and overall bandwidth.
- **Arc connectivity**: Minimum number of arcs removed to break into two disconnected networks; it is a measure of fault tolerance and contention.
- **Cost**: Number of communication links in the network.

The possible topologies are shown in a hierarchy in Figure 13.6. Examples of these are shown in Figure 13.7, Figure 13.8, and Figure 13.9.

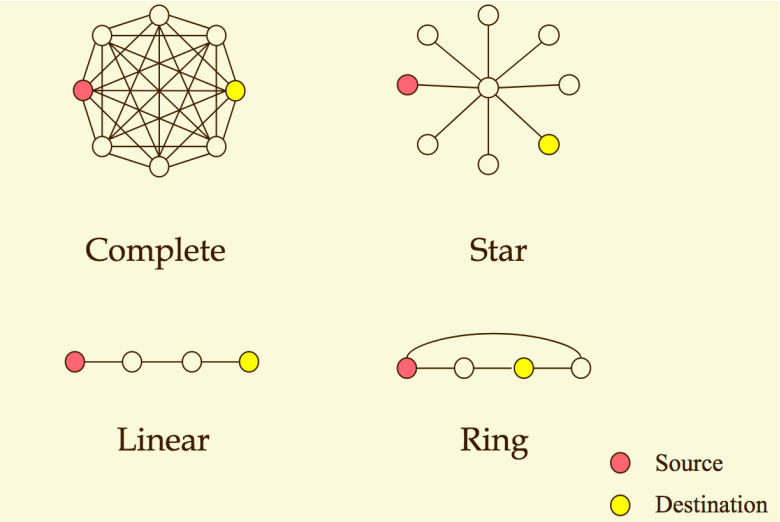


Figure 13.7: Complete, Star, Linear and Ring network arrangements.

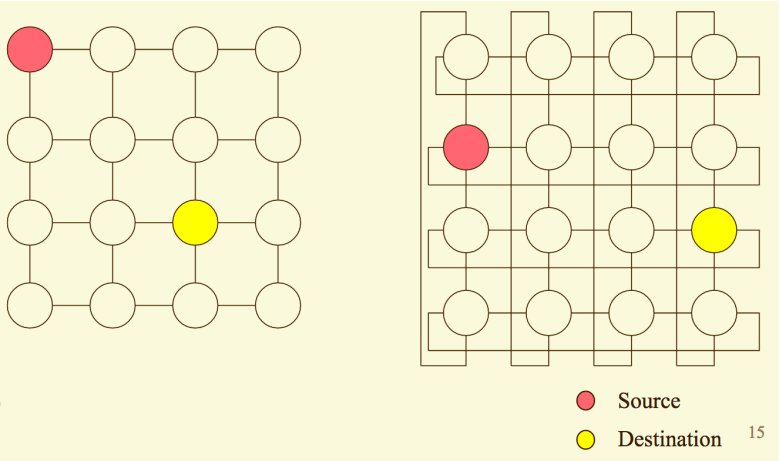


Figure 13.8: Mesh and Torus network arrangements.

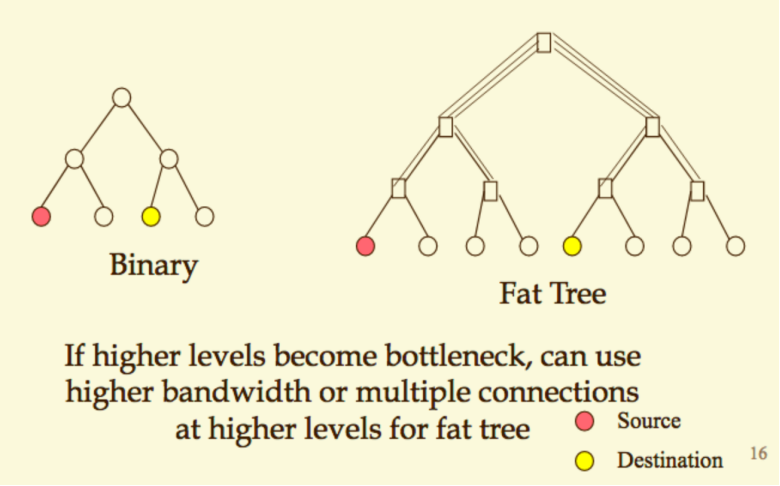


Figure 13.9: Tree arrangements.

Table 13.1: Static network arrangements comparison.

Topology	**Node degree**	**Diameter**	**Bisection width**	**Arc connectivity**	**Cost**
Linear array	1 or 2	$N-1$	1	1	$N-1$
Ring	2	$N/2$	2	2	N
Star	1 or $N-1$	2	1	1	$N-1$
Binary tree	1, 2, or 3	$2\log((N+1)/2))$	1	1	$N-1$
2-D mesh	2, 3, or 4	$2(N^{1/2}-1)$	$N^{1/2}$	4	$2(N-N^{1/2})$
2-D wraparound	4	$N^{1/2}$	$2N^{1/2}$	4	$2N$
3-D cube	3, 4, 5, or 6	$3(N^{1/3}-1)$	$N^{2/3}$	3	$2(N-N^{2/3})$
Hypercube	$\log(N)$	$\log(N)$	$N/2$	$\log(N)$	$(N\log(N))/2$
Completely connected	$N-1$	1	$N^2/4$	$N-1$	$N(N-1)/2$

The HyperCube arrangement arranges 2^N elements such that each element has N connections to other elements (see Figure 13.10). Each element is connected to its *boolean* neighbour - that is, the node number differs by only one bit. While it is hard to build scalable fabric with this arrangement, it features good nearest neighbour connectivity, and any element in the arrangement is at most N hops away from another element.

The MultiStage arrangement is a hybrid of the *cross-bar* arrangement and *buses*. They can provide contention-free access (given that multiple requests are not made to the same module) in the general case, but it is worth noting that contention can still occur at certain traffic levels. Unlike buses, they are scalable and can be expanded to accommodate larger systems. As MultiStage is a combination of two arrangements, there are many possible topologies; the simplest is the shuffle exchange network, shown in Figure 13.11.

Various static network arrangements are compared in Table 13.1.

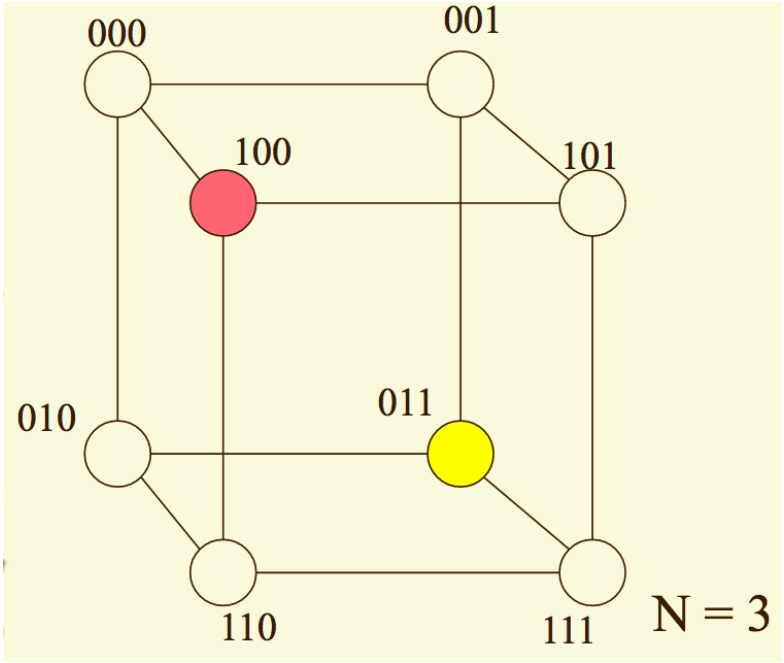


Figure 13.10: HyperCube network arrangement.

