UM SISTEMA LISP PARALELO

Wellington Santos Martins [1]
Geraldo F. Guidacci da Silveira [2]

Resumo

Este artigo apresenta um sistema para o processamento paralelo do Lisp, constituído de uma extensão do compilador Lisp e da máquina S.E.C.D., bem como da arquitetura do sistema multiprocessador. Esta arquitetura é composta por um conjunto de máquinas S.E.C.D. e de um processador dedicado para a coleta de lixo ("garbage collector"). A extração do paralelismo é automática, não sendo da responsabilidade do programador a indicação de pontos de paralelismo. Para a sua avaliação este sistema foi inicialmente desenvolvido em C para um microcomputador compatível com o IBM PC/AT e o sistema operacional D.O.S. O sistema multitarefas foi desenvolvido em C, assim como o ambiente de programação para o Lisp. O compilador para esta linguagem é escrito no próprio Lisp. A análise de desempenho de alguns programas é utilizada na avaliação do sistema proposto.

Abstract

This paper presents a system for the parallel processing of Lisp, composed of an extension of the Lisp compiler and the S.E.C.D. machine, as well the multi-processor architecture. This architecture is constituted by a collection of S.E.C.D. machines and a processor dedicated to the garbage collection. The parallelism extraction is automatic, with the programmer free from the responsibility of indicating points of parallelism. For an evaluation, this system was implemented in the C language for an IBM PC/AT compatible computer and the D.O.S. operating system. The multi-task system was build in C, as well as the environment for the Lisp program development. The Lisp compiler was implemented in this same language. For an evaluation of the proposed system, the performance analysis of some bench-mark programs is utilized.

---

[1] Mestre em Sistemas de Computação pela PUC/RJ * Professor da Universidade Federal de Goiás - DEI * Campus Samambaia, CP:131, CEP: 74000, Goiânia (GO) * Fone: (062) 205-1000 r. 132. * C-elt: VELL@LNCC.BITNET

[2] Docteur d'Etat / Sciences pela Université Paul Sabatier/Toulouse * Professor da Pontifícia Universidade Católica do Rio de Janeiro - DEE * Rua Marquês de São Vicente, 255, Gávea, CEP: 22453, Rio de Janeiro(RJ) * Fone: (021) 529-9445.

1. Introdução

O desempenho dos supercomputadores convencionais está se aproximando do seu limite físico, que é imposto pela velocidade da luz. Assim, mesmo que seja construído uma unidade de processamento com dimensões bastante reduzidas e que opere com sinais enviados na velocidade da luz, conseguiríamos apenas a execução de alguns bilhões de instruções por segundo, o que representa muito pouco se comparadas às atuais aplicações (processamento de imagem em tempo real, cálculo de barragens, estudo das turbulências, visão por computador, reconhecimento de voz). A solução para este problema pode ser alcançada através da utilização do paralelismo.

O avanço da tecnologia dos semicondutores e VLSI vem fazendo com que as unidades de processamento individuais tornem-se cada vez mais baratas, o que vem possibilitando a implementação de computadores com vários processadores a um custo não muito alto. As inúmeras possibilidades de organização de um grande número de processadores tem produzido uma grande variedade de computadores paralelos.

O desafio da criação de computadores paralelos não se restringe à sua estrutura física ("hardware"), mas, principalmente, à sua programação ("software"). Desta maneira, embora não seja muito difícil construir uma máquina paralela, sua programação não é muito trivial. A busca por uma linguagem adequada à programação paralela só está começando.

Algumas linguagens descendentes das tradicionais vem sendo estudadas. Tais linguagens possuem construtores que permitem ao programador iniciar e coordenar várias tarefas em paralelo. Tal solução torna-se impraticável se tivermos milhares de tarefas para serem controladas. Para que tenhamos um alto grau de paralelismo, é necessário que a linguagem utilizada possua grande quantidade de paralelismo implícito a fim de amenizar a carga imposta ao programador.

Apesar de várias outras linguagens não tradicionais estarem sendo estudadas, uma atenção especial vem sendo dada à linguagem Lisp, pois além de possuir grande quantidade de paralelismo implícito, é a mais utilizada no campo da Inteligência Artificial.

