Software for MIMD Message Passing Machines

- Old languages with additions for concurrent programming.
  - Parallel versions of C
  - Parallel versions of Fortran

- Routines are added for access to communication links.

- One or a few processes are placed on each processor.

- Mechanism of inter-process communication depends on process location.

- The hardware changes but the languages remain the same.
  - Important for market acceptance.

Excess Parallelism & Virtual Concurrency

- **Sufficient Parallelism**

  With the *Parallel C/Fortran* approach we extract as much concurrency from the problem as we need. We can then write a program for each processor.

- **Excess Parallelism**

  If instead we extract as much concurrency from the problem as possible, we find that we will often have more parallelism than we have processors. We have *excess parallelism*.

- **Virtual Concurrency**

  In order to support excess parallelism, we run multiple processes on a single processor. This multi-tasking we call *virtual concurrency* because the time-sliced processes must appear to run concurrently.

The Benefits of Excess Parallelism

- ## Masking of Message Latency

  In MIMD message passing systems the latency of message passing is often a limiting factor.

  In order to mask this latency such that it doesn't effect the execution time of the program, we can *deschedule* a process which is waiting for communication such that it no longer gets any CPU time.

  The greater the level of *excess parallelism* the greater the masking effect.

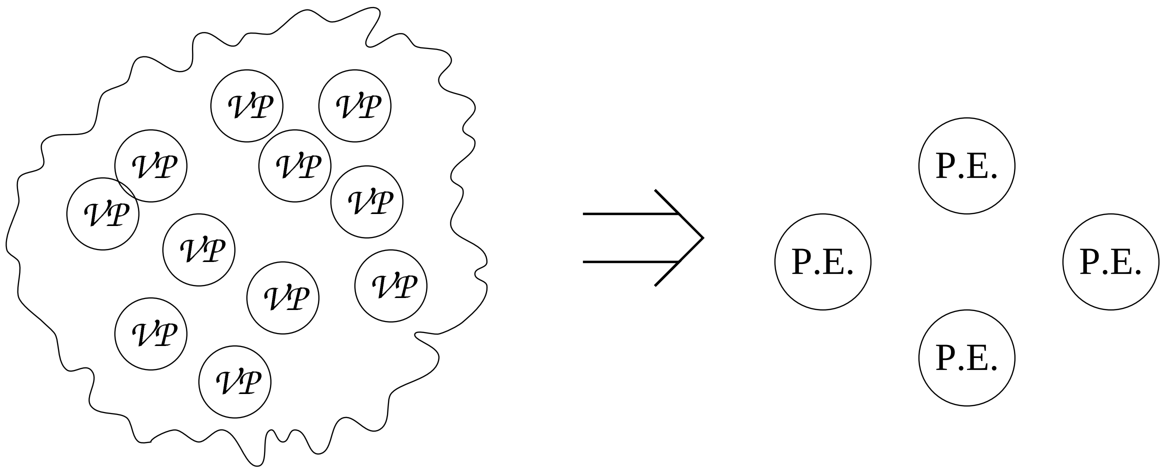
- ## Abstraction

  The number of processors is no longer important. The real concurrency will expand to fit any number of processors until there is only one process on each processor.

  Programs are more portable and easier to write.

Virtual Processors

We can consider this programming style as programming for a set of *virtual processors*[1].



We program as if for an arbitrarily large number of *virtual processors*, one per concurrent process, and then map the *virtual processors* onto the available real processors.

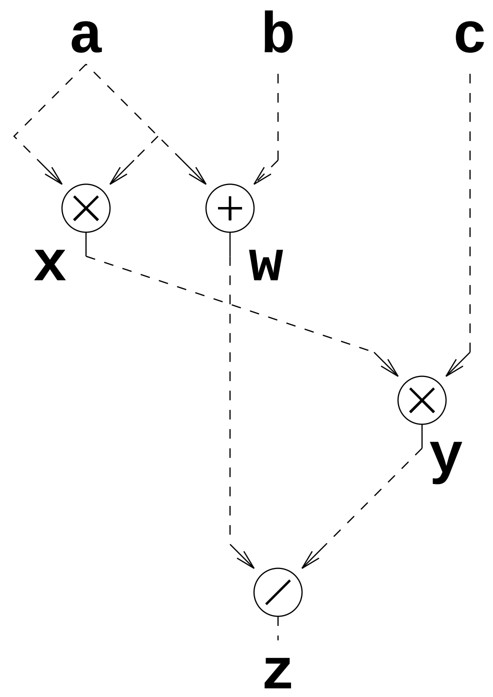
[1] c.f. virtual memory

Programming with MIMD

Occam - A language for MIMD message passing systems.

Occam's SEQ and PAR constructs provide a framework for programming with excess concurrency – it is as simple to describe a parallel process as a sequential one.

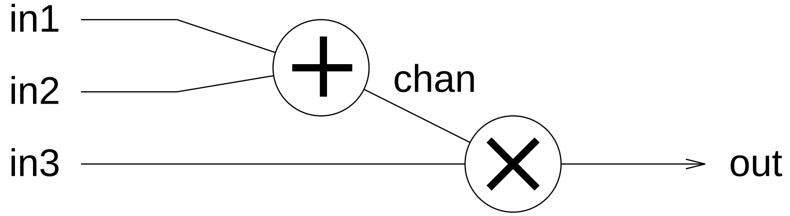
```
SEQ
  PAR
    w := a+b
    SEQ
      x := a*a
      y := x*c
  z := w/y
```



Occam - A language for MIMD message passing systems.

Occam also supports message passing primitives operating over channels. Occam channels provide unbuffered and synchronized communication of values between concurrent processes:

```
PAR
  SEQ
    in1 ? A
    in2 ? B
    chan ! A + B
  SEQ
    chan ? X
    in3 ? Y
    out ! X * Y
```



Problems of programming with MIMD

Deadlock

In the following code segments the parallel processes wish to swap data via communication channels.

The both versions of following occam code illustrate deadlock:

```
PAR
  SEQ
    chan1 ? A
    chan2 ! B
  SEQ
    chan2 ? X
    chan1 ! Y
```

```
PAR
  SEQ
    chan2 ! B
    chan1 ? A
  SEQ
    chan1 ! Y
    chan2 ? X
```

The communication cannot take place until both processes are ready to proceed.

Problems of programming with MIMD

Deadlock

The state in which two or more processes are deferred indefinitely because each is awaiting another process to make progress, and no process is able to make progress.

Deadlock may be avoided with careful programming:

```
PAR
  SEQ
    chan1 ? A
    chan2 ! B
  SEQ
    chan1 ! Y
    chan2 ? X
```

Deadlock

- **Message passing MIMD systems**
  Although we have illustrated the problem in occam all message passing MIMD systems are subject to deadlock, although sometimes the onset of deadlock is unpredictable[2].

- **Shared Memory MIMD systems**
  Poor programming with semaphores will also result in deadlock.

- **Networks**
  A poorly designed network may deadlock. This can usually be avoided by careful control of buffers and routing strategies[3].

[2] e.g. deadlock only happens when a buffer becomes full.
[3] e.g. dimension order routing in grids and hypercubes.