Referências

1. Pacheco, P., An Introduction to Parallel Programming, Morgan Kaufmann Publishers, 2011.

Multiplicação de matrizes

Multiplicação de matrizes

Multiplicação de matrizes é fundamental em algebra linear que é uma operação importante para algoritmos numéricos.

Se A, B e C são matrizes $n \times n$, então $C = A \cdot B$ é também uma matriz $n \times n$, e o valor de cada elemento em C é definido como:

$$C_{ij} = \sum_{k=0}^{n} A_{ik} B_{kj}$$

Multiplicação de matrizes

algoritmo serial

```
for (i = 0; i < n; i++)
   for (j = 0; j < n; j++) {
      C[i][j] = 0.0;
      for (k = 0; k < n; k++)
         C[i][j] = C[i][j] + A[i][k]*B[k][j];
   }
```

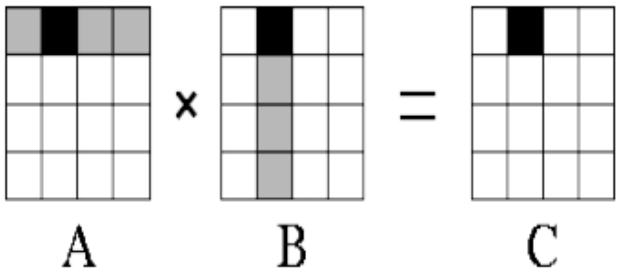
Onde:

A, B e C são matrizes de dimensão $n \times n$

Multiplicação de matrizes (v1)

Como distribuir as matrizes:

▷ decomposição 1-D: uma linha da matriz A e uma coluna da matriz B é enviada para um processo calcular o produto escalar.



Multiplicação de matrizes (v1)

```
int *A, *B, *C;
int *linha, *coluna;

linha = (int *) malloc (N * sizeof(int));
coluna = (int *) malloc (N * sizeof(int));

if (id == MESTRE) {
   A = (int *) malloc( N * N * sizeof(int));
   B = (int *) malloc( N * N * sizeof(int));
   C = (int *) malloc( N * N * sizeof(int));
   inicializa_exemplo(A, B);
   // ... continua
}
```

Multiplicação de matrizes (v1)

```
if (id == MESTRE) {
   // ... continua
   int processo = 0;
   for (int i = 0; i < N; i++){
      for (int j = 0; j < N; j++) {
         copia_linha  (i, A, linha, N);
         copia_coluna (j, B, coluna, N);
         if (processo % 4 == 0) // MESTRE
             C[i*N + j] = produto_escalar(linha, coluna, N);
         else
             envia(i, j, linha, coluna, N, processo % 4);
         processo++;
      }
```

```
        processo = 0;
        for (int j = 0; j < N; j++) {
            if (processo % 4 != 0) {
                int l, c, r;
                recebe_resultado(&l, &c, &r);
                C[l * N + c] = r;
            }
            processo++;
        }
    }
}
```

Multiplicação de matrizes (v1)

```
if (id != MESTRE) {
      int i, j, n;
      for (int k = 0; k < N*N; k += 4){
         recebe(&i, &j, &linha, &coluna, &n);
         int r = produto_escalar(linha, coluna, n);
         envia_resultado(i, j, r);
      }
}
```

Multiplicação de matrizes (v1)

```
void envia(int i, int j, int *lin, int *col, int n, int destino) {
   int tag = 33;
   int posicao = 0;
   MPI_Request request;
   MPI_Pack(&i, 1, MPI_INT, buffer, 100, &posicao, MPI_COMM_WORLD);
   MPI_Pack(&j, 1, MPI_INT, buffer, 100, &posicao, MPI_COMM_WORLD);
   MPI_Pack(&n, 1, MPI_INT, buffer, 100, &posicao, MPI_COMM_WORLD);
   MPI_Isend(buffer, 100, MPI_PACKED, destino, tag,
                                          MPI_COMM_WORLD, &request);
   MPI_Isend(lin, n, MPI_INT, destino, tag,
                                          MPI_COMM_WORLD, &request);
   MPI_Isend(col, n, MPI_INT, destino, tag,
                                          MPI_COMM_WORLD, &request);
}
```

Multiplicação de matrizes (v1)

```
void recebe(int *i, int *j, int **lin, int **col, int *n){
   int tag = 33;
   int posicao = 0;

   MPI_Recv(buffer, 100, MPI_PACKED, MPI_ANY_SOURCE, tag,
                                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
   MPI_Unpack(buffer, 100, &posicao, i, 1, MPI_INT, MPI_COMM_WORLD);
   MPI_Unpack(buffer, 100, &posicao, j, 1, MPI_INT, MPI_COMM_WORLD);
   MPI_Unpack(buffer, 100, &posicao, n, 1, MPI_INT, MPI_COMM_WORLD);
   MPI_Recv(*lin,  *n, MPI_INT, MESTRE, tag, MPI_COMM_WORLD,
                                                MPI_STATUS_IGNORE);
   MPI_Recv(*col, *n, MPI_INT, MESTRE, tag, MPI_COMM_WORLD,
                                                MPI_STATUS_IGNORE);
}
```