Este trabalho se insere neste contexto, onde são abordados aspectos referentes tanto à arquitetura de uma máquina paralela, quanto à oportunidade para extração do paralelismo de programas Lisp.

Com relação à linguagem utilizada, optamos por deixar o programador livre de qualquer indicação de pontos de paralelismo, isto é, a extração do paralelismo é automática e a cargo do compilador. A linguagem utilizada, portanto, não possui construtores adicionais para a indicação do paralelismo. Outro

ponto importante é que a linguagem utilizada não permite o uso da operação de atribuição, que implicaria na existência de efeitos colaterais e tornaria o controle do paralelismo mais complexo.

A arquitetura proposta é um sistema multiprocessador com memória compartilhada. Sua descrição não leva em conta detalhes de implementação sendo descrita através de blocos funcionais.

Para a simulação deste sistema foi criado um ambiente de desenvolvimento de programas em Lisp (utilizando a linguagem C) que conta com editor, compilador, gerente de memória e máquinas virtuais, representadas por processos de um núcleo de multiprogramação. A análise do desempenho de alguns programas é utilizada para a avaliação do sistema proposto [Mart 90].

2. A linguagem utilizada

A linguagem utilizada é uma extensão do dialeto Lisp chamado LispKit, dialeto este proposto por Henderson [Hend 80]. Esta linguagem não possui construções que causam efeitos colaterais, sendo portanto enquadrada na categoria de Lisp Puro.

Os pontos encontrados para a extração do paralelismo na linguagem LispKit foram: chamada de função, bloco let e primitiva não-determinística.

A extração do paralelismo é automática, não devendo o programador se preocupar em indicar pontos de paralelismo. Assim, programas que já existem ou mesmo aqueles que tenham sido construídos sem qualquer atenção para o paralelismo, podem usufruir da máquina paralela aqui proposta. Isto deve-se ao fato do compilador se encarregar de inserir instruções que mais tarde podem permitir a execução paralela.

Passamos a seguir a descrever a extensão proposta para a linguagem LispKit, ou seja, as modificações realizadas no compilador.

2.1 Chamada de função

Normalmente quando invocamos uma chamada de função, seus argumentos são avaliados um após o outro e posteriormente a função é aplicada aos valores obtidos.

Nossa proposta é que os argumentos da função sejam avaliados em paralelo, desde que existam máquinas suficientes para tal. Associado a cada argumento que estiver sendo avaliado em paralelo existirá um objeto que representa um pseudo-resultado e, inicialmente, posssuirá o estado *não avaliado*. Mais tarde, quando o valor de algum destes objetos for necessário, seu estado é verificado. Se for *avaliado*, o processo segue normalmente, caso contrário ele é bloqueado.

Aqui vale esclarecer que inicialmente o programa a ser executado é considerado como o processo pai e à medida que são disparadas operações em paralelo são criados processos filhos, que podem, se necessário, gerar outros filhos.

Caso a máquina esteja saturada (com todos os processadores ocupados), a execução é automaticamente chaveada para o modo sequencial de operação.

Embora pareça que pouco paralelismo possa ser alcançado numa chamada de função, nos casos onde temos esta chamada recursiva, a quantidade de paralelismo torna-se considerável.

O código gerado pelo compilador é ligeiramente diferente daquele proposto por Henderson.

*(fun e1 ... ek) * n =*

*= (LDC NIL) | (FPE ek*n | (RTR)) | (CONS) | ... | (FPE e1*n | (RTR)) | (CONS) | fun*n | (AP)*

Temos assim a inclusão de duas novas instruções para a máquina S.E.C.D.. As transições de estados, ou seja, a transferência de conteúdo dos registradores para estas novas instruções são:

```
s   e  (FPE c.c')  d  -->  (α.s)  e  c'  d    processo pai
                           NIL    e  c   NIL  processo filho
```

ou, caso a máquina esteja saturada,

```
s   e  (FPE c.c')  d  -->  ((c.e).s)  e    c'   d

(x.s)  e    (RTR)     d  -->    (x.s)     e  (RTR)  d
```

A instrução FPE ("Fire Parallel Evaluation") realiza o disparo de uma avaliação em paralelo, caso exista algum processador disponível. O objeto deixado no topo da pilha, α, possui o estado *não avaliado* e assim a execução pode prosseguir até que o valor de α seja necessário.

