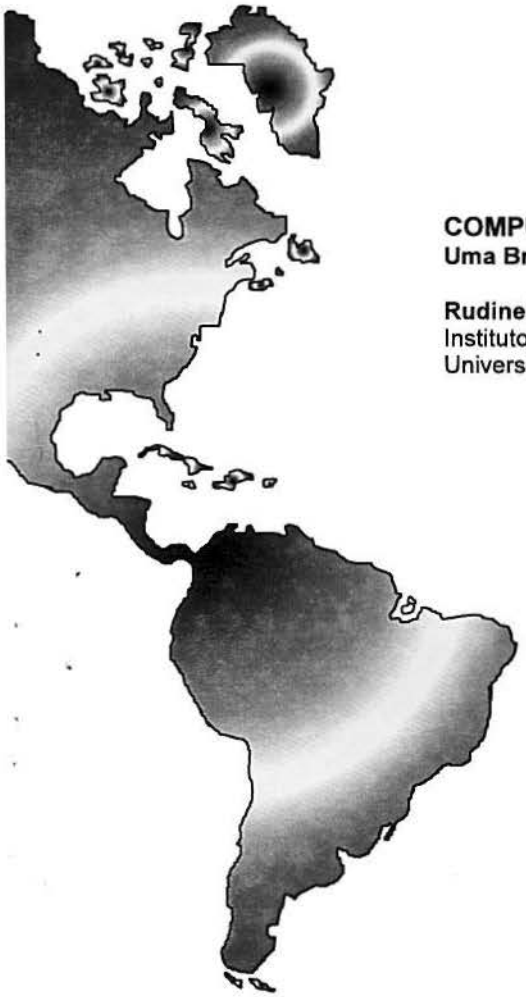
XX Congresso Nacional de Matemática Aplicada e Computacional CNMAC

Sociedade Brasileira de Matemática Aplicada e Computacional



MINICURSO

COMPUTAÇÃO PARALELA:
Uma Breve Introdução

Rudinei Dias da Cunha
Instituto de Matemática
Universidade Federal do Rio Grande do Sul

08 a 12 de setembro de 1997
SERRANO CENTRO DE CONVENÇÕES
GRAMADO RS

SBMAC

Secretaria da SBMAC

Rua Lauro Muller, 455
22290-160 Rio de Janeiro, RJ
Tel.: (021) 541-2132 Ramal 166

XX CONGRESSO NACIONAL DE MATEMÁTICA
APLICADA E COMPUTACIONAL - CNMAC
Sociedade Brasileira de Matemática Aplicada e Computacional - SBMAC

COMPUTAÇÃO PARALELA:
uma breve introdução

Rudinei Dias da Cunha
Instituto de Matemática
Universidade Federal do Rio Grande do Sul

GRAMADO - RS

1997

O autor é Ph.D. (Computer Science) pela University of Kent at Canterbury, G.B. e Bel. em Ciências de Computação pela Universidade Federal do Rio Grande do Sul.

Atualmente é professor do Instituto de Matemática e do CPG em Matemática Aplicada da UFRGS e membro do Comitê Diretor do Laboratório Integrado de Computação Científica daquele Instituto.

Sumário

1. Introdução

Em nossa opinião, o termo "computação paralela" refere-se, de forma geral, ao uso de múltiplos recursos computacionais cooperando para a consecução de uma determinada tarefa.

Dessa forma, aplicações que a maioria de nós tem contato no seu cotidiano, como por exemplo a operação de caixas bancárias automáticas, enquadram-se nesse tipo de computação.

A computação paralela, então, engloba uma variada gama de aplicações, desde aquelas de cunho científico "per se" - nos campos da Física, Química, Engenharias e outros - bem como aquelas de cunho comercial ou gerencial.

No entanto, no cerne de todas essas aplicações que possam ser consideradas como sendo "computações paralelas", encontram-se as seguintes idéias:

coordenação de atividades

distribuição de dados

Conforme veremos ao longo deste trabalho, essas idéias serão responsáveis pelo desenvolvimento de aplicações paralelas, desde o mais simples algoritmo até um grande sistema, talvez até mesmo distribuído geograficamente.

No tocante à *computação paralela científica* temos, no entanto, um grande problema - *a existência de máquinas cada vez mais poderosas trouxe consigo uma necessidade de reciclagem dos princípios que norteiam o desenvolvimento de algoritmos.*

Infelizmente, o desenvolvimento do *hardware* e do *software* paralelos não ocorreu de forma similar, causando muitos potenciais usuários de computadores paralelos a não valerem-se dos mesmos, pelo simples fato de que *programar máquinas paralelas não é uma tarefa fácil.*

Vejamos, por exemplo, um fato muito comum, e certamente vivenciado em inúmeros centros de computação paralela no mundo: um pesquisador desenvolve uma determinada aplicação e agora necessita validá-la para diferentes situações, o que vai demandar, no seu computador pessoal (uma estação de trabalho, por exemplo) meses para ser completada.

