Linguagens Funcionais e Supercomputadores

Silvio Lemos Meira

Departamento de Informática
Universidade Federal de Pernambuco
Cidade Universitária
50.739, Recife-PE, Brasil.

Computing Laboratory
University of Kent
Canterbury, UK

Sumário

Neste artigo nós consideramos alguns dos problemas encontrados na programação de máquinas não estritamente von Neumann usando linguagens tipicamente sequenciais, e apresentamos uma alternativa, ou seja, o uso de linguagens funcionais de alta ordem. Estas linguagens não têm conceito de estado ou processamento sequencial, e é interessante ver como o tipo de cálculo matemático normalmente descrito como *number crunching* pode ser expresso neste paradigma.

Abstract

We consider some of the common problems that have to be circumverted while trying to program vector and array processors, i.e., modified von Neumann machines, using languages that do not normally cope with parallelism. We then present another alternative, higher-order functional programming languages, which have no state (or side-effects) and in which the *parallelism* is inherent. We describe a few small examples of number crunching like calculations using this paradigm.

1 Introdução

Este artigo versa sobre o uso de linguagens funcionais como um dos veículos de programação de máquinas paralelas, em oposição ao uso corrente de linguagens da família de Fortran. De fato, o artigo é visto no sentido oposto da frase anterior, isto é, *máquinas paralelas como um mecanismo de implementação de linguagens funcionais.*

O nosso uso de *paralelismo* está restrito ao conceito de mais de um processador trabalhando simultaneamente para levar uma computação a termo, computação esta descrita em uma linguagem qualquer. Não há aqui nenhuma consideração sobre concorrência, explicitamente. Não há necessidade de se descrever *concorrência* em linguagens funcionais, para computação numérica, pois o paralelismo é intrínseco. Mais ainda, o paralelismo não está restrito a vetores e outras estruturas homogêneas.

Uma característica interessante das linguagens funcionais é que o mesmo *script* fonte pode ser executado, independentemente, em uni- ou multiprocessadores, sem qualquer modificação, e tomando proveito de um ambiente de multiprocessadores.

A seguir, analisamos algumas questões relativas à programação pseudo-paralela de máquinas do mesmo tipo.

2 Fortran, seus pares e hardware

A linguagem Fortran, largamente utilizada em computação científica, é uma bisavó das linguagens imperativas consideradas modernas, como Modula-2, C e Ada. Fortran mais se assemelha a assembler, em sua forma original, do que a uma linguagem de programação de alto nível. Há uma razão elementar para esta semelhança, que é a aparente necessidade de permanecer *perto* do nível de máquina (assembler), para não perder eficiência. Em outros casos, estar perto do nível de máquina pode não ter o mesmo significado, pois a máquina pode ter sido projetada *para* a linguagem.

O método de projeto de linguagens de programação, à época em que Fortran foi concebido –e em alguns casos ainda hoje, ver discussão sobre [PeZ] e [Pet] abaixo– era o de *fazer* a linguagem *para* a máquina. Ou seja, se os projetistas de hardware conseguem implementar uma máquina de 13^{15} processadores, dispostos em um hipericosaedro irregular, é *dever* dos projetistas de linguagem encontrar um meio razoável de programar tal aberração. E levando em conta o comprimento dos fios que ligam bancos de memória e processadores, se for o caso...

Talvez para complicar ainda mais o retrato, muitos dos super-processadores implementados recentemente foram projetados tendo Fortran como objetivo, ou seja, não passam de variações sobre um tema de von Neumann. Exemplos destas máquinas são os Cray[Rus] e Cyber200(205)[CDC], que são processadores *vector* e o DAP[Red], do tipo *array*. A diferença básica entre os dois conceitos é que, no *vector*, uma operação em uma série de operandos é sobreposta em um *pipeline*, enquanto que em um *array*, um número de elementos de processamento, independentes mas sob o controle de um único fluxo de instrução, executa as operações em paralelo.

Este artigo não discute (mas menciona, em várias partes) máquinas como a *Connection Machine*[Hil] ou *Computing Surface*[MCS], que pertencem a famílias de super-hardware diferentes das citadas acima.

A maioria dos processadores *array* e *vector* foi otimizada, como o próprio nome indica, para realizar computações sobre matrizes, com performance na região de dezenas ou mesmo centenas de MFLOPs. Isto dentro de restrições de estilo de programação que normalmente trariam muitas rugas a um programador preocupado com a clareza, expressividade e correção do seu programa. Isto é o que discutiremos a seguir.