Multiplicação de matrizes (v1)

Como distribuir as matrizes:

▷ decomposição 1-D: uma linha da matriz A e uma coluna da matriz B é enviada para um processo calcular o produto escalar.

$$C_{ij} = \sum_{k=0}^{n} A_{ik} B_{kj}$$

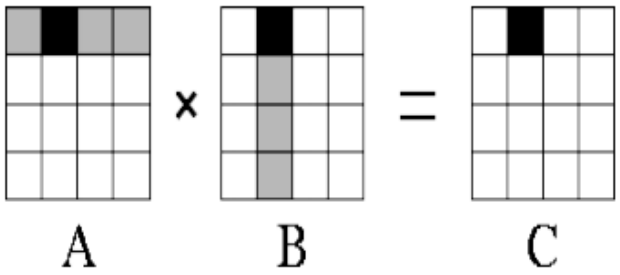
Problemas:
▷ Muitas trocas de mensagens

Complexidade: $O(N^3/P)$ (computação)

Multiplicação de matrizes (v2)

Vamos tentar diminuir a quantidade de dados transferidos:

▷ decomposição 1-D: uma linha da matriz A e uma coluna da matriz B é enviada para um processo calcular o produto escalar.



Multiplicação de matrizes (v2)

Inicialmente, o mestre inicializa A e B, em seguida distribui a matriz para todos os processos.

```
A = (int *) malloc( N * N * sizeof(int));
B = (int *) malloc( N * N * sizeof(int));

if (id == MESTRE) {
   C = (int *) malloc( N * N * sizeof(int));
   inicializa_exemplo(A, B);
}

MPI_Bcast(A, N * N, MPI_INT, MESTRE, MPI_COMM_WORLD);
MPI_Bcast(B, N * N, MPI_INT, MESTRE, MPI_COMM_WORLD);
```

Multiplicação de matrizes (v2)

```
if (id == MESTRE) {
   int processo = 0;

   for (int i = 0; i < N; i++){
      for (int j = 0; j < N; j++) {
         if (processo % 4 == 0)
             C[i*N + j] = multiplica_linha_coluna(A, B, i, j, N);
         else
             envia(i, j, processo % 4);

         processo++;
      }
```

```
      processo = 0;
      for (int j = 0; j < N; j++) {
         if (processo % 4 != 0) {
            int l, c, r;
            recebe_resultado(&l, &c, &r);
            C[l * N + c] = r;
         }
         processo++;
      }
   }
} // if mestre
```

Multiplicação de matrizes (v2)

```
int multiplica_linha_coluna(int *A, int *B, int linha,
                                            int coluna, int n){
   int r = 0;

   for (int i = 0; i < n; i++) {
      r += A[linha * n + i] * B[i * n + coluna];
   }

   return r;
}
```

Multiplicação de matrizes (v2)

```
void envia(int i, int j, int destino) {
   int tag = 33;
   int posicao = 0;
   MPI_Request request;
   MPI_Pack(&i, 1, MPI_INT, buffer, 100, &posicao, MPI_COMM_WORLD);
   MPI_Pack(&j, 1, MPI_INT, buffer, 100, &posicao, MPI_COMM_WORLD);
   MPI_Isend(buffer, 100, MPI_PACKED, destino, tag, MPI_COMM_WORLD,
                                                          &request);
}
```

Multiplicação de matrizes (v2)

```
void recebe(int *i, int *j){
   int tag = 33;
   int posicao = 0;

   MPI_Recv(buffer, 100, MPI_PACKED, MPI_ANY_SOURCE, tag,
                                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
   MPI_Unpack(buffer, 100, &posicao, i, 1, MPI_INT, MPI_COMM_WORLD);
   MPI_Unpack(buffer, 100, &posicao, j, 1, MPI_INT, MPI_COMM_WORLD);
}
```

Multiplicação de matrizes (v2)

```
void envia_resultado(int i, int j, int r){
   int tag = 44;
   int posicao = 0;
   MPI_Request request;

   MPI_Pack(&i, 1, MPI_INT, buffer, 100, &posicao, MPI_COMM_WORLD);
   MPI_Pack(&j, 1, MPI_INT, buffer, 100, &posicao, MPI_COMM_WORLD);
   MPI_Pack(&r, 1, MPI_INT, buffer, 100, &posicao, MPI_COMM_WORLD);
   MPI_Isend(buffer, 100, MPI_PACKED, MESTRE, tag, MPI_COMM_WORLD,
                                                          &request);
}
```