A instrução RTR ("Return Result") copia o resultado obtido, situado no topo da pilha, para o endereco do objeto α e altera seu estado para *avaliado*.

2.2 Bloco let

O bloco **let** permite que o programador crie variáveis locais ao bloco em questão, isto contribui para que o programa possa ser melhor entendido além de simplificar o trabalho do programador. Por exemplo, ao invés de utilizarmos

```
(func (add '1 n) (mul (add '1 n) '2) (div (add '1 n) '2))
```

podemos usar

```
(let (func x (mul x '2) (div x '2))
   (x . (add '1 n)) )
```

A avaliação do bloco **let**[3] normalmente envolve a avaliação sequencial de cada uma de suas expressões (e1...en), a ligação dos resultados obtidos às variáveis (x1...xn) e a avaliação do corpo do bloco (e).

A modificação proposta para o código gerado pelo compilador para o bloco **let** é idêntica àquela proposta para a chamada de função, pois os dois casos são semanticamente equivalentes. Assim, as expressões (e1...en) bem como o corpo do bloco (e) são avaliados em paralelo, caso haja máquinas, e cada variável (x1...xn) é associada a um objeto representando um pseudo-resultado possuindo o estado *não avaliado*. À medida que o valor de alguma destas variáveis for necessário, seu estado é inspecionado e, caso seja *avaliado*, o processo segue normalmente, caso contrário ele é bloqueado.

O código gerado pelo compilador para a implementação deste esquema é

*(let e (x1 . e1) ... (xk . ek)) * n =*

*=(LDC NIL) | (FPE ek*n | (RTR)) | (CONS) |...| (FPE e1*n | (RTR)) | (CONS) | (LDF e*m | (RTN)) |(AP)*

2.3 Primitiva não-determinística or

A modificação que aqui propomos interpreta a expressão (**or** e1 e2) da seguinte maneira:

a) Se houver máquina disponível, utilize-a para seguir o caminho definido por e2 e continue a execução com o caminho definido por e1.

b) Caso não haja máquina disponível, comporte-se como antes, ou seja, como definido por Henderson (siga o caminho definido por e1 e guarde e2 para uma possível utilização posterior).

A primeira máquina que encontrar algum resultado fará com que todas as outras parem.

Como exemplo podemos citar o problema das rainhas que se resume em posicionar **n** rainhas num tabuleiro **n** x **n** de tal modo que elas não se ataquem de acordo com as regras do jogo de xadrez.

---

[3] (let e (x1 . e1) (x2 . e2) .... (xk . ek))

Uma função para resolver este problema pode ser:

```
(letrec queensoln
   (queensoln lambda (n)
      (addqueen '1 n 'nil) )
   (addqueen lambda (i n place)
      (let (if (attacks i j place)
               (none)
               (let (if (eq i n)
                        newplace
                        (addqueen (add '1 i) n newplace))
                  (newplace . (cons (cons i j) place)) ))
        (j . (choice n)) ))
   (choice lambda (n)
      (if (eq n '1)
          '1
          (or (choice (sub n '1)) n) ))
   (attacks lambda (i j place)
      (if (eq place 'nil)
          'f
          (let (if (eq i il)
                   't
                   (if (eq j jl)
                       't
                       (if (eq (add i j) (add il jl))
                           't
                           (if (eq (sub i j) (sub il jl))
                               't
                               (attacks i j (cdr place)) ))))
            (il . (car (car place)))
            (jl . (cdr (car place))) ))))
```

Aqui, a expressão (or (choice (sub n '1)) n))) chamada recursivamente, faz com que várias máquinas sejam disparadas, cada uma tentando um solução. Logo que um resultado for encontrado, todas as máquinas serão ordenadas a parar.