3 Supercomputadores e Linguagens Imperativas

Aqui vamos proceder a uma análise crítica e comparativa de dois trabalhos recentes, de Petersen[Pet] e Perrot/Z.-Aliabadi[PeZ], nos quais se discute o problema dos supercomputadores e suas linguagens, do ponto de vista imperativo. Os termos *imperativo* e *funcional*, neste trabalho, se referem aos paradigmas de programação (e concepção de programas) representados por linguagens baseadas em estado e comandos (procedimentais) e funções e aplicações (aplicativas).

Os dois artigos citados acima são recentes, significativos e largamente divulgados, razões que nos levaram a escolher ambos como objeto de discussão. Não está em jogo, obviamente, a qualidade dos dois artigos.

3.1 Linguagens "para" Supercomputadores

Uma evidência clara do critério de projeto de linguagens que é atualmente usado para supercomputadores é dada em [PeZ] (pp.8)

> ...there are two general approaches that can be employed for designing supercomputer languages:
>
> **(1)** An existing sequential language may be adapted;
>
> **(2)** A high-level language, based on the hardware features of the supercomputer, may be formulated.

Em relação a (1), os autores apontam que a escolha pode ter as seguintes desvantagens ([PeZ], pp.8):

> **(α)** The language mirrors the underlying seriality of the computers for which it was designed;
>
> **(β)** Any changes introduced in the chosen language can take on the appearance of patches to its basic design, which can lead to inconsistencies;
>
> **(γ)** The user, because of familiarity with the sequential language, constructs algorithms based on the manipulation of scalar quantities.

Em relação a (α), pode-se acrescentar que não só a linguagem é moldada na serialidade, mas certas características básicas (modos de endereçamento, tamanho de palavra, etc.) do hardware que a suporta.

Mais adiante (pp.9), os autores explicam que, de fato, a escolha de uma linguagem para programação paralela recaiu sobre Fortran padrão porque

> **(a)** Using standard Fortran promotes program portability and its attendant benefits;
>
> **(b)** Software construction is independent of the target machine hardware parallelism features.

deixando para o compilador a detecção das partes do programa que são *vetorizáveis*, e esclarecem que

> ...the task of a compiler intending to generate optimal code for a supercomputer (is) very complex.

Daí, aparece o grupo de linguagens chamado Fortran *paralelo*, que incorpora

> ...adaptations that represent the parallel processing features of a particular machine, allowing relevant parallel machine-related details to be conveyed to a compiler. These languages are by their very nature non-portable but *should*[1] facilitate the generation of efficient code.

Vamos primeiro chamar atenção para o fato de que a escolha de Fortran & Cia. para programar supercomputadores é mais que natural. A parte da comunidade científica interessada em *number-crunching* programa em Fortran, e há bilhões de dólares de software escrito em Fortran. No entanto, em qualquer das citações acima, nota-se que este mesmo software não pode, no seu estado sequencial, fazer uso das características *paralelas* dos processadores *vector* e *array*. No caso do Cray, por exemplo CFT (Cray Fortran) só vetoriza loops do tipo `DO`, e mesmo assim um destes não será vetorizável se ele contém ([PeZ], pp.11)

- input/output statement;
- `CALL` statement and subprograms named in an `EXTERNAL` statement;
- `GO TO` statement;
- `IF` statement.

Após desfiar um rosário de problemas similares relativos ao Cyber200(205)[2] os autores dizem que

> ...only a few features (of vector Fortran) are common to both systems (Cray and Cyber)... the number of common standard subroutines between the two systems is small...

o que não é nada surpreendente, dado que em cada sistema as características paralelas da linguagem derivam diretamente do hardware no qual estão implementadas, e a conclusão é que

> (1) For programs written in standard Fortran involving both systems, *portability can be severely affected if vectorization is to be exploited*;
>
> (2) *Because the onus is placed on the programmer* to structure and write programs in a restrictive manner for vectorization to take place, *it may not allow the most natural or direct expression of an algorithm* for the solution of a problem; *this would destroy* a would-be optimal program structure and *possibly introduce errors*.

