Parallelism

Let us consider the programming of a parallel machine, this will give us some insight into what we want of our parallel machine.

For each problem to be solved, the following must be performed:

- Identify Parallelism

  We must identify operations which can be done in parallel.

- Express Parallelism

  We must write (or re-write) code to indicate the parallelism present.

- Exploit Parallelism

  We must be able to distribute these parallel operations amongst our *processing elements*.

Identify Parallelism

Two computations may be done in parallel provided that the result from one is not required (directly or indirectly) for the completion of the other.[1].

Take:

```
x := a + b;
y := b + c;
```

These computations are independent.
We can carry out operations in parallel.

Take:

```
x := a + b;
y := x + c;
```

`y` is dependent on the calculated value of `x` (we have *data flow* between them).
We *can't* carry out operations in parallel.

[1] i.e. there is no flow of data from one to the other

Express Parallelism

We must use a language which can cope with parallelism.

OCCAM provides us with two basic structures to distinguish between sequential and parallel operations.

```
SEQ
  A
  B
  C
```

Causes A, B & C to be evaluated in strict SEQuence.

```
PAR
  A
  B
  C
```

States that A, B & C *may* be evaluated in PARallel.

Express Parallelism

Thus

```
x := a + b;
y := b + c;
```

can be expressed as

```
PAR
  x := a + b
  y := b + c
```

Whereas

```
x := a + b;
y := x + c;
```

must be expressed as

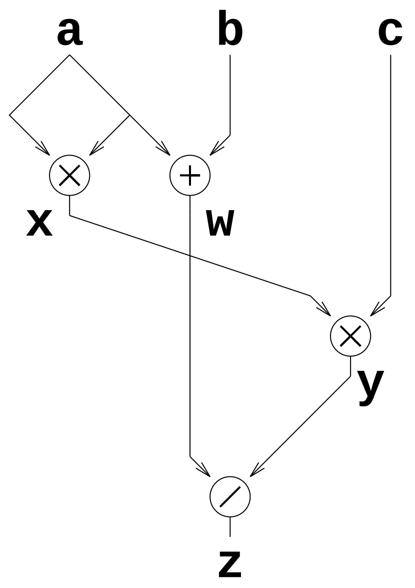
```
SEQ
  x := a + b
  y := x + c
```

Express Parallelism

For a more formal approach we can use data flow analysis

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

Sequential Form
(Pseudo Pascal)



Data Flow Graph

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

Parallel Form
(Pseudo Occam)

Note that the scope of each PAR/SEQ is indicated by indentation.

Parallel Algorithms

Given a single problem, there are frequently several algorithms for its solution.

- Usually one algorithm dominates all others due to its suitability for computing.
- What if this algorithm exhibits no potential for exploiting parallelism?
- We may be able to find an alternative algorithm specifically for parallel machines.

Thus we have specialised sequential algorithms which run efficiently on sequential machines and inefficiently on parallel machines, and specialised parallel algorithms which run efficiently on parallel machines and inefficiently on sequential machines.

Sequential Sum

The sequential sum problem in an excellent example where the 'natural' algorithm is unsuitable for a parallel implementation.