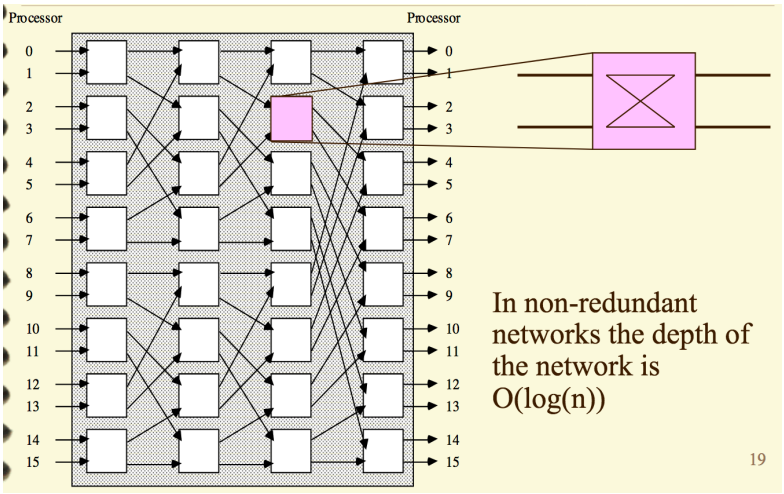


Figure 13.11: MultiStage network arrangement (shuffle exchange).

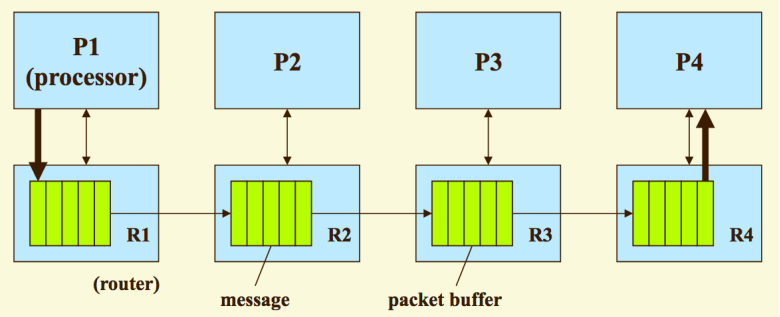


Figure 13.12: Packet switching.

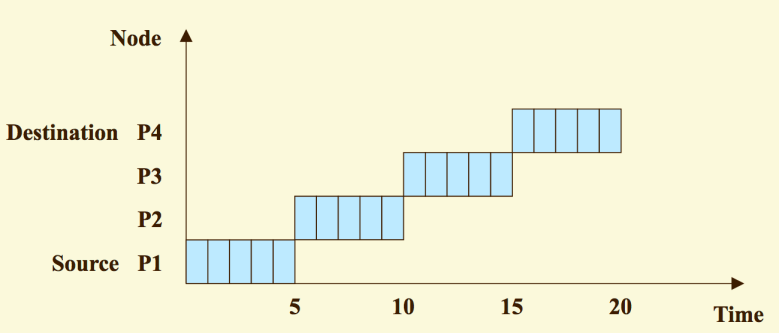


Figure 13.13: Network latency in packet switching.

13.2 Switching Techniques

The network structure defines pathways for the data, but switching techniques are responsible for defining how to move data through the network - that is, removing the message from the input buffer and placing it in the output buffer.

13.2.1 Packet switching

Data is divided into packets, which are small packages of data. They include a header, body and tail. The entirety of each packet is stored at each node before being sent on, shown in Figure 13.12. This means the latency can be described by

$$\text{Latency} = (\text{Packet length} \times \text{Distance})/\text{Channel Bandwidth}$$

and is shown in graph form in Figure 13.13.

13.2.2 Circuit switching

A path between the source and destination is constructed through three distinct phases:

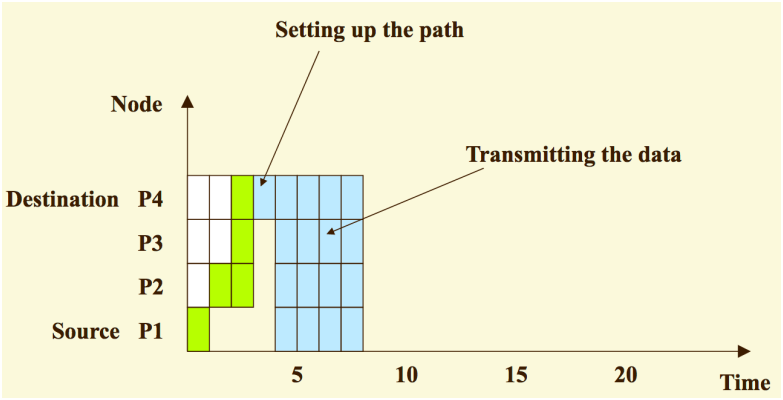


Figure 13.14: Network latency in circuit switching.

1. **Circuit establishment phase**: A probe is sent through the network, opening up the path.
2. **Transmission phase**: The data is sent through the routing path, assuming that it has already been established.
3. **Termination phase**: The circuit that was constructed is then ripped up.

The latency can be described by

$$\text{Latency} = (\text{Probe Length} \times \text{Distance} + \text{Message Length})/\text{Channel Bandwidth}$$

and is shown in graph form in Figure 13.14.

The advantages of circuit switching include not needing to build packets, avoiding the use of buffering, and only having to route once per message (instead of per packet).

13.2.3 Virtual cut-through

Virtual cut-through is a combination of packet switching and circuit switching, shown in Figure 13.15. The packets are broken into Flow Control Digits (FLITS). If channels are free, FLITS are forwarded in circuit switched mode; otherwise, the message is broken and buffered.

The latency can be described by

$$\text{Latency} = (\text{Length of header FLIT} \times \text{Distance} + \text{Message Length})/\text{Channel Bandwidth}$$

and is shown in graph form in Figure 13.16.

As long as the required channels are free, the message will be forwarded between the nodes FLIT-by-FLIT. If the channel is busy, FLITS are buffered at intermediate nodes. If the buffers are large enough, the entire message is buffered and the system will behave like packet switching. If the buffers are not large enough, the message will be buffered across several nodes holding the links.

Wormhole routing is a special-case of virtual cut-through where the buffer size is the size of a flit; it is depicted in Figure 13.17. It is capable of packet replication, which is used to implement

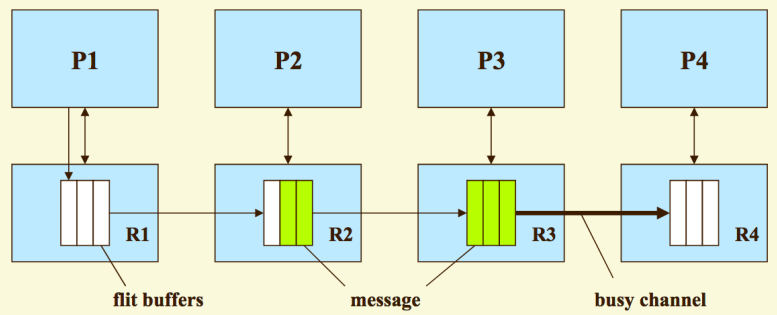


Figure 13.15: Virtual cut-through.

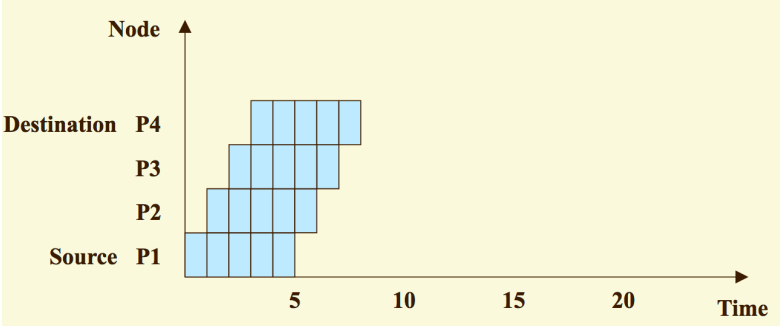


Figure 13.16: Network latency in virtual cut-through/wormhole routing.

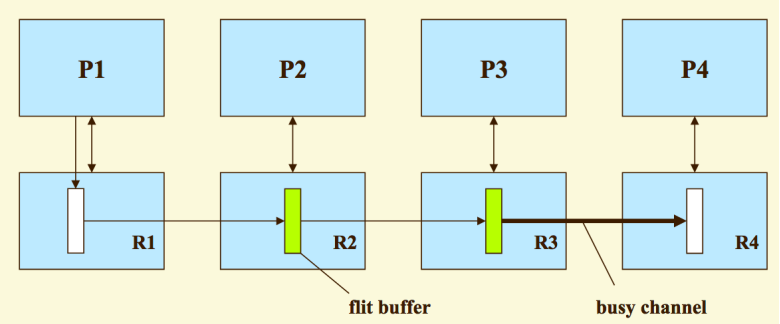


Figure 13.17: Wormhole routing.