O pesquisador, então, fica sabendo da existência de um computador dotado de *muitos* processadores no centro de computação de sua universidade. Entusiasmado ele corre para lá, pois pensa que, agora, ele vai obter suas simulações em uma fração do tempo esperado

Ao conversar com os técnicos do centro, sofre uma grande decepção: fica sabendo que, para obter uma redução factível do tempo de execução de sua aplicação, necessitará *reescrever* o programa, e mais ainda:

utilizar um outro tipo de método numérico, pois se utilizar aquele que está implementado no seu programa, há uma grande chance de que o uso daquele computador paralelo cause um *aumento (!)* do tempo de execução.

Vejam então, o paradoxo existente: temos máquinas muito rápidas, que permitem resolver um problema em uma fração do tempo anteriormente necessário. Mas, para que isso aconteça, temos de dispender um certo período de tempo - meses, até - desenvolvendo novos métodos e/ou algoritmos que sejam adequados às características do computador paralelo a ser utilizado.

É claro que, em certos casos, e tomando alguns cuidados, é possível utilizar determinados tipos de computadores paralelos com o mínimo de esforço de programação por parte do usuário. Em outros, no entanto, recairemos na árdua - porém interessantíssima - tarefa de desenvolver algoritmos paralelos.

2. Arquiteturas e programação paralelas

Hoje em dia, temos à disposição basicamente três tipos diferentes de arquiteturas paralelas disponíveis comercialmente:

2.1 *Vetoriais Paralelas*

Caracterizam-se por apresentarem um número (usualmente pequeno) de processadores com capacidade *vetorial* - i.e., a unidade aritmética é capaz de realizar uma operação sobre um conjunto de palavras ao mesmo tempo. Dentre as operações executadas dessa forma, encontram-se a *adição*, *multiplicação* e *multiplicação-por-escalar-e-adição* (ou, para os iniciados nas BLAS ("Basic Linear Algebra Subroutines"), uma _AXPY. Os registradores vetoriais permitem armazenar 64 ou 128 palavras ao mesmo tempo, usualmente, dependendo do fabricante. Os processadores componentes compartilham um barramento, o que obviamente impõe limites ao número de processadores que pode ser utilizado eficientemente, devido a congestionamentos no barramento.

Os computadores que utilizam esse tipo de arquitetura tem normalmente uma estrutura hierárquica de memória, com memórias locais a cada processador e uma memória global, acessível por qualquer processador ou, ainda, dispõe de um mecanismo de consistência de memória que permite a um processador acessar a memória local a outro processador; nesse último caso, a memória global é o conjunto formado por todas as memórias locais disponíveis.

Em termos de linguagens de programação - seja em FORTRAN (77/90/95) ou em C - temos a facilidade de se "enxergar" *toda* a memória, não sendo necessário se preocupar com o particionamento dos dados entre os processadores, tarefa realizada pelo compilador. Cabe ressaltar que os compiladores são responsáveis, na maioria das vezes, pela análise semântica do código fonte, a fim de detectar a existência de laços (DO .. END DO ou for (..);) que possam ser *vetorizados*, e gerando código de máquina apropriadamente.

De forma semelhante é feita a paralelização, e em ambos os casos, o programador pode forçar ou não a existência de código vetorizado/paralelizado. Por essa razão, dizemos que a programação nesse tipo de arquitetura tem um **paralelismo implícito**, descrito pelas relações existentes entre os dados e os comandos presentes no programa.

Note ainda que, por questões de acumulações de erro de ponto-flutuante, é possível que uma operação vetorizada não seja numericamente equivalente a mesma operação executada de forma *escalar* (ou seja, elemento-a-elemento). Felizmente, tais erros ocorrem usualmente nos limites da precisão existente no computador em uso e pode-se conviver com eles de forma razoável.

Dentre as máquinas vetoriais paralelas existentes no mercado atualmente, citamos os Cray T90, Cray J90, NEC SX-3, NEC SX-4 e Fujitsu VPP-500.

2.2 *Paralelas de memória distribuída*

São máquinas que usualmente dispõe de um número maior de processadores, se comparados aos da classe anterior, sendo cada qual equipado com uma memória local.

Nesses computadores, não existe o conceito de memória global, na medida em que um processador só pode acessar um dado residente na memória de um outro processador através de uma operação *explícita* de leitura.

São certamente mais difíceis de serem programadas, na medida em que fica a cargo do programador a *distribuição dos dados* e a *sincronização dos processos* - por isso, nessas máquinas a paralelização é **explícita**.

Um dos mecanismos de comunicação mais utilizados é o de **troca de mensagens**, onde a aplicação paralela nada mais é do que uma coleção de diferentes processos que, ao longo da sua execução, trocam dados entre si e, ao o fazerem, *sincronizam-se*. Visto por outro ângulo, pode-se dizer que os processos executam *seqüencialmente* em cada processador, à máxima velocidade possível, e só param o tempo suficiente para receber e/ou enviar um dado que necessitam ou é necessário em outro processador.