We start with a vector A[1..n] and wish to obtain a vector of 'sums' SUM[1..n],

such that

$$SUM[i] = A[i] + SUM[i-1]$$

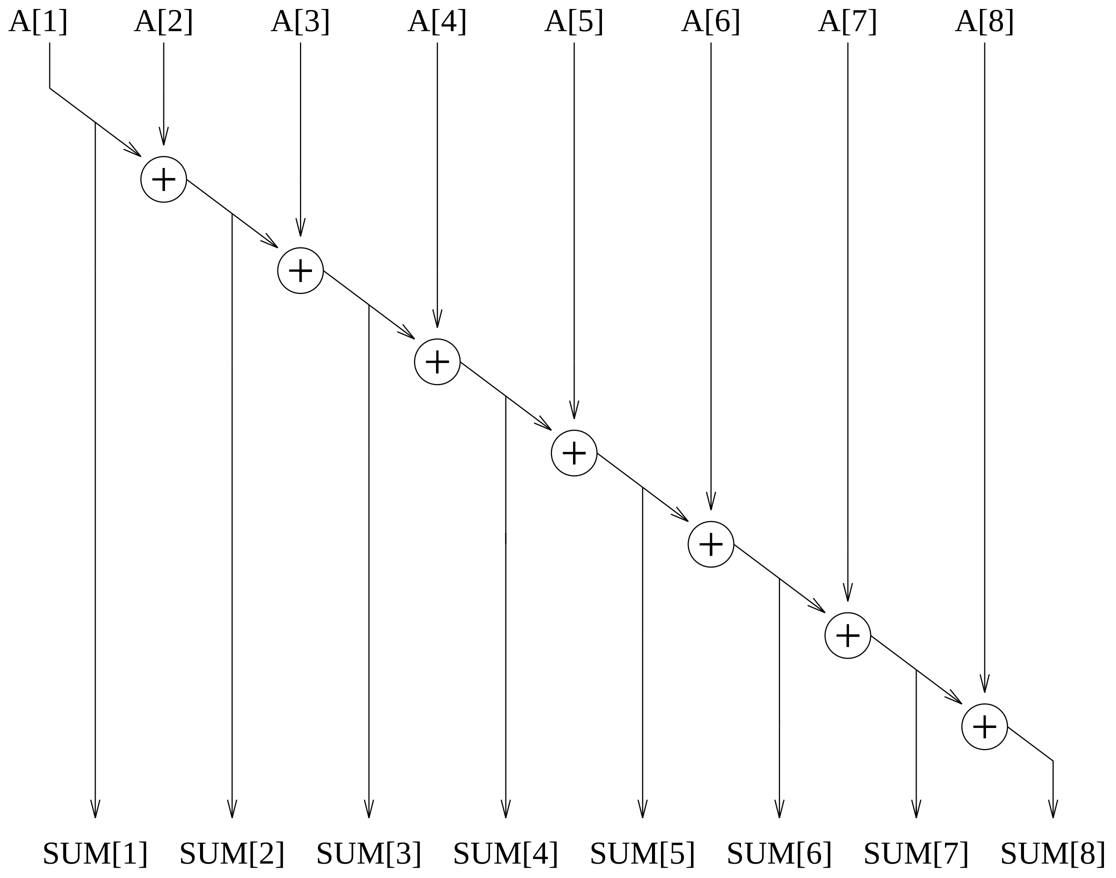
where

$$SUM[0] = 0$$

The problem is defined as a recursion, each result depends upon the previous one. It appears that we must produce the results in strict sequence with no parallelism.