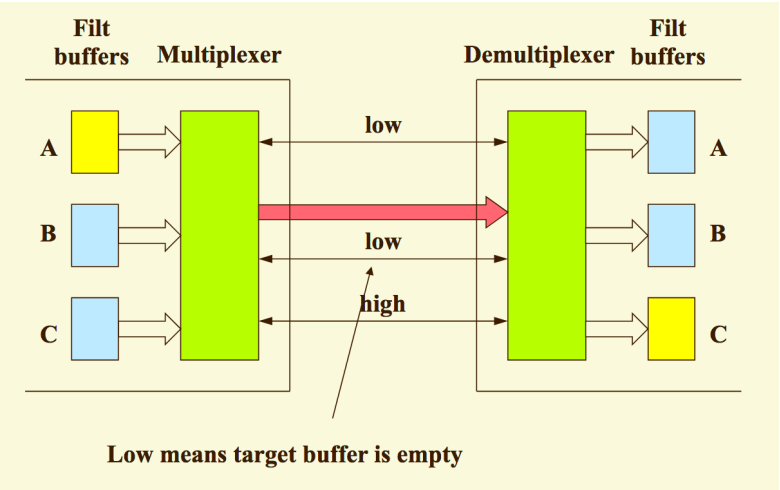


Figure 13.18: Virtual channels in routing.

broadcasts - packets can be sent out on multiple output channels concurrently, which is not possible with circuit switching.

Wormhole routing is better than circuit switching when switch contention is high, as circuit switching blocks the path for the whole message. Wormhole routing breaks into FLITS, which leads to higher efficiency; however, if the path is blocked, the intermediate channels will also be blocked. Additionally, the contention may occur on intermediate nodes within the path.

13.2.4 Virtual Channels

Virtual channels allow multiple independent messages to share the same physical channels. This is done by *multiplexing* multiple FLITS onto a single channel, such that the number of FLIT buffers exceeds that of the number of channels (depicted in Figure 13.18).

Virtual channels allow for increased network throughput by reducing the idle time of physical channels. They can also be used to avoid deadlocks. They allow the mapping of the logical topology of the network onto the physical network; as the mapping is not exact, some logical

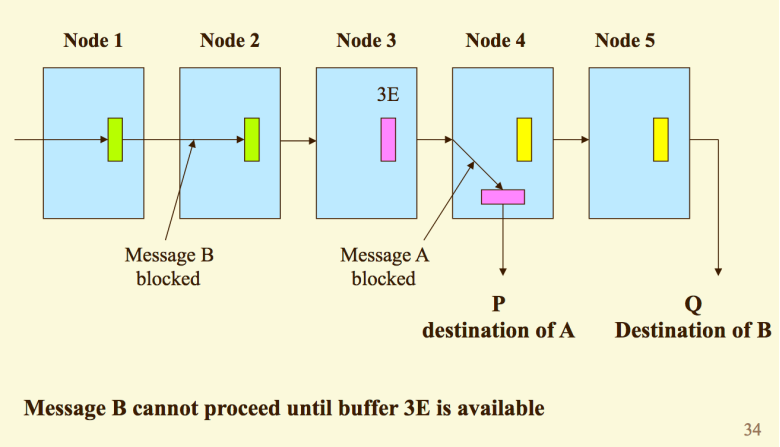


Figure 13.19: Routing without virtual channels.

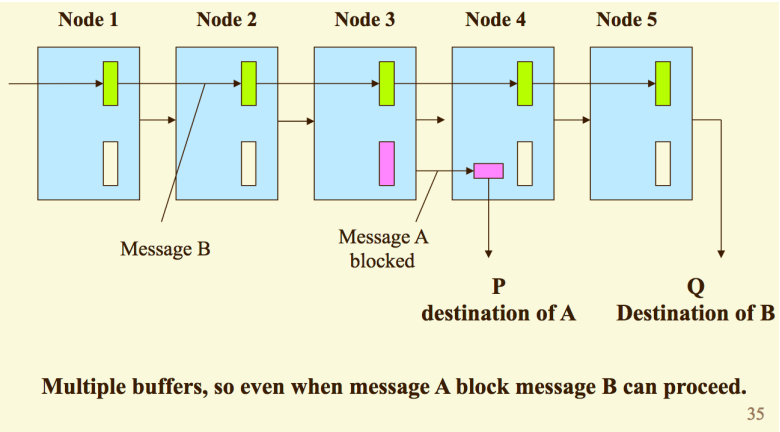


Figure 13.20: Routes with virtual channels.

channels can be reserved for critical functions like debugging and monitoring.

A comparison between routing without virtual channels and routing with virtual channels is seen in Figure 13.19 and Figure 13.20 respectively.

13.2.5 Deadlocks

Deadlocks can occur in scenarios like Figure 13.21, where each node is waiting on a message from its neighbour. This is a *cyclic dependency graph*, constructed from the interconnection network and routing algorithm. If cycles are present, deadlock may occur.

Deadlocks can be avoided by:

- pre-empting messages via rerouting (that is, using an alternate path for one message so that the cycle is broken)

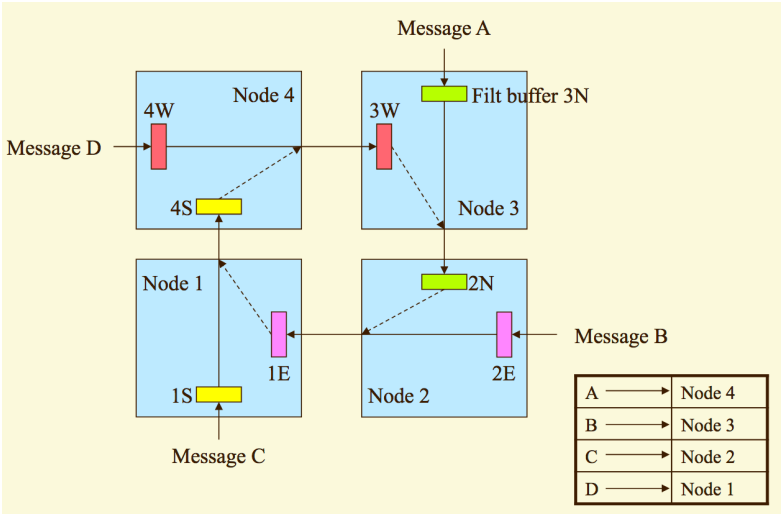


Figure 13.21: Typical deadlock scenario.

- pre-empting messages by discarding them and transmitting them on an alternate path
- application of virtual channels; splitting physical channels into multiple virtual channels is sufficient to break cycles in the dependency graph

13.2.6 Livelocks

When a livelock occurs, a message is endlessly propagated through the network but fails to ever reach its destination. This can occur in flow control policies that avoid collisions on channels and buffers by misrouting messages to find an alternative path. Packet switching and virtual cut-through are very sensitive to livelock, but circuit switching and wormhole routing are typically livelock free.

13.3 Routing protocols

Routing protocols can be put in a hierarchy, as depicted in Figure 13.22.

13.3.1 Deterministic Routing

In deterministic routing, the route depends only on the source and destination pair and no other information.

13.3.1.1 Street Sign Routing

In street sign routing, the message header contains the complete path information; the routing information is stored in the form of direction changes. This is shown in Figure 13.23.

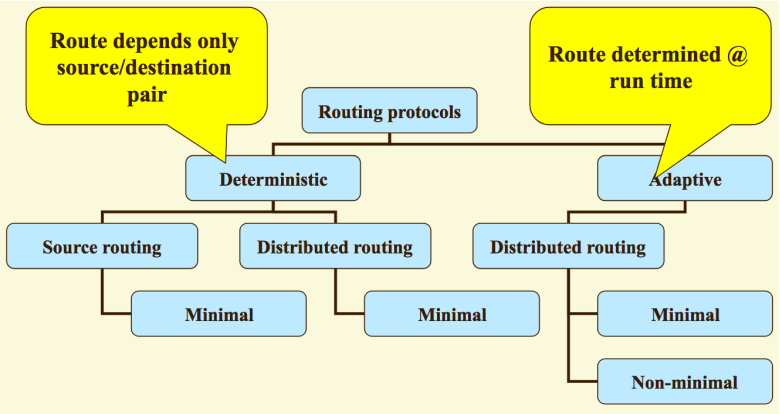


Figure 13.22: Routing protocol hierarchy.