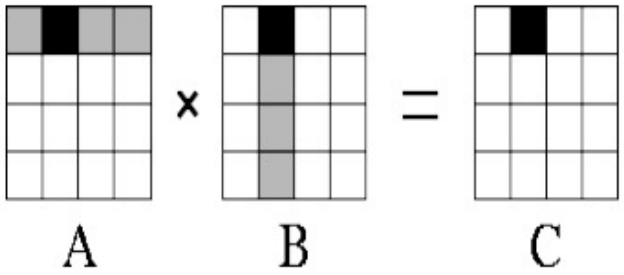
Multiplicação de matrizes (v2)

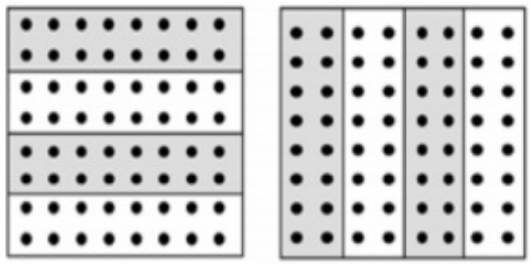
```
void recebe_resultado(int *i, int *j, int *r){
   int tag = 44;
   int posicao = 0;

   MPI_Recv(buffer, 100, MPI_PACKED, MPI_ANY_SOURCE, tag,
                                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
   MPI_Unpack(buffer, 100, &posicao, i, 1, MPI_INT, MPI_COMM_WORLD);
   MPI_Unpack(buffer, 100, &posicao, j, 1, MPI_INT, MPI_COMM_WORLD);
   MPI_Unpack(buffer, 100, &posicao, r, 1, MPI_INT, MPI_COMM_WORLD);
}
```

Multiplicação de matrizes

▷ partição por faixas: uma matriz é divida em grupos de linhas ou colunas completas, cada grupo será atribuida a um processo.





Multiplicação de matrizes

Cada processo recebe um subconjunto de:

- matriz linha ou matriz coluna

Para computar uma linha da matriz C cada processo deve ter

- uma linha da matriz A, e
- acesso a todas as colunas de B.

Multiplicação de matrizes (v3)

```
int *A, *B, *C, *local;
r = N / p;
A = (int *) malloc( N * N * sizeof(int));
B = (int *) malloc( N * N * sizeof(int));
C = (int *) malloc( N * N * sizeof(int));
local = (int *) malloc (r * N * sizeof(int));

if (id == MESTRE) {
     inicializa_exemplo(A, B);
}

MPI_Bcast(A, N * N, MPI_INT, MESTRE, MPI_COMM_WORLD);
MPI_Bcast(B, N * N, MPI_INT, MESTRE, MPI_COMM_WORLD);
```

Multiplicação de matrizes (v3)

```
int inicio, fim, r;
inicio = id * r;
fim    = id * r + r;
int u = 0;
for (int i = inicio; i < fim; i++){
   for (int j = 0; j < N; j++) {
         local[u] = 0;
         for (int k = 0; k < N; k++) {
            local[u] += A[i*N + k] * B[k*N + j];
         }
         u++;
   }
}
MPI_Gather(local, r*N, MPI_INT, C, r*N, MPI_INT, MESTRE,
                                              MPI_COMM_WORLD);
```

Multiplicação de matrizes (v3)

```
if (id == MESTRE){
      printf("Matriz C:\n");
      mostra_matriz(C, N);
}
free(A);
free(B);
free(C);
free(local);
```

Multiplicação de matrizes (v3)

Nesta versão de partição por faixas

- ▷ Há menos troca de mensagens:
- ▷ Cada processo sabe computar a sua faixa da matriz.

Complexidade: $O(N^3/p)$

Topologias Virtuais

Topologias Virtuais

Descrevem um mapeamento de processos MPI.

Duas principais suportadas em MPI:

- Cartesiana (grid) e
- grafo (graph).

Topologias Virtuais

Tais topologia são geralmente virtuais:

- ▷ Sem relação com a topologia física.
- ▷ Mas pode existir exceções.

São programadas com comunicadores e grupos:

- ▷ Programadas pelo desenvolvedor.