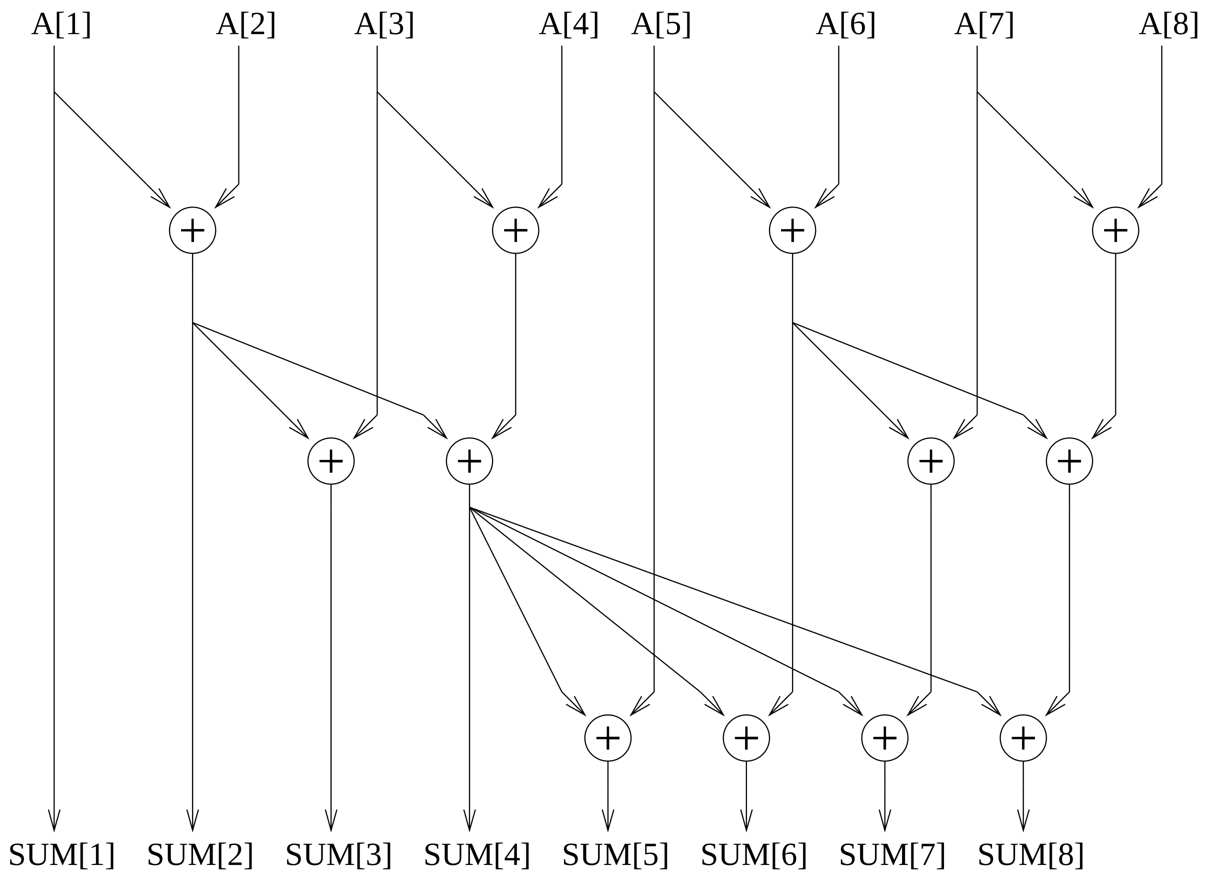
Sequential Sum - Sequential Algorithm

The sequential algorithm is straightforward and takes $n - 1$ additions all performed in strict sequence.



Sequential Sum - Parallel Algorithm

The parallel algorithm is more complex taking $log_2(n) * n/2$ additions to perform the same task.



Parallel Speedup

We wish to compare our two algorithms.

- How much faster is our parallel algorithm?

For comparison purposes we invent a *Paracomputer* on which to run the software.

The Paracomputer is an ideal parallel computer with an arbitrarily large number of processors and no overheads for communication or co-ordination.

The concept of such an ideal computer allows us to compare algorithms simply, without considering particular machines.

Algorithm Performance

For sequential algorithm

We have $n - 1$ additions carried out in sequence.
hence

$$T_{seq} = (n - 1)T_{add}$$

where

$$T_{add} = time\ for\ a\ single\ addition$$

For parallel algorithm

We have $log_2(n)$ additions in the critical path
hence

$$T_{par} = (log_2(n))T_{add}$$

Parallel Speedup

$$\frac{T_{seq}}{T_{par}} = \frac{n - 1}{log_2(n)}$$

Algorithm Performance

Algorithmic Parallelism

There is another measure of algorithmic performance.

- How many processing elements do we need for our algorithm?

Clearly our sequential algorithm only needs 1 processing element.
We say it has

$$Algorithmic\ Parallelism = 1$$

Our parallel algorithm needs $n/2$ processing elements.
We say it has

$$Algorithmic\ Parallelism = n/2$$

Algorithm Performance

For any real machine

If

$$Algorithmic\ Parallelism < Number\ Of\ Processing\ Elements$$

Then we are unable to make use of all of our processing elements.

If

$$Algorithmic\ Parallelism > Number\ Of\ Processing\ Elements$$

Then we must expect a lower parallel speedup.

Note that the parallel speedup for any particular machine is likely to be less than predicted by the Paracomputer due to the overheads of communication and co-ordination.

Identifying Parallelism

Process Parallelism

So far we have looked at *process parallelism*.

Process parallelism exists where any number of related or unrelated processes can be performed in parallel.

Data Parallelism

Data parallelism is a subset of process parallelism.

Data parallelism exists where we can manipulate data structures in parallel.

Express Parallelism

If we are adding two arrays together, *element by element*, we find that there is no data flow between the different additions.[2] Hence we can perform the operations in parallel.

Taking a sequential algorithm: (Pseudo Pascal)

```
for i = 0 to 99
begin
  C[i] := A[i] + B[i];
end
```

we find that it exhibits data parallelism.

We can re-express it in data parallel form: (Pseudo Fortran 90)

```
C(0:99) = A(0:99) + B(0:99)
```

or even

```
C = A + B
```

Data parallelism in algorithms is easy to identify and easy to express.

[2]This is also true for any *elemental* operation

Exploiting Parallelism

Having identified and classified the parallelism within your problem you must exploit it on a parallel machine.

Due to the large number of scientific and engineering problems which exhibit data parallelism, many parallel machines have been built specifically to exploit this market.
These are *Array Processors* and *Vector Processors*.

In the early years of parallel computing, the most successful machines have been Supercomutpers based on *Pipeline Vector Processors*.