# PROGRAMAÇÃO PARALELA USANDO MPI

**Maurílio Boaventura**

**DCCE/IBILCE/UNESP**

**São José do Rio Preto - SP**

Nos computadores convencionais, um programa é um conjunto de instruções que são transmitidas à unidade de processamento para serem executados de forma sequencial. Esses computadores em geral possuem um único processador e mesmo possuindo mais que um, executam somente programas sequenciais.

O conceito de programação paralela está diretamenta relacionado ao uso de computadores paralelos, ou seja, computadores dotados de várias unidades de processamento, com capacidade de executar programas paralelos.

No caso de programas paralelos temos instruções sendo executadas em diferentes processadores ao mesmo tempo. Além disso, há geração e troca de informação entre os processadores. Tais processos, em geral, apresentam grandes dificuldades de serem implementados.

Diante de tais dificuldades poderíamos questionar se a utilização de algoritmos paralelos é compensadora visto que o resultado final é equivalente ao do seqüencial. Porém, notemos os principais objetivos da programação paralela são os seguintes:

**Redução do tempo de execução:**

se um programa sequencial precisa de *t* unidades de tempo para executar em um único processador então, idealmente um programa paralelo equivalente completaria a tarefa em *t/p* unidades de tempo quando executado em *p* processadores.

**Aumento da quantidade de memória:**

se um programa precisa de *w* palavras de memória e cada processador tem *w/p* palavras disponíveis então *p* processadores ofereceriam conjuntamente a capacidade necessária.

# MODELOS DE PROGRAMAÇÃO PARALELA

Memória Compartilhada: diferentes processadores acessam uma mesma memória compartilhando os dados entre si.

Memória Distribuída: cada processador possui sua própria memória fazendo-se com que dados devam ser trocados entre si por mecanismos específicos de troca.

No modelo de memória distribuída um dos mecanismos de comunicação mais comum é o de troca de mensagens, que consiste em processos sequenciais trocando informações entre si.

Assim, os processadores devem ser sincronizados para receberem e/ou enviarem mensagens.

Essas mensagens podem ser síncronas ou assíncronas:

**Síncrona:** comunicação na qual o processador que envia a mensagem aguarda, do destinatário, o recebimento de um sinal confirmando o recebimento da mensagem.

**Assíncrona:** comunicação na qual o processo que envia a mensagem não espera que haja um sinal de recebimento dessa mensagem pelo destinatário.

Tais mensagens podem ainda ser de um dos três tipos:

Um - para - Um: um processo enviando mensagem para outro.

Um - para - Muitos: um processo enviando mensagem para outros, não todos.

Um - para - Todos: um processo enviando mensagem para todos os outros.

Para realizar a comunicação entre os processos existem vários “softwares” comunicadores entre os quais podemos destacar o PVM (“Parallel Virtual Machine”) e o MPI ( “Message Passing Interface”). Com a utilização desses “softwares”, que são de domínio público, podemos implementar nas linguagens FORTRAN ou C.

Temos como principais modelos de algoritmos paralelos utilizando o ambiente PVM, o modelo mestre -escravo e o SPMD (“Single-Program, Multiple-Data”).

No ambiente MPI os programas paralelos devem ser desenvolvidos segundo o modelo SPMD, podendo-se simular o modelo mestre escravo sobre o SPMD.

O mestre escravo, pode ser utilizado quando existem, no mínimo, dois tipos bem definidos de uma mesma aplicação, um deles chamado de “mestre” e os outros de “escravos”. A cada um deles será atribuído um processo diferente.

A computação paralela engloba uma grande variedade de aplicações tanto no campo científico (Física, Química, Matemática, Engenharia, Medicina), como na área comercial (Computação gráfica, sistemas de bancos de dados) ou mesmo gerencial.

O grande desafio para os matemáticos interessados na programação paralela é além, de gerar algoritmos paralelos, implementá-los de forma eficiente.