Para implementarmos esta modificação não necessitamos alterar o código gerado pelo compilador, o já proposto por Henderson, mas precisamos definir outras mudanças de estado para as instruções SOR e NON. Estas mudanças são:

```
s  e  (SOR c1 c2.c) d r  -->   s  e  c1  (c.d)  r   processo 1
                               s  e  c2  (c.d)  r   processo 2
```

ou, caso a máquina esteja saturada,

```
s  e  (SOR c1 c2.c) d    r   -->  s  e  c1  (c.d)  (s e c2 (c.d).r)

s  e  (NON)  d  (s' e' c' d'.r)   -->   s'  e'  c'  d'  r

   ou

s  e  d  nil   -->   finaliza e vai para a fila de ociosas
```

A instrução SOR ou dispara uma avaliação paralela, caso exista alguma máquina disponível, ou comporta-se como definido por Henderson. A instrução NON ou finaliza a avaliação ou comporta-se como definido na máquina original.

Uma observação importante é que programas que utilizem as primitivas **or** e **none** não podem ter a avaliação dos argumentos de suas funções, ou mesmo as variáveis de blocos **let**, realizadas em paralelo, pois poderia ocorrer uma situação como a descrita a seguir. Suponha a seguinte chamada de função: (func e1 (or e2 e3))

A função func, sendo avaliada pela máquina 1 dispararia a avaliação de e1 na máquina 2, a avaliação de (**or** e2 e3) na máquina 3 e aguardaria o resultado destas duas máquinas. A máquina 3 seguiria o caminho definido por e2 e dispararia a máquina 4 para avaliar o caminho definido por e3. As máquinas 3 e 4 encontrariam a instrução NON e finalizariam. A função func não receberia nenhum resultado de (or e2 e3) !.

Devido a esta incompatibilidade, desenvolvemos duas versões para o compilador, uma para os programas que não utilizam as primitivas **or** e **none** e outra para os que assim o fazem.

3. A arquitetura proposta

A idéia inicial de uma arquitetura adequada à linguagem Lisp aqui estudada, foi a de uma extensão da máquina SECD [Land 64] de modo que várias destas máquinas pudessem operar em paralelo. Esta extensão implica basicamente num sistema multiprocessador onde cada unidade processadora é uma máquina SECD acrescida de uma camada de comunicação.

A arquitetura proposta utiliza um sistema com memória compartilhada. Neste sistema a comunicação é realizada através da memória global compartilhada. A conexão entre os processadores e a memória pode ser feita através de uma rede de interconexão que basicamente pode ser um barramento comum, um barramento cruzado ou uma rede de múltiplos estágios.

Um dos fatores que limita a expansão destes sistemas é a degradação de desempenho devido à contenção de memória quando dois ou mais processadores tentam acessar a mesma unidade de memória. Outro fator é a própria rede de interconexão, pois esta influi na contenção de comunicação, que acontece quando mais de um processador necessita utilizar um mesmo caminho, ou trecho de caminho, na rede em questão.

Embora, como visto anteriormente, os sistemas fortemente acoplados possuam problemas de contenção de memória e comunicação, existem maneiras de se amenizar estes problemas de modo a termos um bom desempenho. Os pontos a serem atacados para conseguirmos isto são: organização da memória e o projeto da rede de interconexão.

A arquitetura aqui proposta é então um sistema multiprocessador com memória compartilhada conforme mostra a Figura 1.