E as coisas são ainda mais complexas nos processadores to tipo *array*, por motivos óbvios. A partir destas considerações, os autores propõem uma linguagem paralela baseada em Pascal, Actus[Per], na qual as operações paralelas devem ser explicitadas pelo programador. A Fig. 1 mostra um trecho de programa em Actus.

Sem discutir os problemas semânticos que são causados pelas construções acima (para as quais os autores não dão a semântica), o ponto de contenção aqui é que esta

[1] Nossa ênfase.

[2] O qual foi descrito ao autor, por um experiente programador do ACMC da University of Georgia at Athens, USA, como *actively user hostile*...

```
parconst
  sequence = 60:70;
  {sequence representa o conjunto 60,61,...70}
var
  v : array[p:q] of integer;
  {v e um array de integer,
   que pode ser indexado por uma "var" do tipo "index"}
index
  range = 10:20;
  {range e um indice, de "range" 10,11,...20}
begin
  ...
  v[range] := sequence;
  {faz v[10]=60, v[11]=61, etc...}
end.
```

Figura 1: Programa em Actus.

capacidade é justamente um complicador a mais, do ponto de vista de utilidade e usabilidade da linguagem.

Certamente deve ser mais fácil programar o DAP em Actus do que em Fortran, mas já era mais fácil programar em *standard* Pascal do que Fortran. A pergunta que deve ser respondida talvez deva ser: *é apropriado programar uma máquina paralela com uma linguagem imperativa paralela, derivada de uma linguagem sequencial?*

Parte da resposta é que mesmo com a linguagem tendo meios de explicitar o paralelismo que o usuário quer, todos os problemas devidos ao estado, atribuições, efeitos colaterais, conceito limitado de funções e outros mais continuam presentes, e cada vez tendo maior importância, devido às características do hardware. E ainda temos que ser convencidos que vai ser realmente mais simples usar o paralelismo explicitamente, pelo menos em linguagens como Fortran e Pascal.

Em linguagens declarativas, ao contrário, não existe *ordem* de execução, atribuição ou efeitos colaterais de qualquer tipo, e a tarefa do ompilador para identificar o paralelismo é muito simplificada, pois independe de restrições como as vistas acima para as linguagens imperativas. Esta tarefa é ainda mais simples nas linguagens (como Miranda[3][Tur]) onde o usuário pode definir, algebraica e precisamente, os tipos de dados que vai usar e suas características de implementação. Isto não significa que o compilador (ou a linguagem) é mágico e que tudo vai ser resolvido em paralelo. Longe disso. Mas os problemas são mais simples e estão razoavelmente mais bem definidos.

[3]Miranda is a trademark of Research Software Ltd.

```
      DO 1 I = 1,N
C (1)
        Y1(I,I) = X1(I1 + I*(I2*I3))
C (2)
        Y2(I4) = X2(IFCN(X) + I)
        I4 = I4 + 2*I5
C (3)
        Y3(I6 + I) = ATAN(X3(I))
        Y3(I7 + I) = FCN(X3(I))
C (4)
        Y4(I) = X4(I + I*I9)
        Y5(I) = Y4(I8 + I)
C (5)
        I10 = INDEX(I)
        Y6 = FLOAT(I)
C
   1  CONTINUE
```

Figura 2: Loops não vetorizáveis.

Para encerrar a discussão sobre Fortran, mostramos na Fig. 2 um exemplo (tirado de [Pet], pp.1012), no qual se mostra que mesmo depois de obedecer a todas as restrições já citadas anteriormente, as razões pelas quais o código *não vetoriza*:

1. The spacing computations in Y1 and X1 *are too complicated for CFT...*
2. The increment 2*I5 is not a sum of simple invariants... and the subscript expression for X2 contains a function reference..
3. Reference to array Y3 represents a subtle dependency case...
4. References Y4 seem to CFT to be a dependency... Y5 requires an element of Y4, which is computed at the same step...
5. Array Y6 is referenced in a non-linear way... FLOAT(I) is a direct recalculation of the DO-variable...

Pode-se concluir, com alguma segurança, que Fortran não é bem o que se deseja para escrever programas que vão executar em hardware paralelo. A pergunta seguinte, obviamente, é se precisamos de uma linguagem imperativa *paralela*, isto é, onde o paralelismo das operações deve ser explicitado, para obter o máximo do hardware que dispomos hoje em dia e no futuro.