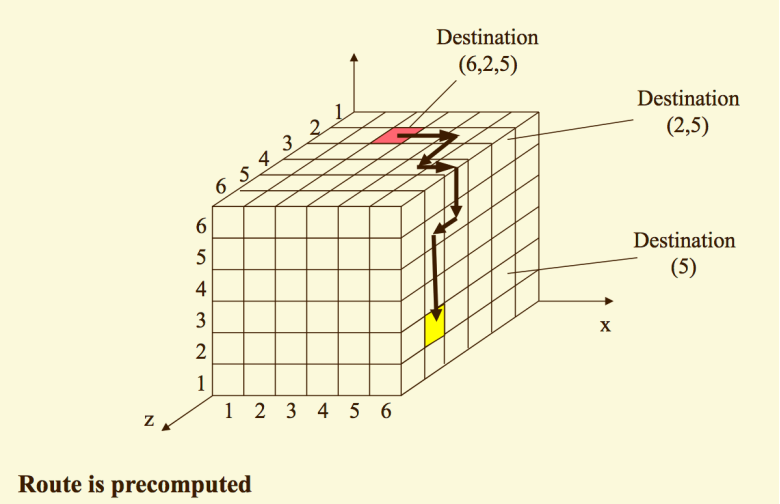


Figure 13.23: Street sign routing.

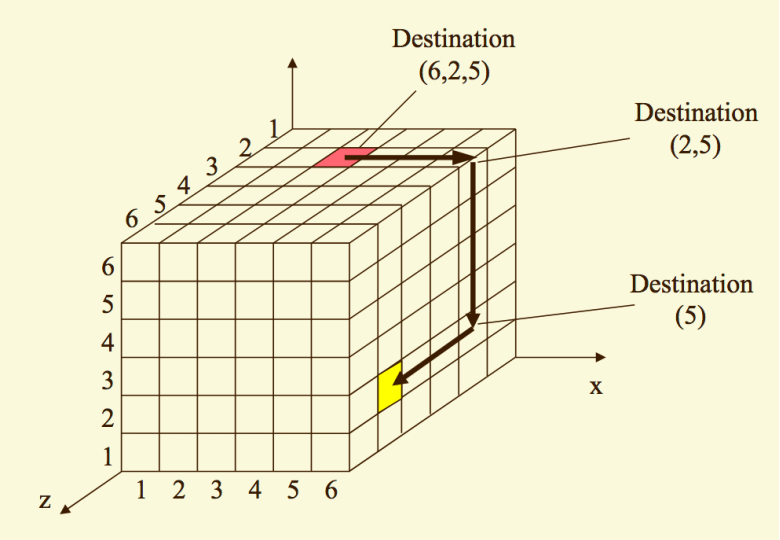


Figure 13.24: Dimension-ordered routing.

13.3.1.2 Dimension-ordered routing

In dimension-ordered routing, the message travels along a certain dimension until it is aligned with the destination on that dimension. It will then proceed to do the same for each dimension until it reaches the destination. This is shown in Figure 13.24.

13.3.1.3 Table lookup routing

In table lookup routing, each node contains a routing table. This table contains the identifier of the neighbouring node to which the message should be routed; there is one entry per destination address. The tables are precomputed and can be large.

13.3.2 Adaptive Routing

Adaptive routing protocols can be put in a hierarchy, as depicted in Figure 13.25.

13.3.2.1 Profitable vs Misrouting

Profitable protocols can only move closer to the goal and represent a conservative view. They will produce the minimum length path, are free from livelock, and are easier to prove deadlock free.

Misrouting protocols can take other choices and represent an optimistic view. They select a non-profitable channel in belief that it will lead to profitable ones. They are more fault tolerant as they do not have to use particular channels for their transmission.

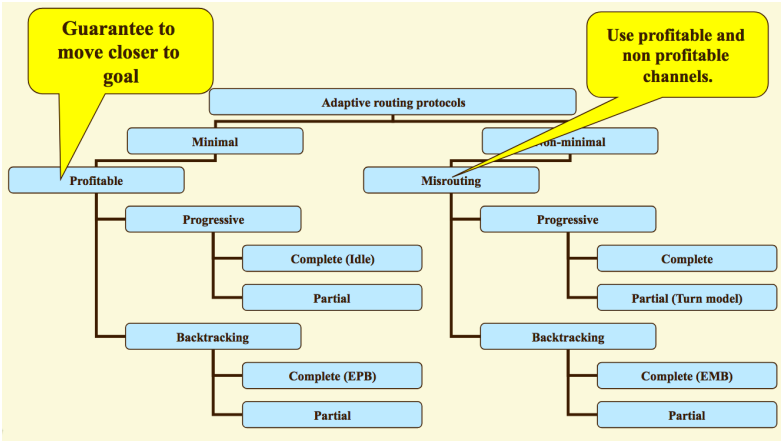


Figure 13.25: Adaptive routing protocols.

13.3.2.2 Progressive vs Backtracking

Progressive protocols cannot backtrack, and may deadlock.

Backtracking protocols can systematically explore all paths and are deadlock free, but are more complex. They need to know where they have been before. If there are no profitable channels available (depending on whether the protocol is profitable or misrouting), the protocol will need to wait for a channel, try a non-profitable free channel, and if that fails step back and try again.

13.3.2.3 Complete vs Partial

A complete routing protocol can explore all channels, while a partial protocol can only use selected channels. A partial choice of channels allows a deadlock-free route to be computed.

> Add nodes used the same ordering of channels and explore them in this order.
>
> e.g. West first: Route west as first choice then choose other directions adaptively

13.4 Machine design choices

MIMD systems can be built from processors that support any of the following:

- Fine-grain message passing: J-Machine
- Medium-grain message passing: Transputer (built around communicating sequential processes, or CSP)
- Coarse-grain message passing: COTS; networks have higher latency, and the processors don't need to understand message passing

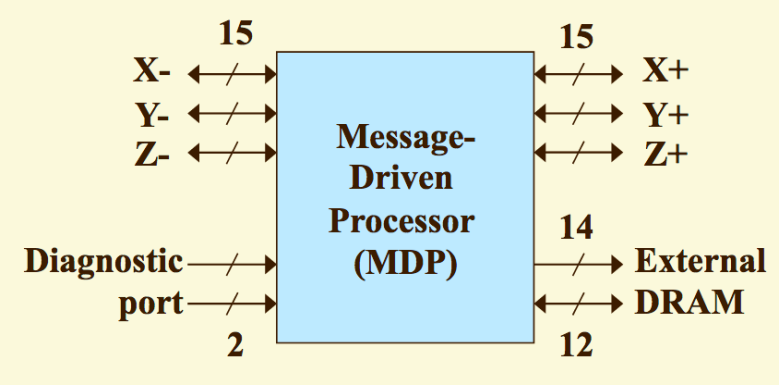


Figure 13.26: J-Machine processor architecture.

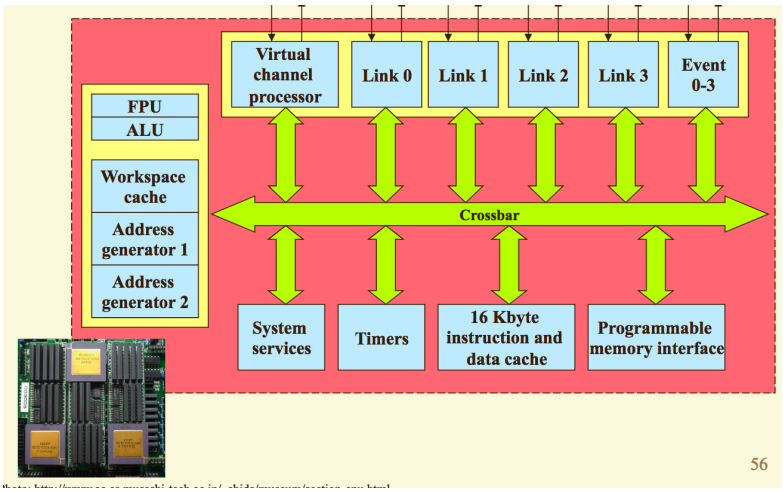


Figure 13.27: Medium-grain Transputer architecture.

13.4.1 Fine-grain

Fine-grain concurrent processing reduces overhead and latency of receiving a message through hardware support. Hardware support additionally reduces context-switching time (through the use of multiple registers and hardware support for synchronisation). Object-oriented programming, when implemented in hardware, provides `call` and `send` operations to objects. The architecture of the J-Machine processor is shown in Figure 13.26.

13.4.2 Medium-grain

Message passing is synchronous; neither the sender or receiver can continue without each other. The first process reaching a channel operation must be suspended until the partner arrives. Transputer is an example of a medium-grain computer; it is depicted in Figure 13.27.