Topologias cartesianas

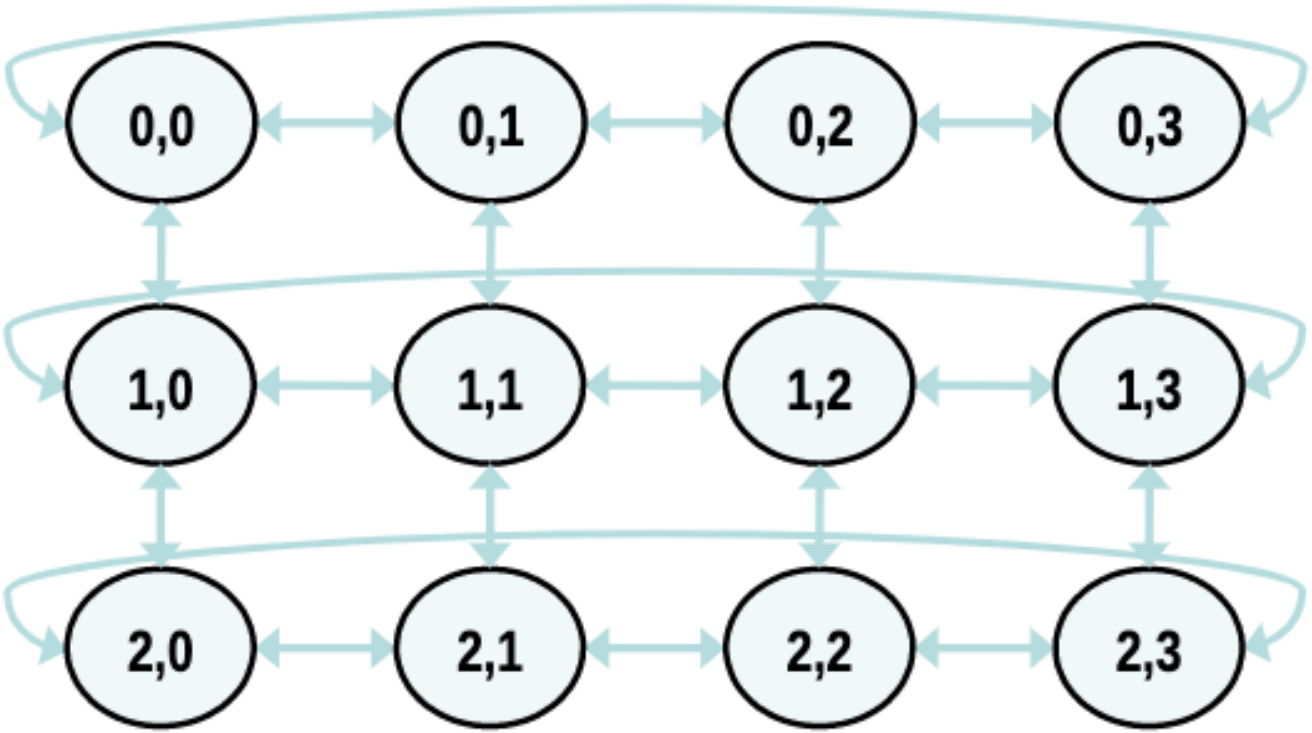
Topologias cartesianas

Cada processo está conectado aos seus vizinhos em um grid virtual

O limites do grid pode ter conexão cíclica

Os processos podem ser identificados pelas coordenadas cartesianas.

Topologias cartesianas



Criando uma topologia cartesiana

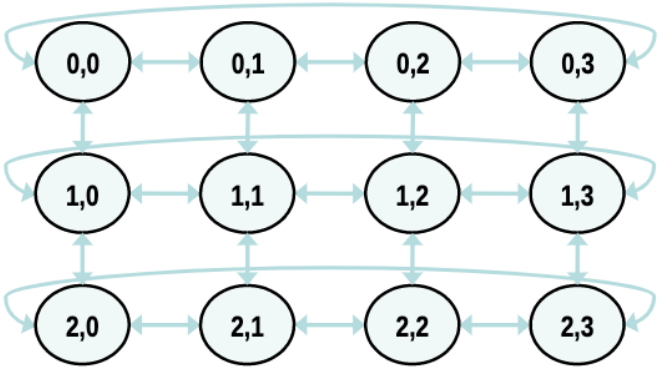
```
int MPI_Cart_create(MPI_Comm comm_old,
                    int ndims,
                    const int dims[],
                    const int periods[],
                    int reorder,
                    MPI_Comm *comm_cart)
```

Criando uma topologia cartesiana

```
int MPI_Cart_create(MPI_Comm comm_old,
                    int ndims,
                    const int dims[],
                    const int periods[],
                    int reorder,
                    MPI_Comm *comm_cart)
```

`comm_old`: comunicador existente
`ndims`: número de dimensões
`dims`: vetor de inteiros de tamanho ndims que especifica o número de processos em cada dimensão.
`periods`: vetor que indica se os limites são cíclicos.
`comm_cart`: novo comunicador cartesiano.