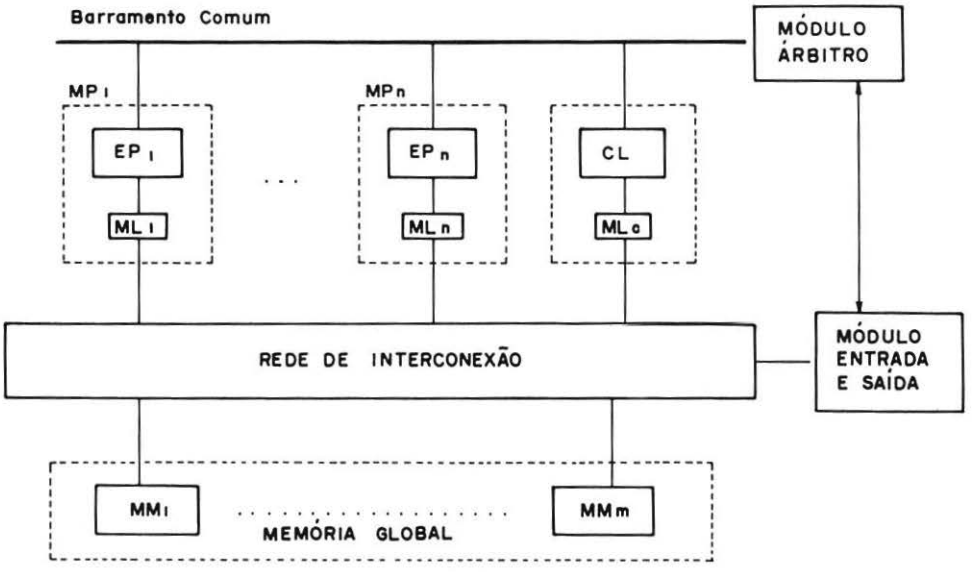


FIGURA 1 Arquitetura da máquina proposta

O sistema é constituído de vários módulos processadores, uma memória global, um módulo árbitro e um módulo de entrada e saída.

O módulo processador representa a máquina SECD acrescida de uma camada de comunicação. Um dos módulos processadores é dedicado à coleta de lixo em paralelo com a execução dos programas Lisp. O algorítmo utilizado para efetuar esta coleta de lixo foi o definido por Dijkstra [Dijk 78], que teve sua fase de marcação alterada visando um melhor desempenho [Mart 90]. Este mesmo algoritmo já foi implementado em "hardware" através de um controlador microprogramado e mostrou-se bastante eficiente [Oliv 90] [Mats 90].

O módulo árbitro gerencia todos os módulos processadores. Ele possui o estado de cada módulo processador e mantém numa fila aqueles que estiverem ociosos. Durante a execução de um programa, o árbitro é consultado sempre que uma operação em paralelo possa ser efetuada. Nenhum módulo processador pode se comunicar diretamente com outro, isto só é possível através do árbitro.

O módulo de entrada e saída é o elo de ligação entre a máquina e o mundo exterior. É através deste módulo que o programador

constroi seus programas e obtem seus resultados. Este módulo também é responsável pela carga de programas na memória da máquina e o disparo de sua execução.

Para a simulação desta arquitetura os problemas de contenção não foram levados em consideração, pois isto implicaria num sistema bem mais complexo.

4. Resultados Obtidos

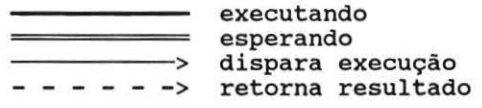
Os resultados obtidos com a avaliação "paralela" são mostrados através dos seguintes tipos de gráficos :

1) Quantidade de máquinas X Tempo gasto para execução

Neste gráfico são mostradas três curvas, cada uma correspondendo a uma aplicação da função com um determinado argumento. Para efeito de comparação, representamos nestes gráficos tanto o desempenho do programa na máquina sequencial quanto na máquina paralela. Assim, temos no eixo referente ao número de máquinas duas indicações do número 1, a primeira se refere aos resultados obtidos na versão sequencial enquanto o segundo e os demais pontos dizem respeito à máquina paralela. A diferença dos valores obtidos no modo sequencial e no modo paralelo com somente uma máquina é devida ao tempo gasto com as instruções a mais que foram inseridas pelo compilador, que na verdade, não surtem efeito nenhum quando o sistema só conta com uma máquina.

2) Tempo X Número da máquina

Este gráfico mostra o comportamento de cada máquina (total de dez) ao longo do tempo gasto para a execução da aplicação da função para um determinado argumento. As convenções adotadas foram:



3) Tempo X Quantidade de máquinas

Este gráfico mostra a quantidade de máquinas (10 no máximo) que estão executando a cada instante.

O tempo representado nestes gráficos é normalizado e representa, na verdade, o número de ciclos executados, onde consideramos que todas instruções gastam o mesmo tempo.

Os exemplos aqui considerados são: Quick-Sort e Queens. O primeiro se aproveita da chamada de função e do bloco **let** para a extração do paralelismo, enquanto o segundo trabalha com a primitiva não-determinística **or** e a primitiva **none**.