Running processes need to be statically mapped onto hardware; this includes links and processors.

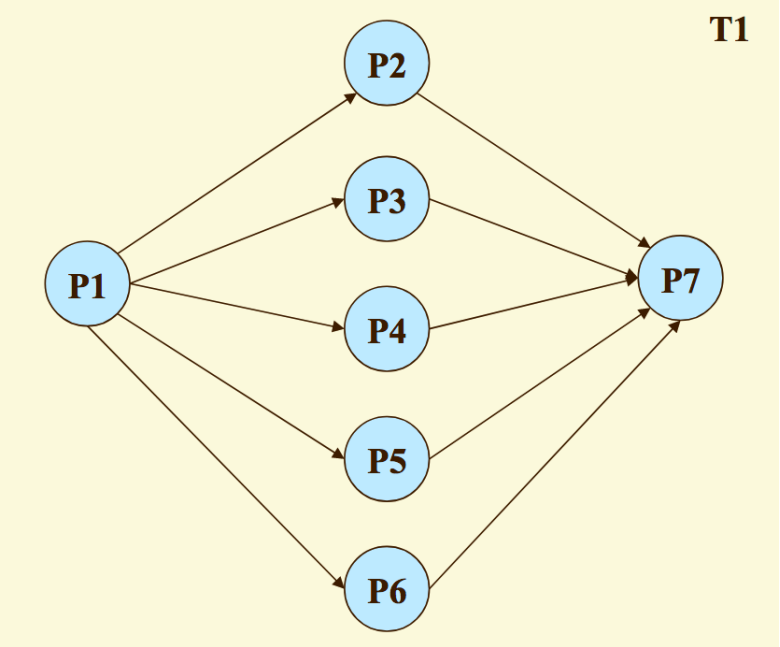


Figure 13.28: A single Transputer process allocation diagram.

All processors are mapped onto a single Transputer; there are no external links used. This leads to an allocation diagram similar to the one in Figure 13.28.

However, if more than one Transputer is used, it may not be possible to provide enough physical links to connect between all of them (as depicted in Figure 13.29); this is solved using virtual links.

Transputers are traditionally connected using links to form a mesh; this mesh is routed using a C104 router chip, which provides a 32-way crossbar. This is depicted in Figure 13.30.

13.4.3 Coarse-grain

Finally, the design space for coarse-grain third-generation multicomputers is reproduced in Figure 13.31.

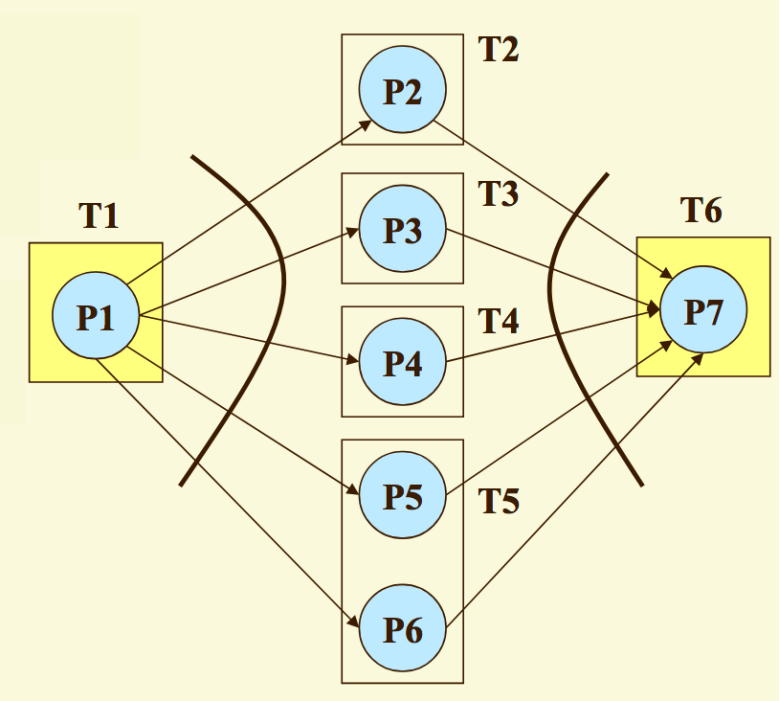


Figure 13.29: Too many physical links for a multi-Transputer arrangement.

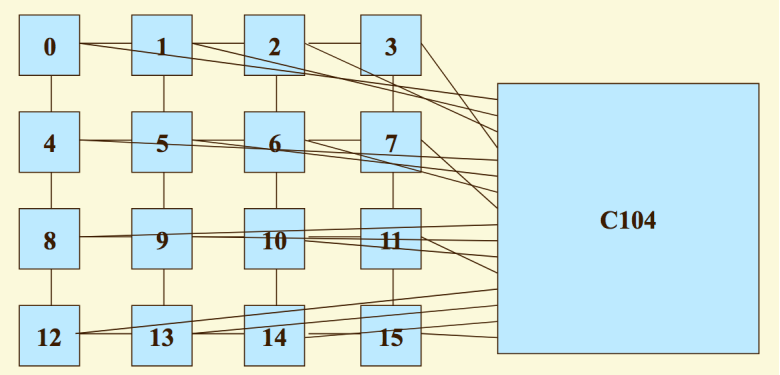


Figure 13.30: Transputer mesh.

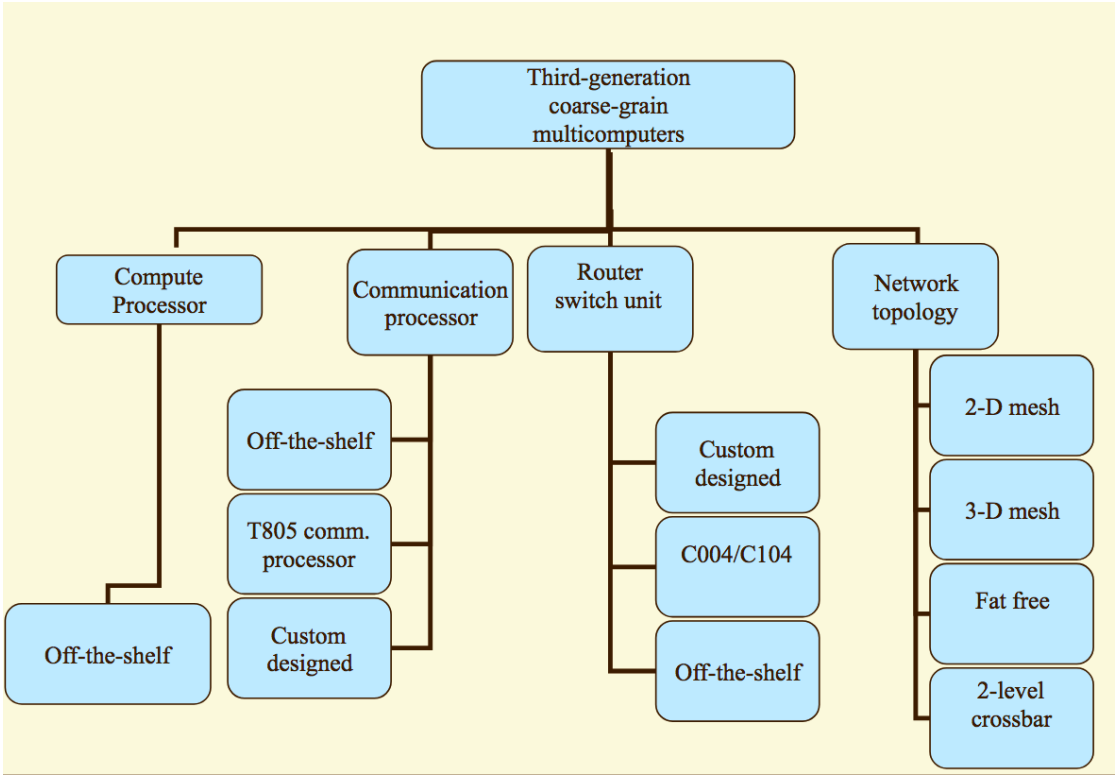


Figure 13.31: Design-space of third-generation coarse-grain multicomputers.

Chapter 14

Superscalar Processing

Superscalar processors rearrange the incoming instruction stream so that instructions can be executed independently across multiple execution units, allowing for increased throughput.

However, in doing so, they must handle parallel decoding, issues that arise from superscalar execution, and must preerve sequential consistency of execution and exception processing.

