Introdução ao MPI

Aleardo Manacero Jr.

Ambientes de troca de mensagens

A programação paralela em sistemas multicomputadores pode ser feita através de vários mecanismos

Um desses mecanismos faz uso de bibliotecas de trocas de mensagens, como o MPI

O MPI

Message Passing Interface, ou MPI, foi desenvolvido por um fórum com o objetivo de ser um padrão entre os ambientes de troca de mensagens

Com esse objetivo, MPI pode ser usado na programação de clusters, redes e até mesmo em máquinas massivamente paralelas

O MPI

Desenvolvido a partir de outras bibliotecas, como o PVM

No MPI o paralelismo está restrito ao modelo SPMD SPMD significa Single Program Multiple Data

Pode ser usada em programas C, C++, Fortran (ou Java, python, R, etc)

Escopo de programação

Num programa paralelo as funções do MPI podem ser usadas dentro do intervalo marcado pelas instruções

MPI_Init (&argc, &argv)

e

MPI_Finalize

Iniciando o MPI

MPI_Init recebe uma cópia dos argumentos passados na chamada do programa (*argc* e *argv*)

Durante sua chamada inicializa a variável *MPI_COMM_WORLD*, que armazena os dados sobre os processos disparados

Programando com MPI

Funções de controle

```
MPI_Comm_size(MPI_COMM_WORLD, &size)
/* retorna o número de processos disparados */

MPI_Comm_rank(MPI_COMM_WORLD, &myid)
/* retorna a identidade (rank) do processo */

MPI_Cart_create(COMM,dims,dsize,wrap,reorder,cart)
/*cria uma topologia cartesiana de dims dimensões, sendo
que wrap diz se é fechada e reorder se pode reordenar os
comunicadores em cart */
```

Programando com MPI

Funções de controle

MPI_Cart_coords(cart, myrank, dims, coords)

/* retorna as coordenadas numa topologia de dims dimensões (coords é um vetor) */

MPI_Cart_shift(cart, dim, displ, &src, &dst)

/* retorna ranks dos vizinhos (a uma distância displ) na topologia */

MPI_Cart_get(cart, ndims, dims, periods, coords)

/* recupera a topologia */

Programando com MPI

Funções de comunicação simples

MPI_Send(msg, length, type, id, tag, comm)

/* envia msg para processo **id** */

MPI_Recv(msg, length, type, id, tag, comm, status)

/* recebe msg do processo **id** */

MPI_Sendrecv(smsg, slength, stype, dest, stag, rmsg, rlength, rtype, source, rtag, comm, status)

/* envia smsg para processo **dest** e recebe rmsg do processo **source** */

Programando com MPI

Identificação dos parâmetros de comunicação

msg é a informação (em um buffer)

length é o tamanho da informação

type identifica o tipo dos dados

id identifica os processos comunicantes

tag identifica a mensagem

status armazena o resultado da comunicação

Tipos de dados pré-definidos

Predefined MPI datatypes

MPI datatype	*C datatype*
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Programando com MPI

Mais funções de comunicação

MPI_Barrier(group)

/* faz processos em *group* esperarem pelos demais */

MPI_Bcast(msg, count, type, root, comm)

/* Comunicação por broadcast, disparado pelo processo com rank *root*, devendo existir em todos os processos do grupo *comm* */

Programando com MPI

Continuando ....

MPI_Scatter(msg, count, type, rmsg, rcount, rtype, root, comm)

/* envia um segmento (de tamanho count) de msg para cada processo em comm, que o recebe em rmsg */

MPI_Gather(msg, count, type, rmsg, rcount, rtype, root, comm)

/* recebe um segmento (msg) de cada processo e o coloca em rmsg) */

Programando com MPI

Mais funções

MPI_Reduce(operand,result,count,type,op,root,comm)

/* faz como MPI_Gather, porém realizando uma operação (op) entre todos os dados enviados */

MPI_Allreduce(operand, result, count, type, op, comm)