4.1 Quick Sort

A função qsort (de Quick Sort) recebe como argumentos uma lista de números e retorna uma lista com estes números ordenados.

```
(letrec qsort
   (qsort lambda (x)
      (qs x 'nil))
   (qs lambda (x rest)
      (if (eq x 'nil)
          rest
          (let (qs (car parts)
                   (cons (car x) (qs (cdr parts) rest)))
            (parts . (partition (car x) (cdr x))) )))
   (partition lambda (elt lst)
      (if (eq lst 'nil)
          (cons 'nil 'nil)
          (let (if (leq (car lst) elt)
                   (cons (cons (car lst) (car cdrparts))
                         (cdr cdrparts))
                   (cons (car cdrparts)
                         (cons (car lst) (cdr cdrparts))) )))
            (cdrparts . (partition elt (cdr lst))) ))))
```

Os resultados obtidos são mostrados a seguir.

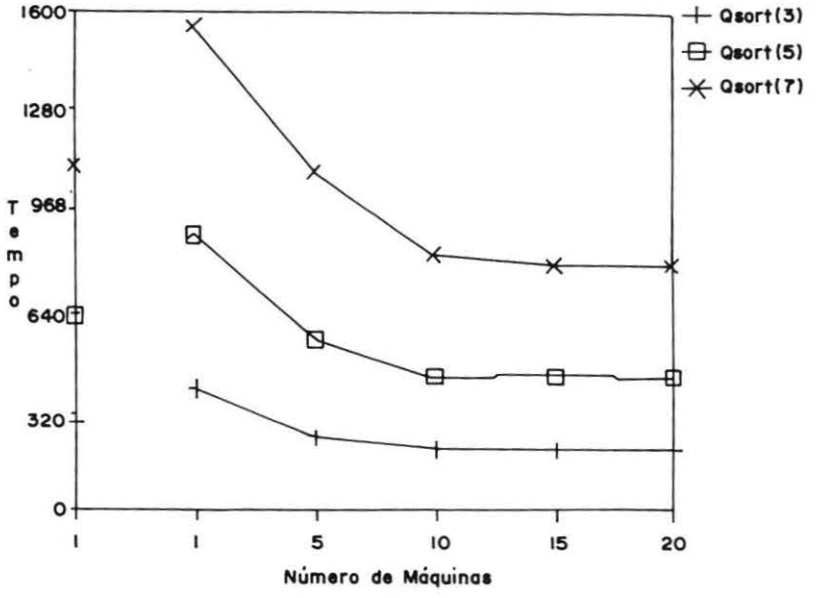


FIGURA 2 Resultados da avaliação da função qsort

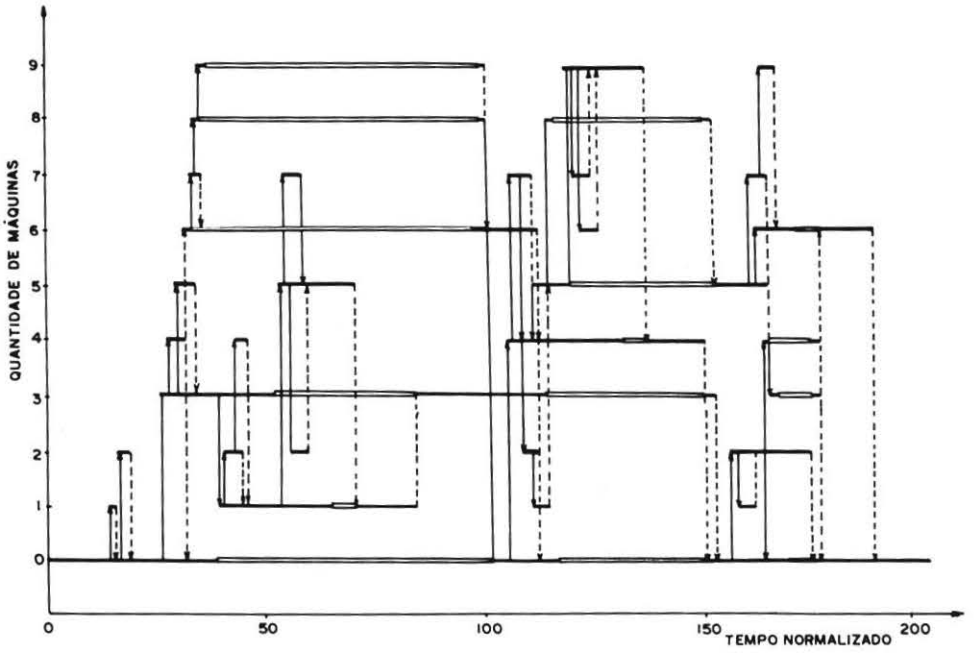


FIGURA 3 Atuação das máquinas no cálculo de QSORT ((3 2 1))