Note que, se por um lado essa arquitetura requer mais cuidado por parte de um programador, por outro ela abre uma enorme gama de possibilidades, na medida em que uma aplicação pode ser implementada utilizando diferentes programas - talvez escritos em diferentes linguagens, se necessário - os quais, quando executados, serão processos diferentes que irão cooperar entre si.

Os processadores, também chamados de *"elementos processadores"* ou de *"nós computacionais"*, são conectados entre si através de uma **rede de comunicação** que pode ser de diferentes topologias, dependendo do fabricante. Entre as mais comuns, temos a *malha bi-dimensional*, *malha tri-dimensional*, o *anel* e o *toro*. Um algoritmo que pretenda ser *eficiente* em uma determinada máquina paralela deve ser projetado de forma *a se valer da topologia da rede existente*.

Existem diferentes paradigmas de programação a serem usados aqui, referentes a forma como os processos são organizados: temos o paradigma do *mestre-e-escravos*, *dirigido-por-eventos* e o *programa-único-múltiplos-dados* (talvez o mais utilizado em aplicações científicas).

No caso de se utilizar o mecanismo de troca de mensagens, hoje em dia existem basicamente dois padrões bastante utilizados: o primeiro é o **MPI** (Message-Passing Interface), criado por um consórcio de universidades e empresas fabricantes de computadores paralelos, o qual oferece definições semânticas de como devem ser executadas operações de envio/recebimento de mensagens dos tipos *ponto-a-ponto* (entre dois processos) e *coletivas* (entre vários processadores). Existem diferentes implementações do padrão MPI, tanto em FORTRAN como em C.

O segundo é chamado de **PVM** (Parallel Virtual Machine), um padrão *"de fato"* criado por um grupo de pesquisadores liderados por J. Dongarra (U. of Tennessee at Knoxville). Também aqui encontramos operações de troca de

mensagens coletivas e ponto-a-ponto, sendo implementadas em FORTRAN 77 e C.

Como exemplos de tal arquitetura, citamos o IBM SP2, SGI Power Challenge e Intel Paragon.

2.3 *Paralelas de memória distribuída com endereçamento global à memória*

Surgiram recentemente, de forma a tentar agregar em uma única arquitetura os melhores aspectos das duas arquiteturas descritas anteriormente. Do ponto de vista do *hardware*, é consideravelmente mais simples montar um computador com arquitetura de memória distribuída do que um de memória compartilhada como os de arquitetura vetorial paralela, o que se traduz em um custo final menor. Mas, quanto à utilização de computadores paralelos, é mais fácil de programá-los se for possível tratar a memória como um todo.

Daí então essa arquitetura, que tem como exemplos o Cray T3E, HP/Convex Exemplar e o (hoje defunto) KSR-1. Analisando-se as diferentes soluções propostas nesses computadores, vemos que todas baseiam-se no uso de duas idéias básicas:

- uso de processadores disponíveis no mercado, i.e., não são fabricados para este fim específico (*"commodity processor"*)
- utilização da própria rede de interconexão - já existente para fins de troca de dados entre os processadores - acoplada a um mecanismo de consistência dos dados residentes nas memórias específicas a cada processador.

Assim, temos no Cray T3E uma conexão em toro (cada *nó computacional*, dotado de dois processadores DEC AXP EV5, conecta-se com outros seis vizinhos); no HP/Convex Exemplar, temos uma rede hierárquica em árvore, com vários anéis conectando os elementos processadores.

Quanto à programação, temos os dois modelos básicos - *compartilhamento de memória* e *memória distribuída* - o que certamente oferece maior flexibilidade para o desenvolvimento de aplicações. Com recursos desse tipo à disposição, dizemos que o paralelismo é **semi-implícito**, na medida em que o particionamento dos dados é especificado pelo programador; porém, os laços são paralelizados automaticamente pelo compilador - podendo ainda utilizar rotinas (normalmente *proprietárias*, portanto não portáveis entre computadores de diferentes fabricantes), que permitem a leitura e/ou escrita *remotas*, i.e. em variáveis residentes em outros processadores.

Particularmente, no Cray T3E é possível escrever um programa dessa forma - quase como se fosse através de troca de mensagens.

3. Estudo de caso

A fim de expor as diferenças existentes entre programas escritos para cada uma das arquiteturas expostas na seção anterior, utilizaremos como "corpo de teste" um algoritmo que calcula o produto externo entre dois vetores u e v, $A=uv^T$, onde, para fins de explanação, A é uma matriz de ordem n e u e v são vetores. Essa operação surge, por exemplo, em cálculos ligados à minimização de funções (mais particularmente, na correção de uma matriz Hessiana a partir da correção efetuada em uma estimativa para o vetor solução). Os elementos de A são obtidos como

$$A_{i,j} = u_i * v_j, \quad i=1,...,n, \quad j=1,...,n.$$

Utilizando a linguagem FORTRAN 77 e as rotinas BLAS, relembramos que é mais eficiente acessar a matriz através de suas colunas, o que nos leva ao seguinte programa *seqüencial*:

Programa 3.1 - Cálculo seqüencial do produto-externo

```
            PROGRAM PRODEXT

      *  Declarações
            INTEGER J,N
            PARAMETER (N=1000)
            REAL A(N,N),U(N),V(N)

      *  Inicialização dos dados
            ...
      *  Cálculo do produto-externo
            DO J=1,N

      *  1. Copia U para a J-ésima coluna de A
                  CALL SCOPY(N,U,1,A(1,J),1)

      *  2. Multiplica U por V(J)
                  CALL SSCAL(N,V(J),A(1,J),1)

            END DO

            STOP
            END
```

A seguir, veremos como podemos programar essa operação nas três diferentes arquiteturas em questão.

3.1 *Computadores Paralelos Vetoriais*

Em computadores desse tipo - Cray Y-MP, Cray T90, NEC SX-3, Fujitsu VPP-500, por exemplo - podemos obter uma *execução* paralela do programa 3.1 de maneira simples: basta compilar o código-fonte com o compilador auto-vetorizador-paralelizador disponível. O compilador analisará o código-fonte e determinará as dependências de dados existentes e quais os laços (**DO .. END DO**) candidatos a serem vetorizados e paralelizados.

No caso específico do programa 3.1, infelizmente, ao submetê-lo ao compilador, obteremos um código-objeto que alcançará uma baixa performance, pois só existem chamadas de sub-rotinas dentro do laço **DO J=1,N .. END DO.** O compilador não irá vetorizar esse código nem tampouco paralelizá-lo.

Nesse caso, temos duas escolhas a tomar: podemos solicitar ao compilador que ele proceda a um *"inlining"* das rotinas **SCOPY** e **SSCAL**, o que fará com que o código daquelas rotinas seja incluído diretamente no nosso código-fonte, substituindo as chamadas àquelas rotinas. Isso seria equivalente a termos o seguinte código-fonte:

Programa 3.2 - Cálculo do produto-externo com "Inlining" das rotinas SCOPY e SSCAL

```
      PROGRAM PRODEXT

*     Declarações
      INTEGER I,J,N
      PARAMETER (N=1000)
      REAL A(N,N),U(N),V(N)

*     Inicialização dos dados
      ...
*     Cálculo do produto-externo
      DO J=1,N

*     1. Copia U para a J-ésima coluna de A
            DO I=1,N
                  A(I,J) = U(I)
            END DO

*     2. Multiplica U por V(J)
            DO I=1,N
                  A(I,J) = A(I,J)*V(J)
            END DO

      END DO

      STOP
      END
```

Ora, é evidente que, se por um lado essa é a opção que menos esforço exige de nossa parte, ela leva a um desperdício de operações, incluindo a

movimentação desnecessária de dados da memória para o processador e vice-versa (a cópia de *u* para uma *j*-ésima coluna de *A*, e a leitura/escrita de posições de *A* no segundo laço **DO I=1,N .. END DO**).

Em nosso caso, é mais vantajoso procedermos a uma otimização manual do código-fonte, combinando os laços em I em um só; assim, teríamos o seguinte programa:

Programa 3.3 - Cálculo do produto-externo (otimizado manualmente).

```
            PROGRAM PRODEXT

      *  Declarações
            INTEGER I,J,N
            PARAMETER (N=1000)
            REAL A(N,N),U(N),V(N)

      *  Inicialização dos dados
            ...
      *  Cálculo do produto-externo
            DO J=1,N

      *  1. Multiplica U(I) por V(J)
                  DO I=1,N
                        A(I,J) = U(I)*V(J)
                  END DO

            END DO

            STOP
            END
```

Agora, temos um programa cujo laço interno em I pode ser vetorizado e paralelizado, pois não existem dependências de dados entre os elementos envolvidos nos laços interno e externo. Se o valor de N for grande o suficiente, de forma a manter com uma alta taxa de utilização os registradores vetoriais dos processadores - o que depende do computador em uso - esse programa apresentará uma boa eficiência.

3.2 *Computadores paralelos de memória-distribuída*

A utilização eficiente de computadores desse tipo requer uma intervenção mais profunda do programador, exigindo muitas vezes um *repensar* do algoritmo em questão. Não basta aqui simplesmente efetuar algumas alterações no programa, pois é bem possível que um programa altamente otimizado para execução seqüencial ofereça um desempenho sofrível nesses computadores.

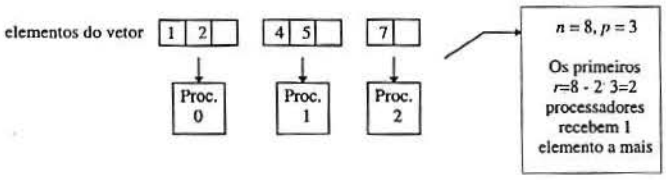
Evidentemente, grandes esforços tem sido feitos para se reduzir ao máximo possível essa intervenção do programador, pois ela demanda um bom

tempo de desenvolvimento, desde a criação de um algoritmo adequado até a otimização final do programa paralelo.