Topologias cartesianas




```
MPI_Comm comm;
int dim[2]    = {4, 3}, // 4 colunas x 3 linhas
    period[2] = {1, 0}, // ciclo: última coluna para a primeira
    reorder = 1;
MPI_Cart_create(MPI_COMM_WORLD, 2, dim, period, reorder, &comm);
```

Funções de mapeamento da topologia cartesiana

Mapeando a coordenada do processo para o rank.

```
int MPI_Cart_rank(MPI_Comm comm, int *coods, int *rank)
```

Funções de mapeamento da topologia cartesiana

Mapeando o rank para a coordenada do processo.

```
int MPI_Cart_coords(MPI_Comm comm, int rank,
                    int maxdims, int *coords)
```

```
MPI_Comm comm;
int dim[2]    = {4, 3}, // 4 colunas x 3 linhas
    period[2] = {1, 0}, // ciclo: última coluna para a primeira
    reorder = 1, coord[2], id;

MPI_Cart_create(MPI_COMM_WORLD, 2, dim, period, reorder, &comm);

if(rank==5) {
   MPI_Cart_coords(comm, rank, 2, coord);
   printf("P:%d my coordinates are %d %d\n",rank,coord[0],coord[1]);
}
if(rank==0) {
   coord[0]=3; coord[1]=1;
   MPI_Cart_rank(comm, coord, &id);
   printf("processor at (%d, %d), rank %d\n",coord[0],coord[1], id);
}
```

```
$mpirun -np 16 ./exemplo

P:5 my coordinates are 1, 2
processor at (3, 1), rank 10
```

Funções de mapeamento da topologia cartesiana

Computando os rankings de processos vizinhos

```
int MPI_Cart_shift (MPI_Comm comm, int direction, int disp,
                                    int *rank_source, int *rank_dest)
```

Esta função não envia dados, apenas retorna o rank dos vizinhos para que possa ser utilizado na comunicação subsequente.

Funções de mapeamento da topologia cartesiana

Computando os rankings de processos vizinhos

```
int MPI_Cart_shift (MPI_Comm comm, int direction, int disp,
                                    int *rank_source, int *rank_dest)
```

`comm`: comunicador

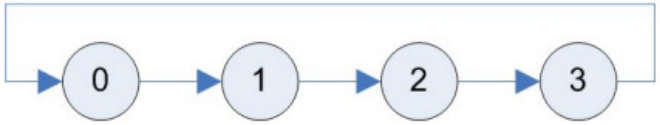
`direction`: coordenada dimensão da comunicação

`disp`: deslocamento (>0: shift para cima, <0 shift para baixo)

`rank_source`: rank do processo origem.

`rank_dest`: rank do processo destino.

Sendrecv com topologia cartesiana 1D




```
int dim[1],period[1];
dim[0] = nprocs;
period[0] = 1;
MPI_Comm ring_comm;

MPI_Cart_create(MPI_COMM_WORLD, 1, dim, period, 0, &ring_comm);
int source, dest;
MPI_Cart_shift(ring_comm, 0, 1, &source, &dest);
MPI_Sendrecv(right_bounday, n, MPI_INT, dest, rtag,
             left_boundary, n, MPI_INT, source, ltag,
             ring_comm, &status);
```

topologia cartesiana 2D

```
MPI_Comm comm;
int dim[2],period[2],reorder;
int up,down,right,left;
dim[0]=4; dim[1]=3;
period[0]=1; period[1]=0;
reorder=1;

MPI_Cart_create(MPI_COMM_WORLD,2,dim,period,reorder,&comm);

if(rank==9){
  MPI_Cart_shift(comm,0,1,&left,&right);
  MPI_Cart_shift(comm,1,1,&up,&down);
  printf("P%d neighbors are r: %d d:%d 1:%d u:%d\n",
                              rank, right, down, left, up);
}
```

Funções de mapeamento da topologia cartesiana

Criando um novo comunicador para uma grid de dimensão menor

```
int MPI_Cart_sub (MPI_Comm comm_old, int* coord, MPI_Comm comm_new)
```

`comm_old`: comunicador atual
`coord`: vetor que indica quais dimensões se manterão no subgrid
`comm_new`: new comunicador

Algoritmo de Fox

Algoritmo de Fox

Sejam A e B matrizes de tamanho $n \times n$, queremos computar $C = A \cdot B$ em paralelo.

Seja $q = \sqrt{p}$ um inteiro tal que divide n, onde p é o número de processos.

Crie uma topologia Cartesiana com processadores (i, j), de modo que $i, j = 0 \cdots q - 1$.

Denote $m = n/q$. Distribua A e B por blocos em p processadores tal que A_{ij} e B_{ij} são $m \times m$ blocos armazenados em processos (i, j).