Primeiro vamos supor que sim. Se este for o caso, o que nos espera do ponto de vista de engenharia de software não é um futuro muito alvissareiro, pois já hoje existem máquinas com dezenas de milhares de processadores, ligados de formas não triviais, e com arquiteturas bem peculiares. Alguns exemplos são a Connection Machine, a Computing Surface e o BBN Butterfly. À medida que estas máquinas forem se tornando, do ponto de vista de arquitetura, cada vez mais complexas, será cada vez maior o trabalho

de programá-las usando uma linguagem que está diretamente ligada às características do hardware[4] Os programas que funcionarem certamente serão bem rápidos, mas levarão um longo tempo (de programadores especializados na arquitetura) para desenvolver, e certamente serão muito difíceis de entender e modificar, sem falar sobre *provar* que algum deles realmente faz o que diz fazer.

Agora vamos supor que não. Que tipo de linguagem, então seria candidato ao posto? No resto deste trabalho nós tentamos, sucintamente, expor o caso a favor das linguagens puramente funcionais. Devido à exiguidade de espaço, no entanto, a discussão será breve, e um outro artigo [Mei], em preparo, deve estendê-la um pouco mais.

4 Programação Funcional e Paralelismo

Linguagens funcionais, como Miranda, são despidas de conceitos como memória, atribuição, comandos, efeitos colaterais, e estado de programa. A programação é baseada em dois conceitos, o de definição de funções e a sua consequente aplicação a parâmetros para produzir resultados. Miranda é uma destas linguagens, onde as funções são definidas através de equações *guardadas*, e o mecanismo computacional que sustenta a linguagem é *fully lazy* (semanticamente, *non-strict*). Isto significa que para todos termos (aplicações de funções a parâmetros) para os quais existe uma forma normal (em termos simples, que não é um *loop*, *erro* ou *indefinido*), esta pode ser computada efetivamente. Mais ainda, o mecanismo computacional garante que estes termos são computados no máximo uma vez.

Um *script*, ou programa, tem a forma

```
fat 0 = 1
fat n = n * fat (n-1)
```

onde a guarda, no caso é o *pattern matching* `fat 0`. Esta, é claro, é a definição da função fatorial, que tem tipo

```
fat :: num -> num
```

O tipo da função não precisa (mas pode) ser declarado pelo programador, pois o compilador infere os tipos das definições. Não há nada especial nesta definição, e ela é suficientemente parecida com a sua definição matemática para exemplificar a clareza na qual podemos expressar definições na linguagem. . . aplicação `f 5` retorna o valor 120.

Agora vejamos algo mais substancial, como

```
map f []     = []
map f (a:x) = f a : map f x
```

que é, como se vê, uma função de dois argumentos. `[]` é a lista vazia, e define a primeira equação como sendo a que trata do caso em que o segundo parâmetro é uma lista vazia. `(a:x)` é uma lista com *head* `a` e *tail* `x`, e `:` é *cons*, o construtor de listas. Nesta equação diz-se que `f` deve ser aplicado à cabeça da lista, e que o resultado deve ser *cons*ed a `map f x`. O tipo da função acima, deduzido pelo compilador, é

[4] No caso da Computing Surface, o hardware não é von Neumann, mas um modelo de execução de processos comunicantes (CSP, occam), e ainda não há um compilador Fortran.

```
map :: (* -> **) -> [*] -> [**]
```

ou seja: o primeiro argumento de `map` é uma função[5] que aplicada a uma lista de objetos de tipo *, retorna uma lista de objetos do tipo **. Não é preciso que os tipos acima sejam diferentes. Diz-se que esta disciplina de tipos de dados é *polimórfica*, pois a mesma definição pode servir para um sem número de tipos. Assim, podemos passar `fat` como argumento para `map`, e ter

```
map fat [1..10]
[1,2,6,24,120,720,5040,40320,362880,3628800]
```

que é a lista dos fatoriais de 1 a 10. Note aqui que enquanto `fat` é sequencial por natureza, `map` não o é. No caso acima, pode ser detectado, usando *strictness analysis*[MeT][6], que `fat` pode ser aplicado, simultaneamente, aos 10 elementos da lista.

Se a linguagem é executada por um conjunto de processadores , as aplicações `f i` poderiam ser distribuídas para os processadores correspondentes, ou os que estivessem livres em um determinado momento. No caso sequencial, o processador único executa cada `f i` em sequência. Nenhum detalhe arquitetural é necessário aqui. Uma arquitetura (simples) que implementaria um mecanismo computacional paralelo para Miranda pode ser descrita como um conjunto de processadores ligados por uma rede (qualquer, rápida) de comunicação, compartilhando memória.