Nesse sentido, temos hoje em dia a linguagem *High-Performance Fortran* (HPF), a qual oferece um conjunto de diretivas de programação (a serem embutidas em um código-fonte por um programador) que permite informar ao compilador como um determinado dado deve ser particionado entre os processadores, como os laços devem ser paralelizados, etc. Em certos tipos de aplicações, onde o padrão de comunicação entre os processadores pode ser dito *regular* (por exemplo, a solução de equações diferenciais parciais por métodos explícitos envolvendo diferenças-finitas ou a solução de um sistema linear de equações por um método direto), é possível obter uma boa paralelização utilizando HPF. Em outros tipos de aplicações, com padrões de comunicação irregulares, a paralelização não será boa. Além disso, em ambos os casos, é bem possível que seja necessário utilizar mecanismos de troca de mensagens entre processos cuja rapidez seja crítica para a obtenção dos resultados desejados.

Em nosso estudo de caso, vamos criar um algoritmo paralelo utilizando troca de mensagens. Para tanto, é necessário que se determine inicialmente quais são os dados a serem tratados. No cálculo do produto-externo, temos os dois u e v e a matriz A; suponhamos que temos à disposição p processadores e que tanto os vetores como a matriz serão particionados entre os processadores.

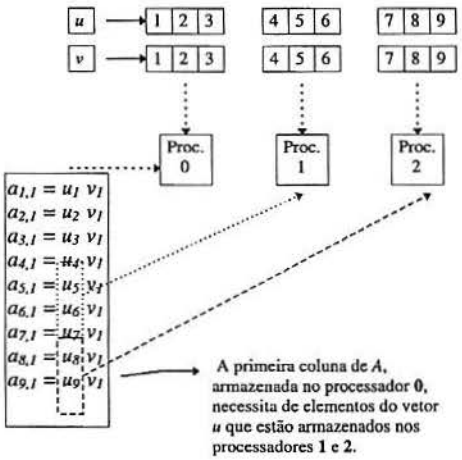
No caso dos vetores (lembrando que ambos tem n elementos), vamos distribuir inicialmente, entre cada processador, um conjunto de n/p elementos contíguos. Se n não for um múltiplo inteiro de p, então os primeiros $r = n - \lfloor n/p \rfloor p$ processadores receberão um elemento a mais, cf. o diagrama abaixo:



Com isso, alcançamos algo que é crítico: o *balanceamento de carga* deve ser tal que as tarefas a serem realizadas pelos processadores consumam aproximadamente o mesmo tempo. Em nosso caso, como a mesma tarefa deve ser executada em todos os processadores - i.e. calcular o produto-externo - então se mantivermos um volume de dados aproximadamente igual entre os processadores, obteremos um balanceamento de carga (note que se um determinado processador tivesse de realizar, por exemplo, o dobro das operações que os demais, então esses devem operar sobre o dobro de elementos do que aquele).

O particionamento dos vetores leva a um particionamento das matrizes. Como em Fortran 77 é mais eficiente operarmos sobre as colunas de uma matriz, optamos por um particionamento da mesma por *colunas contíguas*. Assim, cada processador armazenará no máximo *n/p+1* colunas da matriz, de forma semelhante ao particionamento mostrado acima.

Retornando então ao nosso problema, vemos que cada elemento $A_{i,j}$ é resultado do produto entre os elementos u_i e v_j - e é óbvio, pelo particionamento imposto sobre os vetores e a matriz, que alguns elementos de u e v, cujo produto deve ser armazenado em um processador q, na verdade residem em processadores r e s. Vejamos o diagrama a seguir:



No exemplo acima, temos o cálculo da primeira coluna da matriz A; veja que, pela formulação do produto-externo, a i-ésima coluna de A é o vetor u multiplicado pelo i-ésimo elemento de v. Infelizmente, ao distribuirmos os dados entre os processadores, estamos fazendo com que haja comunicação entre os processadores, de forma a transmitir os dados necessários a um processador específico.

Mais ainda, esse exemplo de operação exige, com o particionamento de dados escolhido, que cada processador envie para todos os demais processadores os elementos de u nele armazenados. Esse é o pior caso possível,

onde temos um padrão de comunicação do tipo *todos-para-todos*, que consome um tempo razoável para ser efetuada e que, se o fabricante do computador ou da biblioteca de rotinas de troca de mensagens não tiver tomado extremo cuidado em sua implementação, apresentará um desempenho sofrível - quase que efetivamente removendo quaisquer ganhos que teríamos ganho pela divisão das tarefas entre os processadores - em nosso caso, as multiplicações entre os elementos de *u* e *v*.

Certamente, cabe a pergunta: por que então não deixar com que cada processador contenha cópias inteiras (i.e. todos os elementos) de *u* e *v*? Essa estratégia remove o problema de comunicação, mas esbarramos aí em outro problema - temos memória disponível para gastar? Na maioria das vezes a resposta será não, daí porque termos de distribuir todos os dados entre os processadores.