# EXEMPLOS DE ALGORITMOS E PROGRAMAS PARALELOS

O primeiro exemplo é um somatório onde queremos somar os números consecutivos de 1 à 1.000.000. Assim, temos:

$$\sum_{i=1}^{10^6} i = 1 + 2 + 3 + \ldots + 10^6 = S$$

A paralelização desse somatório para, por exemplo, 4 processadores (resultado análogo para *n* processadores) seria:

$$proc.1: \sum_{i=1}^{250.000} i = S_1 \quad proc.2: \sum_{i=250.001}^{500.000} i = S_2$$

$$proc.3: \sum_{i=500.001}^{750.000} i = S_3 \quad proc.4: \sum_{i=750.001}^{1.000.000} i = S_4$$

Ou seja, devemos distribuir a soma nos 4 processadores de forma que após o término de cada soma parcial a soma total seja efetuada em um dos processadores para a finalização do programa.

Algumas considerações devem ser feitas em relação a comandos específicos da linguagem FORTRAN relacionados ao software MPI.

**1) MPI_INIT(Erro):** rotina que inicializa o MPI. Têm como único parâmetro, a variável ERRO que é inteira e que associa um número devolvido em caso de erro de execução.

**2) MPI_COMM_SIZE(MPI_COMM_WORLD,Nproc, Erro):** rotina que devolve o número de processadores utilizados tendo como parâmetros:

*MPI_COMM_WORLD:* é uma definição de qual biblioteca padrão está sendo usado pelo MPI.

*Nproc:* variável inteira que associa o número de processadores.

*Erro:* variável inteira que associa um número devolvido em caso de erro de execução.

**3) MPI_COMM_RANK(MPI_COMM_WORLD,MyID, Erro):** rotina que devolve a identificação do processador atual utilizado, tendo como parâmetros:

*MPI_COMM_WORLD:* é uma definição de qual biblioteca padrão está sendo usado pelo MPI.

*MyID:* variável inteira que associa o número do processador. Esse número varia de *0* à *p-1* igual ao total de processadores menos 1.

*Erro:* variável inteira que associa um número devolvido em caso de erro de execução.

**4) MPI_SEND(Variable,QTD, MPI_TYPE,ID,MsgType,MPI_COMM_WORLD,Erro):** rotina utilizada para se enviar uma mensagem do processador atual para outro. Possui como parâmetros:

*Variable:* é a mensagem a ser enviada. Aqui generalizamos com o nome VARIABLE sendo que esta variável pode ser de diversos tipos (INTEGER, REAL, DOUBLE,CHAR,etc....)

*QTD:* Quantidade de variáveis que estão sendo enviadas.

*MPI_TYPE:* é o tipo de dado que esta sendo enviado. Pode ser de vários tipos e deve ser compatível com a variável enviada.

*ID:* é a especificação de para qual processador deve ser enviado.

*MsgType:* é um rótulo usado para a identificação de qual mensagem esta sendo enviada.

*MPI_COMM_WORLD:* é uma definição de qual biblioteca padrão está sendo usado pelo MPI.

*Erro:* variável inteira que associa um número devolvido em caso de erro de execução.

**5) MPI_RECV(Variable,QTD, MPI_TYPE,ID,MsgType,MPI_COMM_WORLD,Erro):** rotina utilizada para receber uma mensagem por um outro processador. Possui como parâmetros:

*Variable:* é a mensagem a ser enviada. Aqui generalizamos com o nome VARIABLE sendo que esta variável pode ser de diversos tipos (INTEGER, REAL, DOUBLE,CHAR,etc....)

*QTD:* Quantidade de variáveis que estão sendo enviadas.

*MPI_TYPE:* é o tipo de dado que esta sendo enviado. Pode ser de vários tipos e deve ser compatível com a variável enviada.

*ID:* é a especificação de para qual processador deve ser enviado.

*MsgType:* é um rótulo usado para a identificação de qual mensagem esta sendo enviada.

*MPI_COMM_WORLD:* é uma definição de qual biblioteca padrão está sendo usado pelo MPI.

*Erro:* variável inteira que associa um número devolvido em caso de erro de execução.

**6) MPI_REDUCE(SBUF, RBUF, Nproc,MPI_TYPE,OP, ID,MPI_COMM_WORLD, Erro):** rotina de computação global. Neste tipo de operação coletiva, o resultado parcial de cada processo em um grupo é combinado e retornado para um específico processo, utilizando-se algum tipo de função de operação (máximo, mínimo, soma, produto).