Vamos tentar outro exemplos maiores para ver como nos saimos com este paradigma de programação. Por exemplo, operações com matrizes são o pão de cada dia dos processadores que citamos no princípio deste artigo. Miranda pode[7] representar uma matriz como uma lista de listas, ou, no caso de números,

```
mat == [[num]]
```

com o operador `==` definindo a igualdade dos tipos `mat` e `[[num]]`. Começando com vetores, um vetor pode ser visto como uma lista de `num`, ou seja, `[num]`. Somar dois "vetores", então, é o mesmo que somar duas listas, o que pode ser definido em Miranda como

```
somav v1 v2 = [ v1!i + v2!i | i <- [0..#v1-1] ]
```

onde a expressão entre colchetes é uma *expressão ZF*, cujo significado pode ser traduzido para *lista de todos* $v1_i + v2_i$, *para* i *entre* 0 *e o índice máximo de* `v1`, que deve ser, neste caso, o mesmo de `v2`. Indexação é indicada por `!`, e a seta `<-` significa tirado de[8] Veja que, aqui, a recursão é implícita na descrição da solução. A partir daí, pode-se descrever, no mesmo estilo, a soma de duas matrizes pode ser definida por

```
somam m n = [ [m!l!c + n!l!c | c <- [0..#m!0-1]] | l <- [0..#m-1] ]
```

[5] Funções podem ser passadas como parâmetros e retornadas como resultado, sem restrição. Por isso Miranda é uma linguagem de *alta ordem*.

[6] Strictness Analysis é um método de detectar em que parâmetros uma função é estrita, isto é, para que parâmetros $f\perp = \perp$. O método pode ser usado para forçar cálculo em paralelo, como no caso acima.

[7] Estamos usando aqui uma representação trivial. É possível escrever uma representação muito mais elaborada de matrizes, usando *abstract data types*. O tipo de dado *array* não é fundamental em Miranda.

[8] Detalhes adicionais em [MeJ].

```
linear :: [(num,num)] -> (num,num)

linear lpairs = (a0,a1)
                where
                a0        = (sy*sx2 - sx*sxy)/divisor
                a1        = (m*sxy - sx*sy)/divisor
                divisor   = m*sx2 - sx^2
                m         = #lpairs
                abcissas  = map fst lpairs
                ordinates = map snd lpairs
                sx        = sum abcissas
                sy        = sum ordinates
                sx2       = sum (map sq abcissas)
                sxy       = sum (times abcissas ordinates)

times [] [] = []
times (a:x) (b:y) = a*b : times x y

sq x = x ^ 2

fst (a,b) = a
snd (a,b) = b
```

Figura 3: Regressão Linear em Miranda.

que descreve a soma de duas matrizes bidimensionais como a soma, por elemento, dos seus "vetores" (linhas).

As definições acima podem ser computadas em paralelo, sem restrição alguma. Como não há atribuição, não existe, é claro, o problema de dependência encontrado anteriormente nos DOs de Fortran. Aliás, outra das restrições importantes em Fortran eram os condicionais. Isto também não é uma limitação no caso funcional, e poderíamos definir

```
cmap cond f []     = []
cmap cond f (a:x) = f a : cmap cond f x, cond a
                  =   a : cmap cond f x, otherwise
```

que aplica `f` somente aos elementos aos elementos (da lista, terceiro parâmetro) para os quais `cond` é verdadeiro. É fácil ver que os `f a` podem ser computados em paralelo, e que isso só depende do valor de `cond a` para cada um. Um sistema paralelo assíncrono pode certamente calcular `cmap` desta forma.