Com base no exposto acima, podemos então apresentar um algoritmo paralelo, para troca de mensagens, para o cálculo do produto-externo conforme segue:

Algoritmo 1. Cálculo paralelo do produto-externo através de troca de mensagens

```
1. "irradia ("broadcast") os elementos de u armazenados localmente para os
     demais p-1 processadores"
2. for j=1,n/p
3.    k=primeiro(q)
4.    for i=1,n/p
5.          A_k,j=u_i v_j
6.          k=k+1
7.    endfor
8. endfor
9. for t=0,p-1
10.   if t ≠ meupid() then
11.         "recebe de um processador q um conjunto de elementos de u,
            armazenando em r"
12.         for j=1,n/p
13.               k=primeiro(q)
14.               for i=1,nels(q)
15.                     A_k,j=r_i v_j
16.                     k=k+1
17.               endfor
18.         endfor
19.   endif
20. endfor
```

O Algoritmo 1 está expresso segundo o modelo **SPMD**, ou seja, um único código é executado em todos os processadores; porém, como cada processador armazena elementos diferentes de *u* e *v*, cada uma das cópias desse código opera sobre um conjunto diferente de dados.

As rotinas primeiro, nels e meupid retornam, respectivamente:

- o índice do primeiro elemento de *u*, *v* e da primeira coluna de *A*,
- o número de elementos armazenados localmente a um dado processador *q*,
- o número de identificação do processador (*0,1,...,p-1*).

Podemos destacar duas seções desse código: os passos 1 a 8 operam exclusivamente sobre dados armazenados localmente, e os demais passos contém operações que envolvem dados armazenados remotamente (i.e em outros processadores). Chamemos a primeira de seção *local* e a segunda de *remota*.

Temos, é claro, dois tipos de ações a serem efetuadas nesse algoritmo: calcular e comunicar. Como organizá-las é de suma importância para se maximizar o desempenho de sua execução real em um computador paralelo.

Vemos que, na seção *local*, é feita inicialmente uma *irradiação* dos elementos armazenados localmente do vetor *u* para todos os demais processadores, e após realiza-se o cálculo de alguns elementos de *A*, apenas com os elementos de *u* e *v* armazenados localmente.

Nada impede que tivéssemos invertido essa ordem de operações - comunica-calcula - para calcula-comunica. Qual seria o resultado dessa modificação?

Com a tecnologia atualmente existente, medimos os tempos de comunicação entre redes de processadores em termos de *milissegundos* ou *microssegundos* (dependendo da tecnologia empregada), ao passo que as operações aritméticas e acesso à memória podem ser medidas em termos de *nanossegundos*. Se organizarmos as operações como calcula-comunica, isso levará a um certo desperdício de tempo, de vez que, quando um processador estiver executando o passo 11 do algoritmo - o recebimento dos elementos de *u* enviados por um outro processador - possivelmente esses dados ainda não terão chegado.

No entanto, se usarmos a forma comunica-calcula, estaremos dando mais tempo para que os dados transmitidos cheguem aos seus processadores-destino de forma que quando um processador precisar operar com um dado remoto, ele já esteja disponível (possivelmente armazenado em um *"buffer"* da máquina).

É claro que toda essa discussão depende das relações existentes entre o tempo de transmissão entre processadores, velocidade de execução das

operações aritméticas, tempo de acesso à memória e quais os valores de n e p para os quais o tempo gasto na seção *local* é suficientemente grande para mascarar o tempo de irradiação dos dados locais.

Apresentamos, a seguir, uma implementação em FORTRAN 77 do Algoritmo 1, usando rotinas de troca de mensagens do padrão MPI:

Programa 3.4 Cálculo do produto-externo usando troca de mensagens via MPI

```
      SUBROUTINE OUPROD(NPROCS,LDA,N, U,V,R,A)
      INCLUDE 'mpif.h'
      INTEGER LDA,N,NPROCS
      REAL A(LDA,*),R(*),U(*),V(*)
      INTEGER I,IERR,IP,J,K,WHO,MYN
      EXTERNAL MPI_BCAST,PRIMEIRO,NELS,MEUPID

      MYN = NELS(MEUPID())
      DO 30 IP = 0,NPROCS - 1
        IF (IP.EQ.MEUPID()) THEN
* Irradia meus elementos de U para os demais processadores
          CALL MPI_BCAST(U,MYN,MPI_REAL,MYID,MPI_COMM_WORLD,IERR)
* Calcula com dados locais
          DO 10 J = 1,MYN
            K = PRIMEIRO(IP)
            DO 20 I = 1,MYN
              A(K,J) = U(I)*V(J)
              K = K + 1
 20         CONTINUE
 10       CONTINUE
        ELSE
* Recebe elementos de U armazenados em WHO
          WHO = IP
          CALL MPI_BCAST(R,NELS(WHO),MPI_REAL,WHO,MPI_COMM_WORLD,IERR)
* Calcula com os elementos de U vindos de WHO, armazenados em R
        DO 40 J = 1,MYN
            K = PRIMEIRO(WHO)
            DO 50 I = 1,NELS(WHO)
              A(K,J) = R(I)*V(J)
              K = K + 1
 50         CONTINUE
 40       CONTINUE
        END IF
 30   CONTINUE

      RETURN
      END
```