*SBUF:* endereço do dado que fará parte da operação de redução ("send buffer")

*RBUF:* endereço do dado que receberá o resultado da operação de redução.

**7) MPI_FINALIZE(Erro):** rotina que finaliza o MPI. Têm como único parâmetro a variável ERRO que é inteira e que associa um número devolvido em caso de erro de execução.

# Somatório dos Números Inteiros

**Código Sequencial:**

```
Integer i,S
S=0
Do i=1,1000000
        S=S+i
End Do
Write(*,*) ‘ A soma total é: ‘,S
Stop
End
```

## Código Paralelo:

```
INCLUDE 'mpi.h'
Integer i, j, STATUS(MPI_STATUS_SIZE),MsgType, Erro, Nproc,
MyID, K, S, S1
Call MPI_INIT(Erro)
Call MPI_COMM_SIZE(MPI_COMM_WORLD,Nproc, Erro)
Call MPI_COMM_RANK(MPI_COMM_WORLD, MyID,Erro)
S=0
k=1000000/Nproc
Do i=(MyID*K)+1,(MyID+1)*K
        S=S+i
End Do
```

```
Call MPI_REDUCE(S,S1,Nproc,MPI_INTEGER,MPI_SUM, 0,
     $                    MPI_COMM_WORLD,Erro)
If(MyID.EQ.O)then
       Write(*,*)' A soma total: ', S1
EndIf
Call MPI_FINALIZE(Erro)
```

```
If(MyID.NE.0) then
        MsgType=100+MyID
        Call MPI_SEND(S,1,MPI_INTEGER,0,MsgType,
$                                   MPI_COMM_WORLD,Erro)
Else
        Do i=1,Nproc-1
        MsgType=100+i
        Call MPI_RECV(S1,1,MPI_INTEGER,MyID+i,
$               MsgType,MPI_COMM_WORLD,STATUS,Erro)
        S=S+S1
        EndDo
Endif
```

O “loop” seguinte é talvez a parte mais importante do programa. Detalhemos as variáveis de controle utilizadas. O “loop” varia de *i=(MyID*k)+1 à (MyID + 1)*k.* Vejamos o que cada processador executa tomando, por exemplo, o caso de *4* processadores. Lembrando que a numeração de processadores começa de *0* temos:

$$proc.0 \Rightarrow i = (0 * 250.000) + 1 (0 + 1) * 250.000 \Rightarrow i = 1, 250.000$$

$$proc.1 \Rightarrow i = (1 * 250.000) + 1 (1 + 1) * 250.000 \Rightarrow i = 250.001, 500.000$$

$$proc.2 \Rightarrow i = (2 * 250.000) + 1 (2 + 1) * 250.000 \Rightarrow i = 500.001, 750.000$$

$$proc.3 \Rightarrow i = (3 * 250.000) + 1 (3 + 1) * 250.000 \Rightarrow i = 750.001, 1.000.000$$

Observe que neste modelo de programação paralela, um mesmo código fonte é executado em cada processador. Assim, cada variável *S* é distinta das "demais" relativamente a cada processador. É o que acontece com o "loop". Cada processador "enxergará" somente a parte que lhe compete. No caso da variável *K*, essa não muda o seu valor durante a execução do programa, logo todos os processadores assumirão o mesmo valor.

Outras observações sobre o programa:

1) Dependendo do número de processadores envolvidos no sistema, a variável *k* poderia assumir valores não inteiros. Em alguns programas isso não influenciaria, porém, nesse, caso, como estamos trabalhando com inteiros, devemos prever tal acontecimento.

2) Ao receber as mensagens dos demais processadores, o processador *0* deve receber tais valores na variável *S1* pois como já mencionado a variável *S* é distinta em processadores distintos.

3) Escolhemos o processador *0*, para receber os valores parciais dos demais processadores por uma questão de conveniência, porém, nada nos impede de acumulá-los em outro qualquer.