Ainda não falamos dos tipos de dados (estão por vir), mas uma definição mais substancial é encontrada na Fig. 3, onde o método de regressão linear para ajuste de curvas é descrito em Miranda. O leitor pode identificar o paralelismo que existe na definição, e basicamente da mesma forma, o compilador pode detectar onde *forçar* o

paralelismo no cálculo, no caso de um sistema paralelo. `sum` é uma função padrão (como `map`) e pode ser definida por

```
sum [] = 0
sum (a:x) = a + sum x
```

Esta definição de soma, como se vê, não é paralela, não importa quantos processadores se tenha à mão. Em [HiS], Hillis e Steele apresentam exemplos do que se convencionou chamar *data parallel algorithms*. Os exemplos que usam lista e vetores são intrinsecamente imperativos e dependem da capacidade da linguagem expressar *update in place*. Sem o conceito de memória, as linguagens funcionais certamente não são capazes de expressar tal comportamento. Um dos exemplos é um algoritmo de complexidade $\log n$ em n processadores, para obter as somas parciais de uma lista. No fim do processamento, o i-ésimo elemento armazena a soma $\sum_0^i$. O último, é claro, contém a soma de todos, que é o mesmo valor computado por `sum` (ver [His], pp.1180). A soma (paralela) dos elementos pode ser descrita por

```
psum [a] = a
psum  x  = psum sumx, even sizex
         = psum (sumx ++ lastx), otherwise
           where
             sizex = #x
             sumx  = [x!i + x!(i+1) | i <- [0,2..sizex-2]]
             lastx = [x!(sizex-1)]

even n = n mod 2 = 0
```

Note que todas as somas em `sumx` podem ser calculadas em paralelo, e que os índices usados na expressão ZF poderiam corresponder, numa implementação paralela, aos índices dos processadores envolvidos. `where`, acima, introduz definições *locais* a `psum`. As considerações sobre o paralelismo da definição acima independem do hardware, e a lista original é mantida intacta.

A implementação atual de Miranda, infelizmente, leva tempo $O(i)$ para computar `x!i`, mas este não é um problema fatal. Há várias formas de compensar isto, e há trabalhos em andamento no sentido de implementar *functional arrays*. A Fig. 4 mostra um *script* contendo algumas operações gerais com matrizes, com tipos de dados definidos pelo programador. Apenas a parte mais elementar do sistema de tipos é usado no *script* mostrado.

É provável que alguns problemas serão intrisecamente mais lentos na sua solução funcional do que quando descritos por uma linguagem que é perfeitamente *casada* com o hardware. Mas deve ser lembrado aqui que nos anos 60 muita gente era contra Fortran porque *assembler* era mais rápido. Talvez hoje a questão esteja de volta no mesmo terreno.

Mais ainda, o problema que discutimos no início, de *programabilidade* em Fortran ou outras linguagens imperativas, é ainda mais importante em grandes pacotes de software. Com as arquiteturas mudando constante e cada vez mais rapidamente, não há porque se investir fortunas em pacotes que são ligados, diretamente, a hardware específico que pode ter uma vida muito curta.

```
|| define (por igualdade) alguns tipos usados no script
mat   * == [[*]]
opvvv * == * -> * -> *
opvxy * ** *** == * -> ** -> ***
opnnn   == opvvv num
matnum  == mat num

|| op_num_mat opera somente com matnums, e sua funcao
|| e' op n a todos os elementos de m

op_num_mat :: opnnn -> num -> matnum -> matnum
op_num_mat op n m = map (map (op n)) m

|| op_val_mat opera somente com mat *, e sua funcao
|| e' op v a todos os elementos de m
|| note que nao era preciso se ter op do tipo opvvv *

op_val_mat :: (opvvv *) -> * -> mat * -> mat *
op_val_mat op v m = map (map (op v)) m

|| op_mat_mat faz operacoes genericas com matrizes, elemento
|| a elemento. as matrizes nao precisam ser do mesmo tipo,
|| mas devem ser bidimensionais

op_mat_mat :: (opvxy * ** ***) -> mat * -> mat ** -> mat ***
op_mat_mat op m1 m2 = map (dop op) (matlpar m1 m2)

matlpar :: mat * -> mat ** -> [([*],[**])]
matlpar [] [] = []
matlpar (a:x) (b:y) = (a,b) : matlpar x y

dop op ([],[]) = []
dop op ((a:x),(b:y)) = a $op b : dop op (x,y)

mul_mat m1 m2 = error "mul_mat: lin m1 ~= col m2", linm1 ~= colm2
              = [ [m1!i!j * m2!j!i | j <- [0..colm2]]
                                   | i <- [0..linm1] ], otherwise
                where
                linm1 = lin m1 - 1
                colm2 = col m2 - 1

lin m = #m
col m = #(hd m)
```

Figura 3: Operações com arrays.