Distribuição *checkerboard*

Exemplo:

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}$$

onde $A_{00} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix}$, $A_{01} = \begin{pmatrix} a_{02} & a_{03} \\ a_{12} & a_{13} \end{pmatrix}$, $A_{10} = \begin{pmatrix} a_{20} & a_{21} \\ a_{30} & a_{31} \end{pmatrix}$ e $A_{11} = \begin{pmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{pmatrix}$

Distribuição *checkerboard*

Todas as multiplicações das submatrizes podem ser computadas com diferentes processos. Uma submatriz $C_{i,j}$ do resultado de C pode ser calculada da seguinte maneira:

$$C_{i,j} = \sum_{k=0}^{q-1} A_{i,k} B_{k,j} = A_{i,0} B_{0,j} + A_{i,1} B_{1,j} + \cdots A_{i,q-1} B_{q-1,j}$$

Para o cálculo total da multiplicação de matrizes, uma quantidade de dados devem ser transmitidas.

Algoritmo de Fox

Ideia:

Cada processo mantém uma matriz de acordo com a distribuição *checkerboard*. Execute uma iteração $k = 0 \cdots q - 1$, o processo $p_{i,j}$ calcula:

$C_{i,j} = C_{i,j} + A_{i,\overline{k}} \cdot B_{\overline{k},j}$, $\overline{k} = (i + k) \mod q$

O algoritmo executa n estágios para matrizes de ordem n para cada termo $A_{i,k}B_{k,j}$ do produto escalar.

Algoritmo de Fox

O algoritmo para p processos contém as seguintes tarefas:

1. Determine o número de submatrizes por linha e por coluna, respectivamente: $q = \sqrt{p}$.
2. Atribua cada processo sua coordenada (i, j).
3. Determine a coordenada do processo para envio $p_{dest} : ((i - 1) \mod q, j)$.
4. Determine a coordenada do processo, o qual os dados são recebidos $((i + 1) \mod q, j)$.

Algoritmo de Fox

5. para $k = 0$ até $q - 1$ faça
   (i) Calcule $\overline{k} = (i + k) \mod q$
   (ii) Broadcast $A_{i,\overline{k}}$ para os processos na linha i.
   (iii) Calcule $C_{i,j} = C_{i,j} + A_{i,\overline{k}} \cdot B_{\overline{k},j}$ localmente.
   (iv) Envia $B_{\overline{k}+1 \mod q,j}$ para o processo p_{dest} e
   (v) Recebe $B_{\overline{k}+1 \mod q,j}$ da origem

Análise do algoritmo de Fox

Sejam A e B matrizes de tamanho $n \times n$, a multiplicação de matrizes $C = A \cdot B$, $C_{ij} = \sum_{k=0}^{q-1} A_{ik} \cdot B_{kj}$
Seja $p = q^2$ o número de processos organizados em um grid $q \times q$.
Armazene blocos $n/q \times n/q$ de A, B e C no processo (i, j).
A execução do algoritmo de Fox requer q iterações, em que cada processo multiplica seu bloco corrente da matriz A e B, e adiciona o resultado com o bloco corrente da matriz C.

Tempo de computação: $q\left(\frac{n}{q} \times \frac{n}{q} \times \frac{n}{q}\right) = \frac{n^3}{q^2} = \frac{n^3}{p}$

Exemplo matriz 2×2

$$A = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix}, \; B = \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix}$$

$$C = \begin{pmatrix} a_{00}b_{00} + a_{01}b_{10} & a_{00}b_{01} + a_{01}b_{11} \\ a_{10}b_{00} + a_{11}b_{10} & a_{10}b_{01} + a_{11}b_{11} \end{pmatrix}$$

Exemplo matriz 2×2

Assuma que temos n^2 processos, um para cada elemento da matriz A, B e C.

p_{00}	p_{01}
p_{10}	p_{11}

Suponha que os processos $p_{00}, p_{01}, p_{10}, p_{11}$ estão organizado num grid cartesiano.

Exemplo matriz 2×2

Estágio $k = 0$:

Broadcast $A_{i,i}$ para os processos na linha i.

a_{00}	a_{00}
a_{11}	a_{11}

Decomponha B no grid, de modo que $B_{i,j}$ seja colocado no processo p_{ij}

a_{00} b_{00}	a_{00} b_{01}
a_{11} b_{10}	a_{11} b_{11}

Exemplo matriz 2×2

Compute $c_{ij} = A \cdot B$ para cada processo.

a_{00} b_{00} $c_{00} = a_{00}b_{00}$	a_{00} b_{01} $c_{01} = a_{00}b_{01}$
a_{11} b_{10} $c_{10} = a_{11}b_{10}$	a_{11} b_{11} $c_{11} = a_{11}b_{11}$

Exemplo matriz 2×2

Agora, está tudo certo para o próximo estágio.