As linhas marcadas em negrito indicam as operações de troca de mensagens. Devido à definição da função *"broadcast"* em MPI, todos os processadores que pertencem a um determinado grupo (em nosso caso, todos eles, conforme especificado pela constante MPI **MPI_COMM_WORLD**) devem chamar a rotina **MPI_BCAST**.

Cabe aqui uma breve explanação dos parâmetros dessa rotina:

MPI_BCAST(DADO,TAMANHO,TIPO,PROC_ID,COMUNICADOR,RESULTADO)

onde

- **DADO**: a variável simples ou arranjo a ser transmitido,
- **TAMANHO**: o número de palavras a serem transmitidas; se for uma variável simples, é uma unidade, se for um arranjo, é o número de elementos do arranjo (seja um vetor ou matriz),
- **TIPO**: informa qual o tipo da variável, por exemplo, **MPI_REAL**,
- **PROC_ID**: o identificador do processo dentro do conjunto indicado pelo **COMUNICADOR**. Em MPI, é possível que um processo pertença a diferentes conjuntos de processos, cada conjunto sendo especificado por um *comunicador*; assim, um mesmo processo pode ter diferentes identificadores, cada um no contexto de um comunicador específico. O identificador **MPI_COMM_WORLD** refere-se ao conjunto de todos os processos ativados por uma aplicação.
- **RESULTADO**: é uma variável inteira que contém o resultado da operação.

Para maiores detalhes ver [8].

De forma genérica, podemos dizer que uma operação de irradiação ou *"broadcast"* envolvendo p processadores tem um tempo de execução proporcional a $log_2 p\ T_c\ (n/p)$, onde T_c é o tempo de comunicação entre dois processadores para se enviar n/p palavras. Já os passos de 2 a 8 envolvem um custo proporcional a $(n/p)^2 O\cdot$, onde $O\cdot$ é o custo de uma multiplicação entre dois elementos de um vetor (i.e., esse tempo envolve o acesso à memória). Assim, podemos ter uma idéia preliminar de para quais valores de n e p ocorrerá um mascaramento da comunicação com a computação, permitindo um desempenho bastante eficiente.

3.3 Computadores paralelos de memória distribuída com endereçamento global

Os mecanismos de *leitura/escrita remota* - i.e. como um processo sendo executado em um processador pode ler/escrever dados armazenados em outro processador - ainda são proprietários, na medida em que cada fabricante oferece as suas rotinas, não existindo ainda um padrão para tais mecanismos. Dessa forma, aqui apresentaremos uma implementação do Algoritmo 1 específica para um computador Cray T3E.

A principal diferença, se comparado a uma máquina de memória distribuída sem endereçamento global é que, quando se ativam os processos - i.e. p cópias do mesmo código - os dados são gerados e/ou lidos por cada processador e a partir de então tornam-se potencialmente disponíveis aos demais processos. Ao programador cabe inserir diretivas de programação que informam ao compilador quais dados devem ser disponibilizados.

Quanto à programação, no entanto, há uma outra diferença: nos computadores de memória distribuída *sem endereçamento global*, o processador que armazena um dado deve tomar a iniciativa de enviá-lo para os demais. Já naqueles *com endereçamento global*, o processador que deseja um dado é que toma a iniciativa de lê-lo. Podemos sumarizar então:

- *troca de mensagens*: cooperação entre os processadores, existe sempre um par envia-recebe (podendo tomar várias formas, *um-para-um, um-para-muitos, um-para-todos,* etc.);
- *leitura/escrita remota*: cada processador age individualmente, sem a necessidade de intervenção de outro processador para que a operação seja completada.

Salientamos que esse é um modo de programação *híbrido,* na medida em que é deixado ao compilador a tarefa de distribuir os dados entre os processadores, de acordo com as diretivas de programação especificadas pelo programador; mas a transferência de dados entre os processadores é feita de forma *explícita*, através das operações de leitura/escrita remotas que o programador colocou no programa.

Retornando ao nosso estudo de caso, podemos notar que no Algoritmo 1 não há a necessidade de *escrita* remota, mas apenas de *leitura.* O passo 1 do algoritmo deve ser desconsiderado e o passo 11 deve ser substituído por uma operação de leitura remota, a qual tem o nome de **SHMEM_GET** no ambiente de programação do Cray T3E.