5 Conclusão

O objetivo deste artigo foi o de mostrar que muitos dos princípios utilizados, hoje em dia, para programação de supercomputadores, não são na realidade, dignos de menção como parte de uma engenharia. Eles são, talvez, mais apropriados a uma caixa de ferramentas do mecânico de manutenção dos compiladores de linguagens imperativas usados nos supercomputadores.

À medida que se torna evidente, em todo o mundo, que o processo de desenvolvimento de software tem que sair da fase do *eu acho* para uma prática mais rigorosa, também é tempo de parar de se tentar programar máquinas que foram sonhadas com pouca ou nenhuma consideração pelas linguagens de programação que suportariam o hardware. Nas palavras de Wirth (em [WiA])

> ...I finally decided to dig into hardware design. This decision was reinforced by my old disgust with existing computer architectures that made life miserable for a compiler designer...
>
> ...It is true that we live in a complex world and strive to solve inherently complex problems, which often do require complex mechanisms. However, this should not diminish our desire for elegant solutions, which convince by their clarity and effectiveness. Simple, elegant solutions are more effective, but they are harder to find...

o que é um tema interessante para refletir enquanto se lê o manual de CFT...

No caso das linguagens, é bom que se diga que uma sintaxe *bonita* não significa melhor usabilidade e programas mais legíveis e manipuláveis. Durante muito tempo, na Europa, a pesquisa em linguagens de programação tem derivado diretamente da pesquisa em semântica, e a pesquisa em arquitetura, da primeira. Um bom exemplo disso é a quádrupla CSP, occam, Transputer e Computing Surface.

Finalmente, este artigo *não* tenta estabelecer que as linguagens funcionais são a única alternativa. Apenas tenta mostrar que as mesmas são uma das alternativas. Em outras áreas, a atividade em programação concorrente deve certamente oferecer uma eficiência mais bem comportada do que Fortran, em futuro próximo.

Com a tecnologia atual de implementação de linguagens funcionais em máquinas sequenciais, já se consegue performances comparáveis às linguagens imperativas. Como as linguagens não têm efeitos colaterais, as perspectivas de seu uso em ambientes paralelos são extremamente boas, em contraste com as linguagens imperativas. E mesmo que se queira continuar programando imperativamente, as linguagens funcionais oferecem um ambiente único de especificação e construção de protótipos, e poderiam ser usadas como *front-end* no processo de desenvolvimento de código imperativo.

6 Agradecimentos

Este trabalho é apoiado pelo Departamento de Informática da UFPE e pelo Computing Laboratory da University of Kent. O autor agradece ao Prof. David Turner e o Theoretical Computer Science Group, pela oportunidade de trabalhar em Kent.

7 Referências

[CDC] *Control Data Corporation,* Cyber 200(205) Computer System, **CDC Pub.** 60256020, 1981.

[Hil] *W. D. Hillis,* The Connection Machine, **MIT Press, Cambridge, Mass.,** 1985.

[HiS] *W. D. Hillis and G. L. Steele, Jr.,* Data Parallel Algorithms, **CACM 29,** (12), Dec. 1986.

[MCS] *Meiko Corp,* The Meiko Computing Surface, (literatura avulsa).

[Mei] *S. L. Meira,* Mathematical Software in Applicative Languages, **DI/UFPE Rep.** (em preparo).

[MeJ] *S. L. Meira,* Programação Funcional. Notas de Curso, **JAI/1986,** Recife PE.

[MeT] *S. L. Meira,* On the Efficiency of Applicative Algorithms. PhD Thesis, **Comp. Lab., Unikent,** 1985.

[Per] *R. H. Perrot,* A Language for Array and Vector Processors, **ACM TOPLAS 1,** (2).

[Pet] *W. P. Petersen,* Vector Fortran for Numerical Problems on CRAY-1, **CACM 26,** (11), Nov. 1983.

[PeZ] *R. H. Perrot, A. Zarea-Aliabadi,* Supercomputer Languages. **Comp. Surveys 18,** (1), Mar. 1986.

[Red] *S. F. Reddaway,* The DAP Approach, **Infotech SoAR 2,** Infotech Intl., 1979.

[Rus] *R. M. Russell,* The Cray-1 Computer System, **CACM 21,** (1).

[Tur] *D. A. Turner,* An Overview of Miranda, **ACM SigPLan Notices,** Dec. 1986.

[WiA] *N. Wirth,* From Programming Language Design to Computer Construction, **CACM 28,** (2), Feb. 1985.