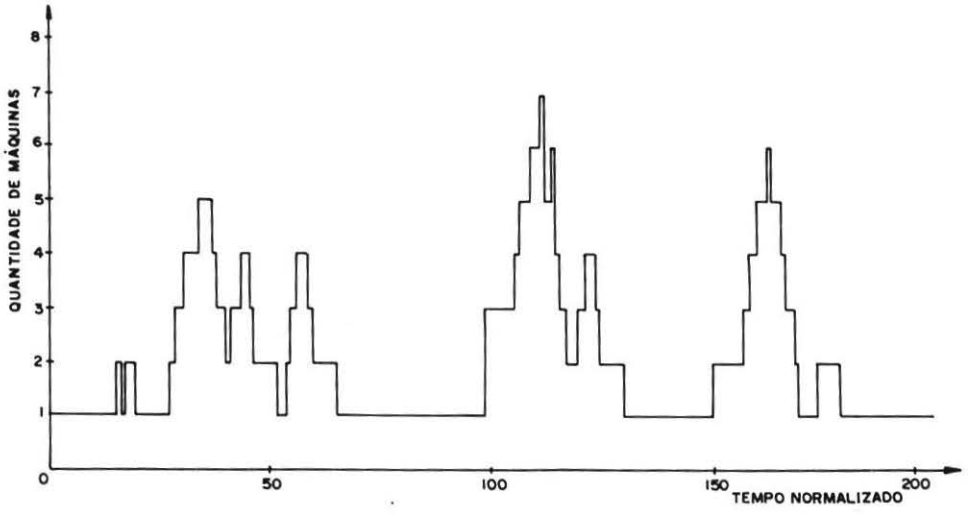


FIGURA 4 Máquinas trabalhando para o cálculo de QSORT ((3 2 1))

Comentários:

Para o gráfico da Figura 2 os argumentos utilizados foram (3 2 1), (5 4 3 2 1) e (7 6 5 4 3 2 1) que representam listas com 3, 5 e 7 elementos respectivamente. No gráfico fazemos referência à Qsort(n) ao invés de Qsort((n ... 1)) por questões de simplificação. Deste gráfico verificamos que à medida que aumentamos o número de máquinas conseguimos uma diminuição do tempo necessário à execução do programa, embora exista um limite para isto. Este limite é função do tamanho da lista dada como argumento.

O gráfico da Figura 3 mostra que todas as 10 máquinas foram requisitadas durante a avaliação de Qsort((3 2 1)). Podemos notar que várias das operações exportadas foram pequenas, pois gastaram muito pouco tempo para serem concluídas. Verificamos também a existência de várias máquinas em estado de espera.

O gráfico da Figura 4 mostra praticamente três regimes nas quais tivemos várias máquinas operando em paralelo. Embora isto tenha ocorrido por um tempo não muito prolongado, contribuiu bastante para uma execução mais rápida.

4.2 Queens

A função definida para solucionar o problema das rainhas (ver item 2.3) utiliza a primitiva não-determinística **or** e a primitiva **none**. Neste caso, cada máquina que for disparada em paralelo tentará achar uma solução para o problema. A primeira solução encontrada é dada como resposta e posteriormente todas as máquinas são ordenadas a parar. Os resultados obtidos são mostrados no gráfico da Figura 5.