14.1 Parallel Decoding

In a superscalar machine, multiple instructions are issued concurrently (Figure 14.1). Decoding these instructions can be difficult, especially when instructions are multiple words. Inter-instruction dependencies need to be checked before an instruction can be issued.

The decoder is on the critical path of execution, so it must be run. However, CISC (Complex Instruction Set Computer) instructions are hard to decode in parallel, as the length of an instruction is variable (Figure 14.2); instruction boundaries are normally determined sequentially.

Additionally, register dependencies need to be considered (Figure 14.3). Sequential issuing of instructions can use ready bits, set during issuing, to detect dependencies; parallel issuing requires combinational logic to determine where dependencies lie.

The pre-decoding stage is executed differently depending on whether the architecture is RISC (Reduced Instruction Set Computer) or CISC.

If it is RISC, the instruction class, the type of resources required and sometimes the target address of the branch is required; this may add 4 bits per instruction when the instruction is written into the instruction cache (I-cache). This is shown in Figure 14.4.

For a CISC system, the instruction boundaries and location of opcodes and operands must be determined; the AMD K5 architecture adds 70% extra bits to an instruction during the fetching stage.

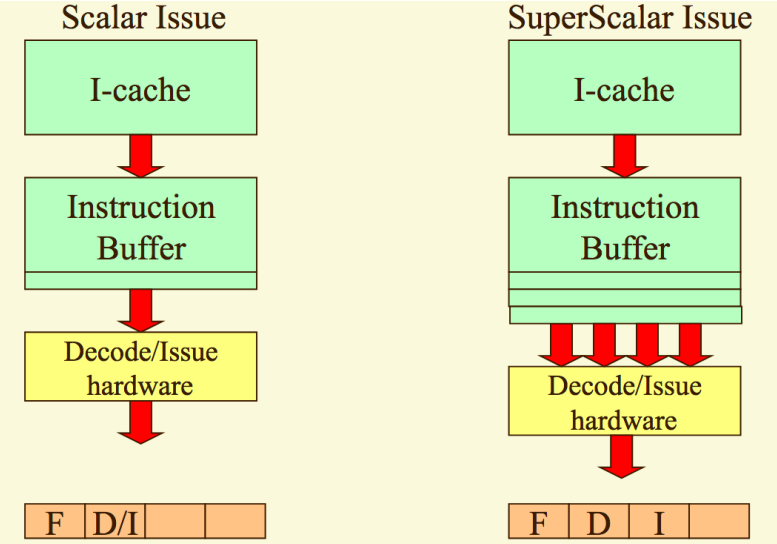


Figure 14.1: Parallel decoding in a superscalar system.

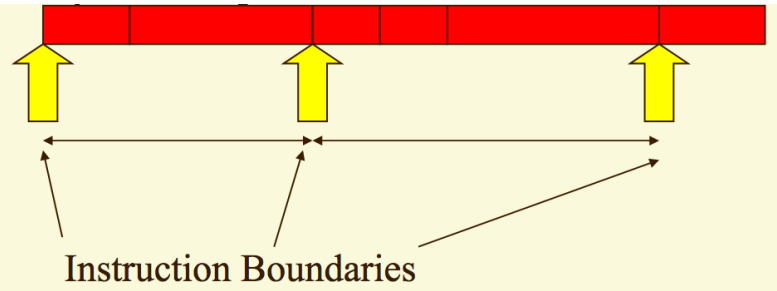


Figure 14.2: Variable instruction boundaries in a CISC system.

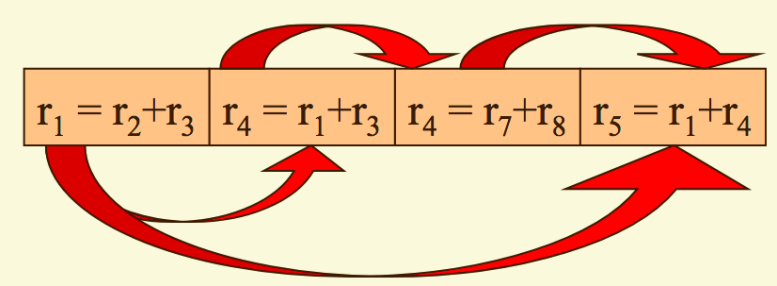


Figure 14.3: Register dependencies.

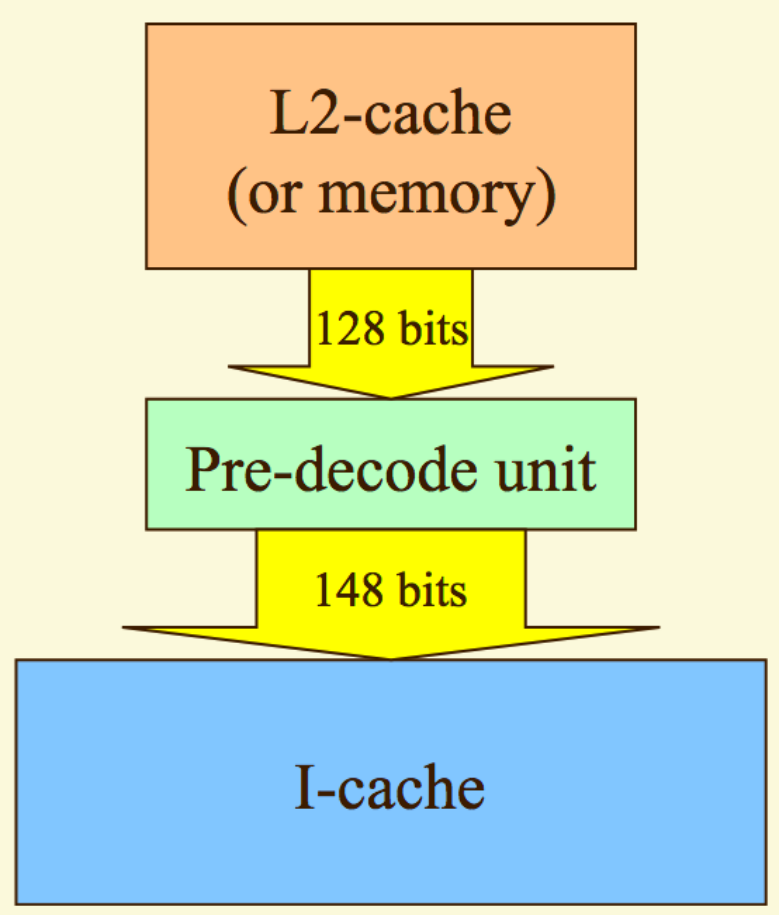


Figure 14.4: Pre-decode unit adding bits.

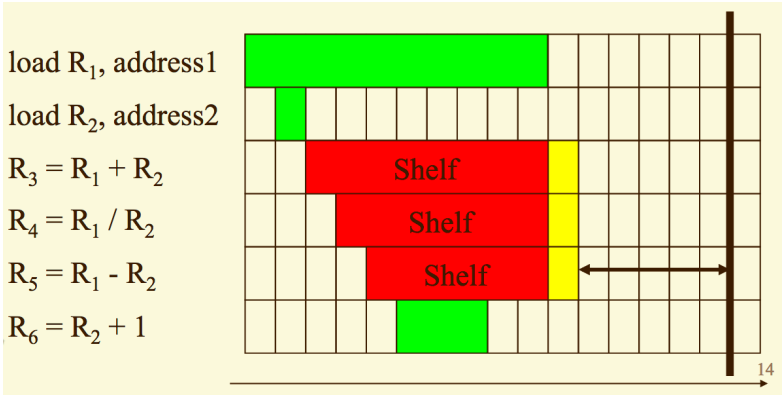


Figure 14.5: Reducing latency with shelving.

14.2 Superscalar Instruction Issues

When evaluating instructions, a variety of issues may arise:

- **False dependencies**: There may be a false dependency on registers that prevents superscalar execution. This can be resolved using register renaming.
- **Unresolved control dependencies**: The next instruction to evaluate may be located after a branch and thus cannot be evaluated until the branch is resolved. This can be resolved by waiting for the result of the branch to be determined, or by speculatively executing the branch and rolling back the result if it is incorrect.
- **Use of shelving**: A shelving buffer can be used to execute instructions that do not have dependencies. Essentially, assume you have an instruction stream like the following:

```
[1]: load R1, address1
[2]: load R2, address2
[3]: R3 = R1 + R2
[4]: R4 = R1 / R2
[5]: R5 = R1 - R2
[6]: R6 = R2 + 1
```

  If the instruction window used to load instructions is three-wide, instructions 1 and 2 will be executed, and instruction 3 will promptly be stalled until 1 and 2 are executed. Assume for some reason that instruction 1, the loading of R1, takes extra time; execution of instructions 3, 4, and 5 will be stalled until R1 is loaded.