# ANÁLISE DA PERFORMANCE

Para avaliar a eficiência de um programa paralelo, devemos levar em consideração dois importantes fatores:

1) O tempo de comunicação que é o tempo necessário para a troca de mensagens entre os processos;

2) O tempo de sincronização, que é o tempo de espera desses processos até que determinadas tarefas sejam concluídas, a fim de dar continuidade às suas execuções.

As medidas mais aceitas para avaliar o desempenho de programas paralelos são a eficiência e o "speed-up". A eficiência é uma medida de quão bem o algoritmo paralelo utiliza o tempo nos vários processadores, e é dada pela seguinte expressão:

$$E(p,n) = \frac{T_s}{pT_p}$$

Onde,

$p$ é o número de processadores envolvidos na aplicação;

$n$ é a dimensão do problema considerado;

$T_s$ *é o tempo de execução do algoritmo sequencial;*

$T_p$ é o tempo de execução do algoritmo paralelo, considerando-se $p$ processadores

# “Speed - Up”

O “Speed-Up” mede o ganho do processamento paralelo sobre o sequencial e é dado pela expressão:

$$Speed - up = \frac{T_s}{T_p}$$

# Aplicações e Análise de Performance

Nessa seção, a título de aplicação, consideraremos a questão de encontrar a solução do problema elíptico:

$$\begin{cases} U_{xx} + U_{yy} = 0 \\ U(0, y) = U(1, y) = 0 \\ U(x, 0) = 0 \\ U(x, 1) = 1 \end{cases}$$

Com,

$$(x, y) \in [0,1]x[0,1]$$

$$0 < y < 1$$

$$0 < x < 1$$

Esse problema na forma discretizada é dado por:

$$Mx = f,$$

$$M = \begin{pmatrix} A_1 & C_1 & 0 & 0 & \ldots & 0 & 0 & 0 \\ B_2 & A_2 & C_2 & 0 & \ldots & 0 & 0 & 0 \\ 0 & B_3 & A_3 & C_3 & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & C_{n-1} \\ 0 & 0 & 0 & 0 & \cdots & 0 & B_n & A_n \end{pmatrix}$$

Para resolução do problema proposto, que essencialmente se reduz a resolução de sistemas lineares de grande porte, utilizamos o algoritmo de Gauss-Seidel em paralelo. A seguir, mostramos tabelas referentes a eficiência e ao “speed-up”, obtidos para esse exemplo.

# MÉTODO DE GAUSS-SEIDEL

| Nº de Equações | T2 | T4 | T6 | T8 |
|---|---|---|---|---|
| 24576 | 0.99 | 0.96 | 0.95 | 0.86 |
| 12288 | 0.97 | 0.94 | 0.91 | 0.85 |
| 6144 | 0.97 | 0.93 | 0.86 | 0.72 |
| 1536 | 0.85 | 0.79 | 0.57 | 0.52 |

**Tabela 1: Eficiência**

# MÉTODO DE GAUSS-SEIDEL

| Nº de Processadores | Eficiência | "Speed-Up" |
|---|---|---|
| 2 | 0.99 | 1.98 |
| 4 | 0.96 | 3.85 |
| 6 | 0.95 | 5.73 |
| 8 | 0.86 | 6.85 |

**Tabela 2: “Speed-Up” e Eficiência para um sistema de 24576 equações**

# MÉTODO DE GAUSS-SEIDEL

| | Ts | T2 | T4 | T6 | T8 |
|---|---|---|---|---|---|
| P1 | 14.79 | 7.47 | 3.83 | 2.56 | 2.13 |
| P2 | | 7.46 | 3.83 | 2.58 | 2.15 |
| P3 | | | 3.80 | 2.58 | 2.15 |
| P4 | | | 3.84 | 2.56 | 2.16 |
| P5 | | | | 2.55 | 2.16 |
| P6 | | | | 2.57 | 2.13 |
| P7 | | | | | 2.16 |
| P8 | | | | | 1.98 |
| TOTAL | 14.79 | 14.93 | 15.30 | 15.40 | 17.02 |

Tabela 3: Tempos em segundos, para o Método de Gauss-Seidel