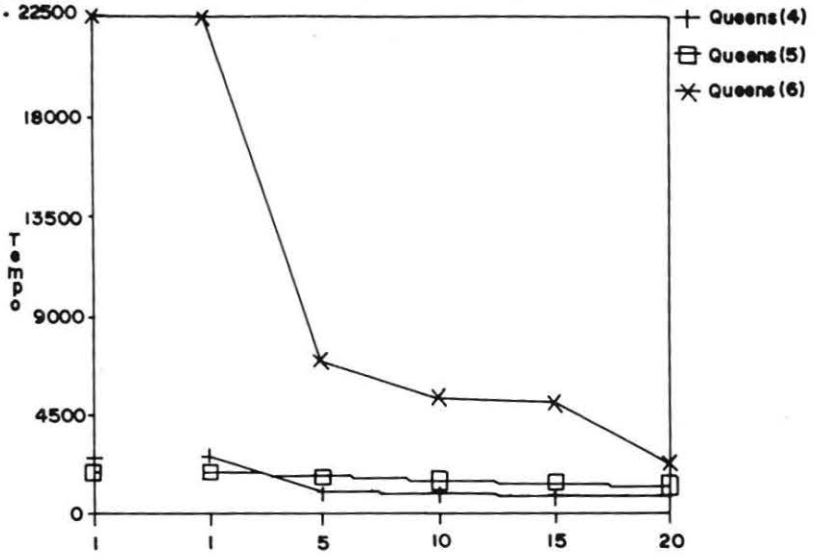


FIGURA 5 Resultados obtidos para o cálculo de queensoln

Comentários:

Neste exemplo só mostramos um gráfico, pois os outros dois não seriam muito úteis visto que o uso da primitiva não-determinística conduz a avaliações paralelas totalmente independentes.

Do gráfico da Figura 5 podemos notar que existe uma diminuição considerável do tempo gasto para encontrar uma solução. No caso de Queens(6) passamos de 22500 no modo sequencial para 2262 no modo paralelo com 20 máquinas !. Neste caso, o resultado obtido com a máquina sequencial é identico ao obtido com a máquina paralela com somente uma máquina pois os dois casos trabalham com o mesmo conjunto de instruções.

Outro ponto importante a ser lembrado aqui, é que programas que utilizam a primitiva não-determinística podem, mesmo não existindo um paralelismo real, encontrar a solução mais rapidamente que a implementação sequencial, pois os vários caminhos vão sendo examinados concorrentemente.

5. Conclusões

Os resultados obtidos demonstram que, dependendo do algorítmo utilizado, os programas Lisp podem ter seu desempenho bastante melhorado num sistema que possibilite a extração de seu paralelismo inerente.

Verificamos uma baixa utilização das máquinas disponíveis. Isto se deve em parte à tentativa de extração do paralelismo sem a intervenção do programador ou mesmo de um estudo prévio do código no sentido de avaliarmos a viabilidade ou não da exportação de uma operação.

O sistema desenvolvido teve como principal fator limitante a máquina utilizada, no caso um microcomputador IBM PC/AT. Seu sistema de memória segmentada limita o número máximo de células em 64 Kbytes, o que impede que programas mais complexos possam ser executados.

Como sugestão propomos que um sistema semelhante seja desenvolvido numa máquina de maior porte e que o próximo passo seja a implementação em "hardware" de um protótipo. Os efeitos da contenção de memória devem ser levados em consideração, pois este é um fator que limita o desempenho do sistema. O uso de combinadores seria bastante interessante, pois implica na ausência de um ambiente ("environment") que é fonte de grande parte dos problemas de contenção.

6. Referências Bibliográficas

[Dijk 78] Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S. and Steffens, E. F. M.
On-the-fly garbage collection: An exercise in cooperation
Commun. ACM 21, 11, Nov. 1978, 966-975.

[Hend 80] Henderson, P.
Functional Programming: Application and Implementation
Englewood Clifts, N.J., Prentice-Hall, 1980.

[Land 64] Landin P. J.
The mechanical evaluation of expressions
Comput. vol. J.6, Jan. 1964, 308-320.

[Mart 90] Martins, W. S.
UM SISTEMA LISP PARALELO: Concepção e Simulação
Tese de Mestrado, DEE/PUC-RJ, 1990.

[Mats 90] Matsumoto, Kensuke
Controlador Micro-programado para a Coleta de lixo Concorrente num Ambiente LISP
Trabalho de Fim de Curso, DEE/PUC-RJ, 1990.

[Oliv 90] Oliveira, Fabiano Saldanha G.
ENTERPRISE: uma Arquitetura LISP
Tese de Mestrado a ser apresentada, DEE/PUC-RJ, 1990.