  In this case, instructions 3, 4 and 5 are *shelved* until their dependency is resolved, and instruction 6 is executed; this allows for a degree of concurrent processing. This is illustrated in Figure 14.5. Note that the instructions will now complete out of order, and that the machine will still need to enforce data dependencies and sequential consistency.

When instructions are loaded from the instruction buffer in memory, they go through a Decode/Check issue stage which checks them for dependencies and other issues prior to submission to the execution units. This is handled by a *reservation station*, shown in Figure 14.6.

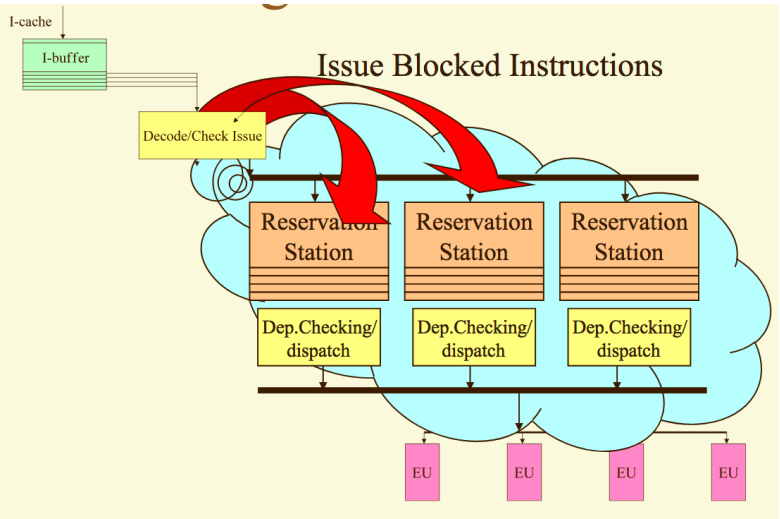


Figure 14.6: Reservation stations for efficient instruction issue.

14.3 Cray-1

The Cray-1 was an early superscalar machine with different address and scalar functional units. The addressing units had 8 A registers and 64 B registers, while the data registers had 8 S registers and 64 T registers. The B and T registers could be used as program-controlled cache. This is depicted in Figure 14.7.

The Cray-1 is a register-register oriented machine; the only instructions that interacted with memory were `load` and `store`. Instructions that manipulate the B and T registers were restricted to memory access and transfer to the A and S files. Information flow was from memory to the A or S registers, or to the B and T registers. Data for the functional units can only come from the S or A register sets. However, it was possible to conduct block transfers between memory and the B and T register files.

Because there are multiple functional units, it is possible to start the next instruction even if the last one is not complete. The only time this process must freeze is when the next instruction depends on the last one. To resolve this, instructions can be issued out of order; an early scheme developed for this is Tomasulo's algorithm, which was originally developed for the IBM 360/91 floating point unit.

14.4 Tomasulo's Algorithm

The algorithm attempts to issue instructions even if they cannot be executed; this will prevent them from holding up later instructions. [1]

Each register is augmented with a Ready bit and a Tag field. Associated with each functional unit is a Reservation Station. Each reservation station can store a pair of operands. Each operand

[1] If you'd like a nice illustration of how this works, please consult the slides. There's about *forty images* worth.

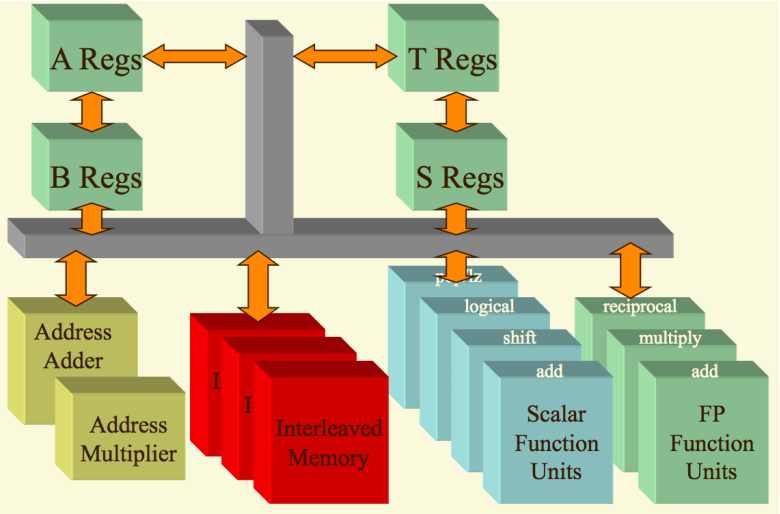


Figure 14.7: Cray-1 architecture.

has its own ready bit and tag bit.

A reservation station can also hold a destination tag. When an instruction is issued, a new destination tag is stored into the destination tag field. These destination tags are assigned from a *tag pool*. Instructions are then issued with little regard to data dependencies.

Instructions are issued if the requested functional unit has an available reservation station, there is a free tag available in the tag pool, and a source register can be guaranteed not to be loaded during this cycle.

When an instruction is issued, the contents of the instruction's source registers, as well as the ready and tag bits of the registers, are copied to the requested functional unit's reservation station. A tag is then allocated from the tag pool and stored in both the reservation station and the tag field of the destination register. The ready bit of the destination register is then cleared.

The crux of Tomasulo's algorithm then applies. If the ready bit in the reservation station is set, the station contains a valid copy of the register's contents. If it is clear, it contains a valid tag; this tag can then be used to find the functional unit responsible for producing the result.

For a functional unit to start execution, all operands must be ready, the unit must not be busy, and it must have access to the bus to return the result to the register file. When an instruction begins execution, it releases the reservation station, reserves the result bus for the appropriate cycle, and attaches the destination tag to the result.

When the result is generated, the tag is compared to all of the tags in all of the reservation stations and all of the registers that have the ready bit clear. The result is stored in all reservation stations that match. The ready bit is then set to indicate that the result is valid, and the tag is returned to the tag pool.

The algorithm supports an unrestricted number of dependent instructions; the tag value is copied to multiple reservation station shelves to accommodate this.

Essentially, this system aims to execute instructions where possible; when instruction execution

is not possible - primarily due to a data dependency - the state will be preserved in a reservation station, which will then be loaded with the data when available. This allows concurrent execution of instructions that do not have data dependencies. In doing so, it enforces flow dependence; this is made possible by the fact that tags are assigned to individual instructions, so that two instructions sharing a register in a false dependency can be separated using different tags.

Tomasulo's algorithm also implicitly implements register renaming; different tags are assigned for different instructions, so that false dependencies are implicitly eliminated.

14.4.1 Implementation

Tomasulo's algorithm can be quite hard to implement. Associative hardware is required to support looking for tags in the register file in one clock cycle; additionally, tag allocation and deallocation hardware is required. It provides a deep flow dependence.

14.5 Thornton's Algorithm

Thornton's algorithm is a simpler issue scheme that removes the tags (and thus allocation and deallocation) and the associative hardware on the register files.

The rules for the reservation station are as follows:

- If the ready bit is set, the operand contains data.
- If the ready bit is clear, the source and destination registers are specified directly.

In Thornton's issue scheme, an instruction is issued if the functional unit has some free reservation slots and the destination register is not busy.

When an instruction is issued, a reservation station for the functional unit is reserved. If a source register is ready, then the data is copied to the reservation station and the ready bit is set. If the register is not ready, then the source register number is placed in the SR field and the associated ready bit is cleared. Afterwards, the ready bit for the destination register must be cleared.

Thornton's algorithm will result in more blocks and reduced performance in comparison to Tomasulo's algorithm, but is considerably simpler to implement. The relative performance is shown in Figure 14.8.

14.6 Other renaming schemes

The compiler can perform static renaming; however, it cannot account [2] for dynamic variation in delays. This means that it will have to use more resources to provide "ideal" performance.

The PowerPC renaming scheme uses a separate register rename file, and associative access to the register file. This means that registers used by executing code are not necessarily the real registers; instead, they are dynamically remapped during execution to maximise performance (Figure 14.9). Operand fetching occurs during renaming.