- ▷ Execute broadcast da próxima coluna (mod n) de A através dos processos e
- ▷ execute um deslocamento para cima dos valores de B.

Exemplo matriz 2×2

Estágio $k = 1$

A próxima coluna de A é $a_{0,1}$ na primeira linha e $a_{1,0}$ para a segunda linha (executou um deslocamento circular, mod n).

a_{01} b_{00} $c_{00} = a_{00}b_{00}$	a_{01} b_{01} $c_{01} = a_{00}b_{01}$
a_{10} b_{10} $c_{10} = a_{11}b_{10}$	a_{10} b_{11} $c_{11} = a_{11}b_{11}$

Execute o broadcast desses elementos nas respectivas linhas.

Exemplo matriz 2×2

Execute um deslocamento para cima dos valores de B.

a_{01} b_{10} $c_{00} = a_{00}b_{00}$	a_{01} b_{11} $c_{01} = a_{00}b_{01}$
a_{10} b_{00} $c_{10} = a_{11}b_{10}$	a_{10} b_{01} $c_{11} = a_{11}b_{11}$

Exemplo matriz 2×2

Compute $c_{ij} = A \cdot B$ para cada processo.

a_{01} b_{10} $c_{00} = c_{00} + a_{01}b_{10}$	a_{01} b_{11} $c_{01} = c_{01} + a_{01}b_{11}$
a_{10} b_{00} $c_{10} = c_{10} + a_{10}b_{00}$	a_{10} b_{01} $c_{11} = c_{11} + a_{10}b_{01}$

O algoritmo está completo após q estágios e os processos $p_{i,j}$ contém o resultado final para $c_{i,j}$

Exemplo matriz 3×3

Exemplo matriz 3×3

Considere multiplicação de matrizes 3×3:

$$\begin{pmatrix} 1 & 2 & 1 \\ 0 & 1 & 2 \\ 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 2 \\ 2 & 0 & 3 \\ 1 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 6 & 2 & 9 \\ 4 & 4 & 5 \\ 4 & 2 & 6 \end{pmatrix}$$

Exemplo matriz 3×3

Estágio $k = 0$:

Processo $(i, i \mod 3)$	Broadcast na linha i
$(0,0)$	$a_{00} = 1$
$(1,1)$	$a_{11} = 1$
$(2,2)$	$a_{22} = 1$

a_{00}, b_{00}	a_{00}, b_{01}	a_{00}, b_{02}
a_{11}, b_{10}	a_{11}, b_{11}	a_{11}, b_{12}
a_{22}, b_{20}	a_{22}, b_{21}	a_{22}, b_{22}

Processo (i, j) computa:

$c_{00} = 1 \times 1 = 1$	$c_{01} = 1 \times 0 = 0$	$c_{02} = 1 \times 2 = 2$
$c_{10} = 1 \times 2 = 2$	$c_{11} = 1 \times 0 = 0$	$c_{12} = 1 \times 3 = 3$
$c_{20} = 1 \times 1 = 1$	$c_{21} = 1 \times 2 = 2$	$c_{22} = 1 \times 1 = 1$

Execute o deslocamento cíclico das colunas de B

Exemplo matriz 3×3

Estágio $k = 1$:

Processo $(i, i+1 \mod 3)$	Broadcast na linha i
$(0,1)$	$a_{01} = 2$
$(1,2)$	$a_{12} = 2$
$(2,0)$	$a_{20} = 1$

a_{01}, b_{10}	a_{01}, b_{11}	a_{01}, b_{12}
a_{12}, b_{20}	a_{12}, b_{21}	a_{12}, b_{22}
a_{20}, b_{00}	a_{20}, b_{01}	a_{20}, b_{02}

Processo (i, j) computa:

$c_{00} = 1 + 2 \times 2 = 5$	$c_{01} = 0 + 2 \times 0 = 0$	$c_{02} = 2 + 2 \times 3 = 8$
$c_{10} = 2 + 2 \times 1 = 4$	$c_{11} = 0 + 2 \times 2 = 4$	$c_{12} = 3 + 2 \times 1 = 5$
$c_{20} = 1 + 1 \times 1 = 2$	$c_{21} = 2 + 1 \times 0 = 2$	$c_{22} = 1 + 1 \times 2 = 3$

Execute o deslocamento cíclico das colunas de B

Exemplo matriz 3×3

Estágio $k = 2$:

Processo $(i, i+2 \mod 3)$	Broadcast na linha i
$(0,2)$	$a_{02} = 2$
$(1,0)$	$a_{10} = 2$
$(2,1)$	$a_{21} = 1$

a_{02}, b_{20}	a_{02}, b_{21}	a_{02}, b_{22}
a_{10}, b_{00}	a_{10}, b_{01}	a_{10}, b_{02}
a_{21}, b_{10}	a_{21}, b_{11}	a_{21}, b_{12}

Processo (i, j) computa:

$c_{00} = 5 + 1 \times 1 = 6$	$c_{01} = 0 + 1 \times 2 = 2$	$c_{02} = 8 + 1 \times 1 = 9$
$c_{10} = 4 + 0 \times 1 = 4$	$c_{11} = 4 + 0 \times 0 = 4$	$c_{12} = 5 + 0 \times 2 = 5$
$c_{20} = 2 + 1 \times 2 = 4$	$c_{21} = 2 + 1 \times 0 = 2$	$c_{22} = 3 + 1 \times 3 = 6$

Implementação

Código adaptado de P. Pacheco, Parallel Programming with MPI,

http://www.cs.usfca.edu/~peter/ppmpi/

Implementação

```
typedef struct {
    int        p;          /* Total number of processes    */
    MPI_Comm   comm;       /* Communicator for entire grid */
    MPI_Comm   row_comm;   /* Communicator for my row      */
    MPI_Comm   col_comm;   /* Communicator for my col      */
    int        q;          /* Order of grid                */
    int        my_row;     /* My row number                */
    int        my_col;     /* My column number             */
    int        my_rank;    /* My rank in the grid comm     */
} GRID_INFO_T;
```

Implementação

```
void Setupgrid (GRID_INFO_T* grid)
{
    int old_rank;
    int dimensions[2];
    int wrap_around[2];
    int coordinates[2];
    int free_coords[2];

    /* Set up Global Grid Information */
    MPI_Comm_size(MPI_COMM_WORLD, &(grid->p));
    MPI_Comm_rank(MPI_COMM_WORLD, &old_rank);
```

```
/* We assume p is a perfect square */
grid->q = (int) sqrt((double) grid->p);
dimensions[0] = dimensions[1] = grid->q;

/* We want a circular shift in second dimension. */
/* Don't care about first                        */
wrap_around[0] = wrap_around[1] = 1;
MPI_Cart_create(MPI_COMM_WORLD, 2, dimensions,
    wrap_around, 1, &(grid->comm));
MPI_Comm_rank(grid->comm, &(grid->my_rank));
MPI_Cart_coords(grid->comm, grid->my_rank, 2,
    coordinates);
grid->my_row = coordinates[0];
grid->my_col = coordinates[1];
```

```
    /* Set up row communicators */
    free_coords[0] = 0;
    free_coords[1] = 1;
    MPI_Cart_sub(grid->comm, free_coords,
        &(grid->row_comm));

    /* Set up column communicators */
    free_coords[0] = 1;
    free_coords[1] = 0;
    MPI_Cart_sub(grid->comm, free_coords,
        &(grid->col_comm));
} /* Setup_grid */
```

Implementação

```
void Fox(
        int                 n           /* in  */,
        GRID_INFO_T*        grid        /* in  */,
        LOCAL_MATRIX_T*     local_A     /* in  */,
        LOCAL_MATRIX_T*     local_B     /* in  */,
        LOCAL_MATRIX_T*     local_C     /* out */) {

    LOCAL_MATRIX_T*  temp_A; /* Storage for the sub-    */
                             /* matrix of A used during */
                             /* the current stage       */
    int              stage;
    int              bcast_root;
    int              n_bar;  /* n/sqrt(p)               */
```

```
int             source;
int             dest;
MPI_Status      status;
n_bar = n/grid->q;
Set_to_zero(local_C);

/* Calculate addresses for circular shift of B */
source = (grid->my_row + 1) % grid->q;
dest = (grid->my_row + grid->q - 1) % grid->q;

/* Set aside storage for the broadcast block of A */
temp_A = Local_matrix_allocate(n_bar);

for (stage = 0; stage < grid->q; stage++) {
    bcast_root = (grid->my_row + stage) % grid->q;
```

```
        if (bcast_root == grid->my_col) {
            MPI_Bcast(local_A, 1, local_matrix_mpi_t,
                bcast_root, grid->row_comm);
            Local_matrix_multiply(local_A, local_B,
                local_C);
        } else {
            MPI_Bcast(temp_A, 1, local_matrix_mpi_t,
                bcast_root, grid->row_comm);
            Local_matrix_multiply(temp_A, local_B,
                local_C);
        }
        MPI_Sendrecv_replace(local_B, 1, local_matrix_mpi_t,
            dest, 0, source, 0, grid->col_comm, &status);
    } /* for */

} /* Fox */
```

Fim