How is Parallelism Found?

There are two approaches:

- We can use a "smart" compiler that is able to find parallelism in existing programs written in standard serial programming languages.
- We can add features to an existing programming language that allows a programmer to show where parallel evaluation is desired.

Concurrentization

Concurrentization (often called parallelization) is process of automatically finding potential concurrent execution in a serial program.

Automatically finding current execution is complicated by a number of factors:

- Data Dependence

  Not all expressions are independent. We may need to delay evaluation of an operator or subprogram until its operands are available.

  Thus in

  `(+ (* x y) (* y z))`

  we can't start the addition until both multiplications are done.

- Control Dependence

  Not all expressions need be (or should be) evaluated.

  In

```
(if (= a 0)
          0
     (/  b a))
```

  we don't want to do the division until we know **a** ≠ **0**.

- Side Effects

  If one expression can write a value that another expression might read, we probably will need to *serialize* their execution.

  Consider

```
(define rand!
 (let ((seed 99))
   (lambda ()
    (set! seed
      (mod (* seed 1001) 101101))
    seed
)) )
```


Now in

`(+ (f (rand!)) (g (rand!)))`

we can't evaluate `(f (rand!))` and `(g (rand!))` in parallel, because of the side effect of `set!` in `rand!`. In fact if we did, `f` and `g` might see *exactly* the same "random" number! (Why?)

- Granularity

  Evaluating an expression concurrently has an overhead (to setup a concurrent computation). Evaluating very simple expressions (like `(car x)` or `(+ x 1)`) in parallel isn't worth the overhead cost.

  Estimating where the "break even" threshold is may be tricky.

Utility of Concurrentization

Concurrentization has been most successful in engineering and scientific programs that are very regular in structure, evaluating large multidimensional arrays in simple nested loops. Many very complex simulations (weather, fluid dynamics, astrophysics) are run on multiprocessors after extensive concurrentization.

Concurrentization has been far less successful on non-scientific programs that don't use large arrays manipulated in nested for loops. A compiler, for example, is difficult to run (in parallel) on a multiprocessor.

Concurrentization within Processors

Concurrentization is used extensively within many modern uniprocessors. Pentium and PowerPC processors routinely execute several instructions in parallel if they are independent (e.g., read and write distinct registers). This are *superscalar* processors.

These processors also routinely *speculate* on execution paths, "guessing" that a branch will (or won't) be taken even before the branch is executed! This allows for more concurrent execution than if strictly "in order" execution is done. These processors are called "out of order" processors.

Adding Parallel Features to Programming Languages.

It is common to take an existing serial programming language and add features that support concurrent or parallel execution. For example versions for Fortran (like HPF—High Performance Fortran) add a parallel do loop that executes individual iterations in parallel.

Java supports threads, which may be executed in parallel. Synchronization and mutual exclusion are provided to avoid unintended interactions.

Multilisp

Multilisp is a version of Scheme augmented with three parallel evaluation mechanisms:

- Pcall
  Arguments to a call are evaluated in parallel.
- Future
  Evaluation of an expression starts immediately. Rather than waiting for completion of the computation, a "future" is returned. This future will eventually transform itself into the result value (when the computation completes)
- Delay
  Evaluation is delayed until the result value is really needed.

The Pcall Mechanism

Pcall is an extension to Scheme's function call mechanism that causes the function and its arguments to be all computed in parallel.

Thus

```
(pcall F X Y Z)
```

causes **F**, **X**, **Y** and **Z** to all be evaluated in parallel. When all evaluations are done, **F** is called with **X**, **Y** and **Z** as its parameters (just as in ordinary Scheme).

Compare

```
(+ (* X Y) (* Y Z))
```

with

```
(pcall + (* X Y) (* Y Z))
```


It may not look like **pcall** can give you that much parallel execution, but in the context of recursive definitions, the effect can be dramatic.

Consider **treemap**, a version of **map** that operates on binary trees (S-expressions).

```
(define (treemap fct tree)
 (if (pair? tree)
  (pcall cons
    (treemap fct (car tree))
    (treemap fct (cdr tree))
  )
  (fct tree)
))
```


Look at the execution of **treemap** on the tree

```
  (((1 . 2) . (3 . 4)) .
   ((5 . 6) . (7 . 8)))
```

We start with one call that uses the whole tree. This splits into two parallel calls, one operating on

```
((1 . 2) . (3 . 4))
```

and the other operating on

```
((5 . 6) . (7 . 8))
```

Each of these calls splits into 2 calls, and finally we have 8 independent calls, each operating on the values **1** to **8**.

Futures

Evaluation of an expression as a future is the most interesting feature of Multilisp.

The call

```
(future expr)
```

begins the evaluation of **expr**. But rather than waiting for **expr**'s evaluation to complete, the call to **future** returns *immediately* with a new kind of data object—a future. This future is actually an "IOU." When you try to use the value of the future, the computation of **expr** may or may not be completed. If it is, you see the value computed instead of the future—it automatically transforms itself. Thus evaluation of **expr** appears instantaneous.

If the computation of **expr** is not yet completed, you are forced to wait until computation is completed. Then you may use the value and resume execution.

But this is exactly what ordinary evaluation does anyway—you begin evaluation of **expr** and wait until evaluation completes and returns a value to you!

To see the usefulness of futures, consider the usual definition of Scheme's map function:

```
(define (map f L)
  (if (null? L)
     ()
     (cons (f (car L))
           (map f (cdr L)))
  )
)
```


If we have a call

```
(map slow-function long-list)
```

where **slow-function** executes slowly and **long-list** is a large data structure, we can expect to wait quite a while for computation of the result list to complete.

Now consider **fastmap**, a version of **map** that uses futures:

```
(define (fastmap f L)
  (if (null? L)
     ()
     (cons
       (future (f (car L)))
       (fastmap f (cdr L))
     )
  )
)
```

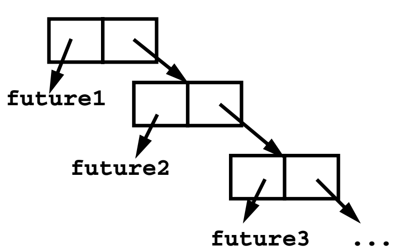
Now look at the call

```
(fastmap slow-function long-list)
```

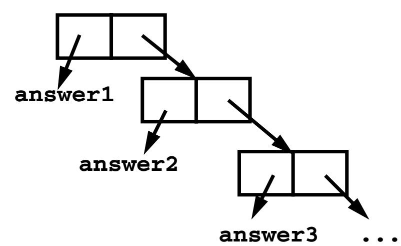

We will exploit a useful aspect of futures—they can be cons'ed together *without delay*, even if the computation isn't completed yet.

Why? Well a **cons** just stores a pair of pointers, and it really doesn't matter what the pointers reference (a future or an actual result value).

The call to **fastmap** can actually return before *any* of the call to **slow-function** have completed:



Eventually all the futures automatically transform themselves into data values:



Note that **pcall** can be implemented using futures.

That is, instead of

```
(pcall F X Y Z)
```

we can use

```
((future F)
   (future X) (future Y) (future Z))
```

In fact the latter version is actually more parallel—execution of **F** can begin even if all the parameters aren't completely evaluated.