[2] The exact wording in the slides was "account account for dynamic variation". I'm not entirely sure if that was meant to be a *can* or a *cannot*, but based on the Levenshtein distance I've opted for the latter.

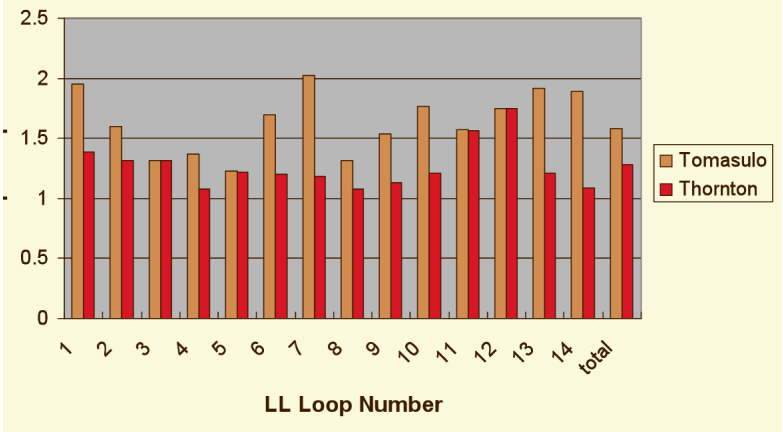


Figure 14.8: Tomasulo algorithm vs Thornton algorithm.

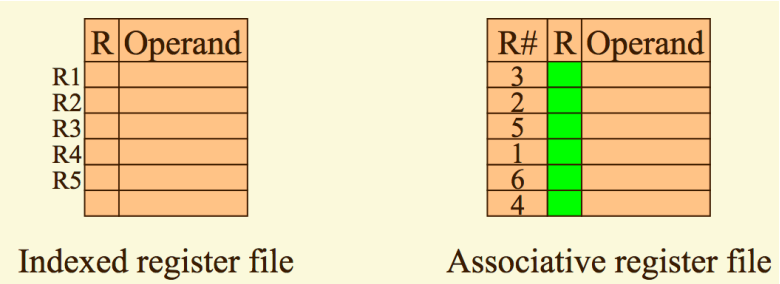


Figure 14.9: Indexed vs associative register file.

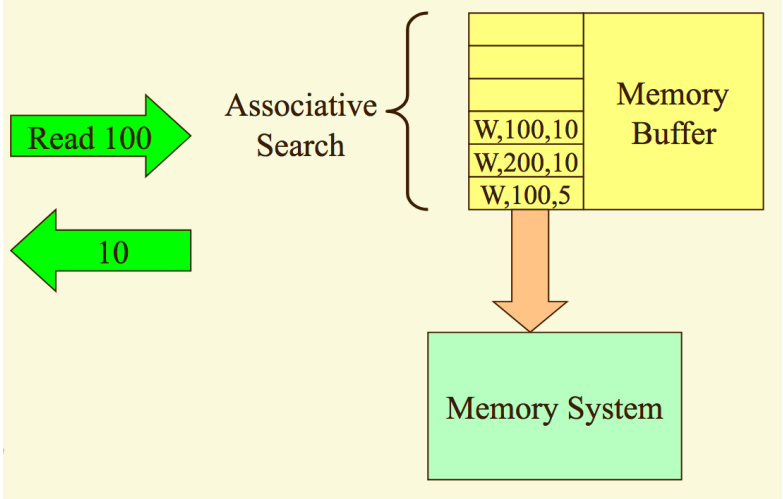


Figure 14.10: Memory dependence checking with an associative queue. $W, 100, 10$ refers to writing 10 to the memory location 100.

However, there may be more than one version of a particular register in use at any given time; this is resolved by using another bit in the register file, the *latest bit*, to indicate whether a particular register is the latest version or not. Fetches from the register file retrieve the most recent (last) write; outstanding instructions take the index of the most recent register to write to.

14.7 Memory data dependence

The schemes discussed to date only work for register-enforced consistency. If memory access is conducted as part of the instruction stream, is it possible to conclude whether data dependence exists?

14.7.1 Static dependence determination

The compiler can determine when data dependence cannot exist. As an example, $A(i) \leftarrow A(i+1)$ must be a different location, and can thus be treated relatively independently.

14.7.2 Dynamic dependence determination

To resolve the problem dynamically, the hardware must keep track of outstanding stores. This is done by maintaining an associative queue of outstanding operations; when a read is requested, the outstanding queue is searched to determine if a pending store is present (Figure 14.10). This is easier to implement on a RISC architecture than a CISC architecture.

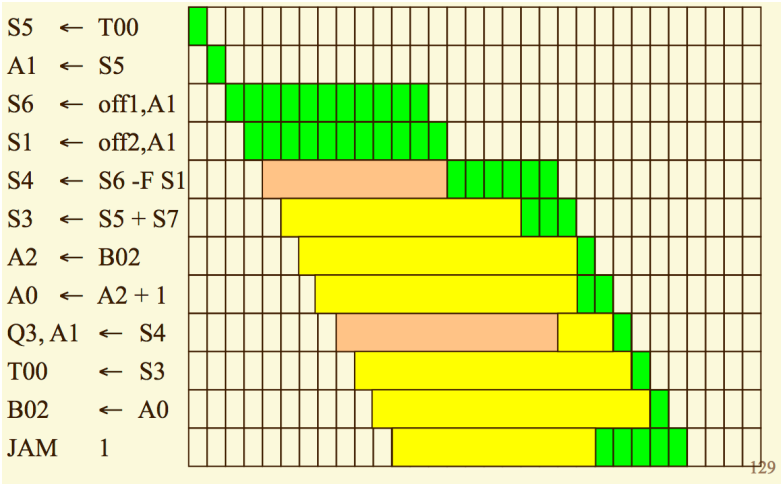


Figure 14.11: In-order completion; out-of-order execution is purposely stalled to ensure that instructions complete in order.

14.8 Preserving sequential consistency

Instructions are issued in order, but may complete out of order. This may cause issues with speculative execution and precise interrupts.

14.8.1 Precise interrupts

On a non-pipelined machine in which instructions are executed in order, a hardware interrupt (i.e. an I/O event or a detected fault) can preserve the current instruction pointer and processor state.

However, if the machine is pipelined, the current instruction pointer is significantly less precise, and preservation of processor state may become problematic.

If instructions finish out of order, then the state of the processor is not sequentially consistent. Some later instructions will be completed, and some earlier instructions may not be completed. This makes it difficult to restart the current instruction flow if recovery is required (e.g. in the case of a page fault). Additionally, debugging information may be incorrect.

If precise interrupts are required, a reorder buffer can be used to enforce in-order completion. Instructions can still be issued even if they are blocked; subsequence instructions can still execute out of order, but they must complete in-order. This is shown in Figure 14.11.

The reorder buffer is implemented using a circular buffer with head and tail pointers (Figure 14.12). The instructions are written to the reorder buffer in strict program order. Each instruction can be in one of three states: issued, in execution and finished. An instruction is only allowed to retire if it is finished and all previous instructions are retired.

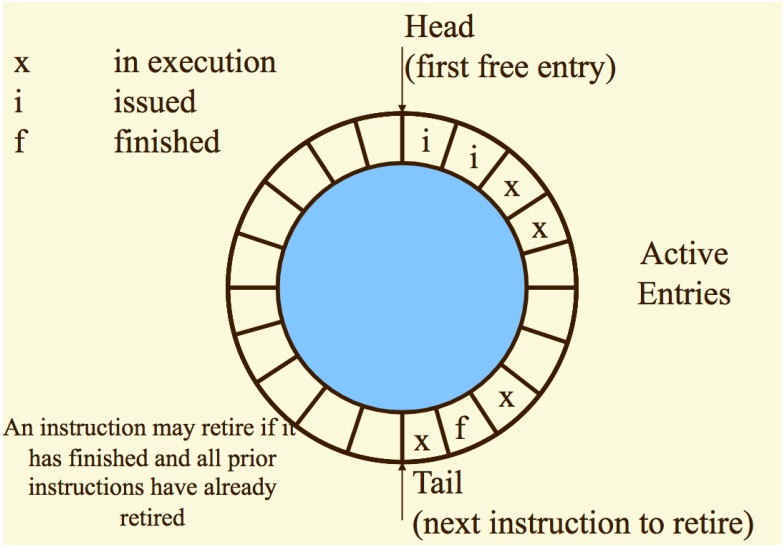


Figure 14.12: Reorder buffer implementation.