Uma possível implementação do Algoritmo 1 para esse ambiente pode então ser como segue:

Programa 3.5 - Cálculo do produto-externo em máquinas de memória distribuída com endereçamento global

```
      SUBROUTINE OUPROD(NPROCS,N,V,A)
      PARAMETER (N$PES=512)
      INTEGER LDA
      PARAMETER (LDA=1024)
      INTEGER LOCLEN
      PARAMETER (LOCLEN=LDA/N$PES)
      INTEGER MYID,MYN,N,NPROCS
      REAL A(LDA,*),V(*)
      REAL R(LOCLEN),U(LOCLEN)
      COMMON /COMPAR/U,R
CDIR$ PE_PRIVATE A, U, V, R
      INTEGER I,IRSLT,IP,J,K,WHO
      INTEGER SHMEM_GET
      EXTERNAL SHMEM_GET,MEUPID,PRIMEIRO,NELS

      MYN = NELS(MEUPID())
      DO 30 IP = 0,NPROCS - 1
        IF (IP.EQ.MEUPID()) THEN
* Calcula com dados locais
          DO 10 J = 1,MYN
            K = PRIMEIRO(IP)
            DO 20 I = 1,MYN
              A(K,J) = U(I)*V(J)
              K = K + 1
 20         CONTINUE
 10       CONTINUE
        ELSE
* Recebe elementos de U armazenados em WHO
          WHO = IP
          IRSLT = SHMEM_GET(R,U,NELS(WHO),WHO)
* Calcula com os elementos de U vindos de WHO, armazenados em R
          DO 40 J = 1,MYN
            K = PRIMEIRO(WHO)
            DO 50 I = 1,NELS(WHO)
              A(K,J) = R(I)*V(J)
              K = K + 1
 50         CONTINUE
 40       CONTINUE
        END IF
 30   CONTINUE
      RETURN
      END
```

As linhas em negrito indicam a declaração de quais variáveis são *locais* a cada processador, bem como a operação de leitura remota. Quando dos testes realizados no Cray T3E, a versão do compilador exigia que as variáveis utilizadas na rotina **SHMEM_GET** pertencessem a um bloco COMMON; aparentemente tal restrição foi removida em versões posteriores do compilador.

4. Considerações finais

Apresentamos nesse trabalho uma breve introdução aos problemas existentes ao se desenvolver algoritmos paralelos, bem como as diferentes estratégias que devem ser tomadas, dependendo da arquitetura paralela a ser utilizada.

Como vimos, é necessário um certo esforço por parte do programador para se utilizar determinados tipos de arquiteturas, envolvendo desde simples otimizações a nível de código-fonte, até mesmo reestruturações dos algoritmos utilizados ou, idealmente, o desenvolvimento de novos algoritmos adequados ao computador em questão.

Acreditamos que é necessário que se dedique tempo ao desenvolvimento de ferramentas de software que facilitem por sua vez o desenvolvimento de aplicações paralelas, incluindo compiladores e analisadores estáticos e dinâmicos de desempenho.

No futuro (não tão distante), já vislumbramos a utilização de computadores óticos, com uma velocidade de operação ordens de magnitude mais rápida do que os mais potentes computadores hoje existentes. Se, por um lado, para muitos problemas essa velocidade será suficientemente grande que não necessitará o uso de programação paralela (a nível de aplicação), sabemos que as necessidades da humanidade sempre ultrapassarão os recursos disponíveis. Acreditamos, portanto, que o uso de arquiteturas e programação paralelas permanecerá sendo de grande valia.

5. Bibliografia

1. Akl, S.G. *The Design and Analysis of Parallel Algorithms.* Prentice-Hall, Englewood Cliffs, 1989.

2. Bertsekas, D.P., Tsitsiklis, J.N. *Parallel , Distributed Computation -- Numerical Methods.* Prentice-Hall International, Englewood Cliffs, 1989.

3. de Carlini, U. , Villano, U. *Transputers , parallel architectures -- message-passing distributed systems.* Ellis Horwood, Chichester, 1991.

4. Hwang, K. , Briggs, F.A. *Computer Architecture and Parallel Processing.* McGraw-Hill, New York, 1984.

5. Jamieson, L.H. , Gannon, D. , Douglass, R.J. *The Characteristics of Parallel Algorithms.* MIT Press, Cambridge, Massachusetts, 1987.

6. Miklosko, J. "Fast algorithms , their implementation on specialized parallel computers". In: *Special Topics in Supercomputing*, V. 5, Ch. 3,. Publishing House of the Slovak Academy of Sciences, North-Holland, Amsterdam, 1989.

7. Modi, J.J. *Parallel Algorithms and Matrix Computation.* Oxford University Press, Oxford, 1988.

8. Message Passing Interface Forum. *"MPI: A message-passing interface standard"*, TR CS-93-214, University of Tennessee, November 1993.

9. Beguelin, A., Dongarra, J., Geist, A., Manchek, R., Sunderam, V.S. *"A user's guide to PVM -- Parallel Virtual Machine"*, Research Report ORNL/TM-11826, Oak Ridge National Laboratory,1992.

10. Schendel, U. *Introduction to Numerical Methods for Parallel Computers.* Ellis Horwood, Chichester, 1984.