/* mesmo que MPI_Reduce, porém retornando o resultado final a todos os processos */

op pode ser uma das operações de redução apresentadas na próxima tabela

Operadores de redução

Predefined reduction operators in MPI

Operation Name	*Meaning*
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

Outras funções

Além dessas funções básicas existem funções para:

- Definição de grupos de processos comunicantes (*communicators*)
- Manipulação de topologia dos comunicadores
- Comunicação síncrona e assíncrona
- Definição de tipos de dados estruturados

Referências

http://www.open-mpi.org página oficial para o openMPI (usado neste curso)

http://www.mcs.anl.gov/mpi página do MPI no Argonne Natl Lab, com muita informação adicional

http://www.usfca.edu/~peter/ppmpi contém fontes dos programas do livro “Parallel Programming with MPI” de P. Pacheco

Exemplo de implementação

Como exemplo do uso de MPI temos o problema de transferência de calor, enunciado da seguinte forma:

> Suponha uma chapa em que as bordas são mantidas em temperatura constante, assim como pontos isolados da chapa
>
> Como determinar a temperatura em um ponto qualquer da chapa?

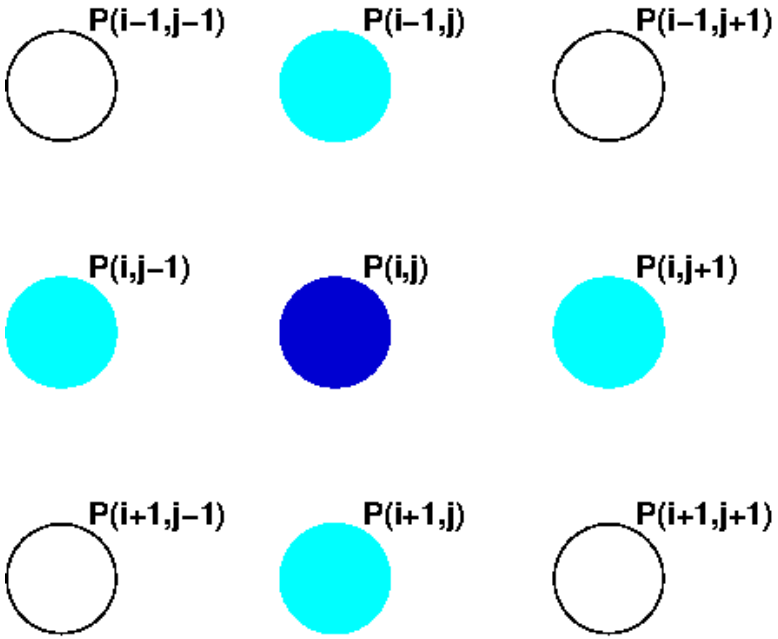
Transferência de calor

Pode-se usar algoritmos de aproximações sucessivas, como o chamado algoritmo *red & black*

Nesse caso, cada ponto da chapa tem sua temperatura, numa iteração, como sendo a média das temperaturas de seus vizinhos e de sua própria na iteração anterior

Transferência de calor

Os pontos a serem considerados como vizinhos em cada iteração são os vizinhos na vertical e na horizontal, como em:



$$P'(i,j) = \frac{P(i,j) + P(i,j-1) + P(i,j+1) + P(i-1,j) + P(i+1,j)}{5}$$

Transferência de calor

Assim, a chapa pode ser vista como uma enorme matriz, em que cada elemento é o valor da temperatura da chapa num ponto

Os valores de contorno e constantes são também considerados elementos da matriz

Transferência de calor

O algoritmo Red & Black faz o cálculo das temperaturas numa iteração considerando valores da matriz red, armazenando os novos resultados na matriz black

Na iteração seguinte a matriz red passa a ser o destino dos resultados calculados a partir da matriz black (daí o nome do algoritmo!)

Transferência de calor

A programação desse algoritmo em paralelo pode ser feita dividindo-se a matriz em vários grupos de linhas ou colunas (dependendo da linguagem usada), destinando cada grupo para um dos processos paralelos

Algumas dessas linhas (ou colunas), que estão nas fronteiras entre grupos, são compartilhadas entre pares de processos (vizinhos dentro da matriz)

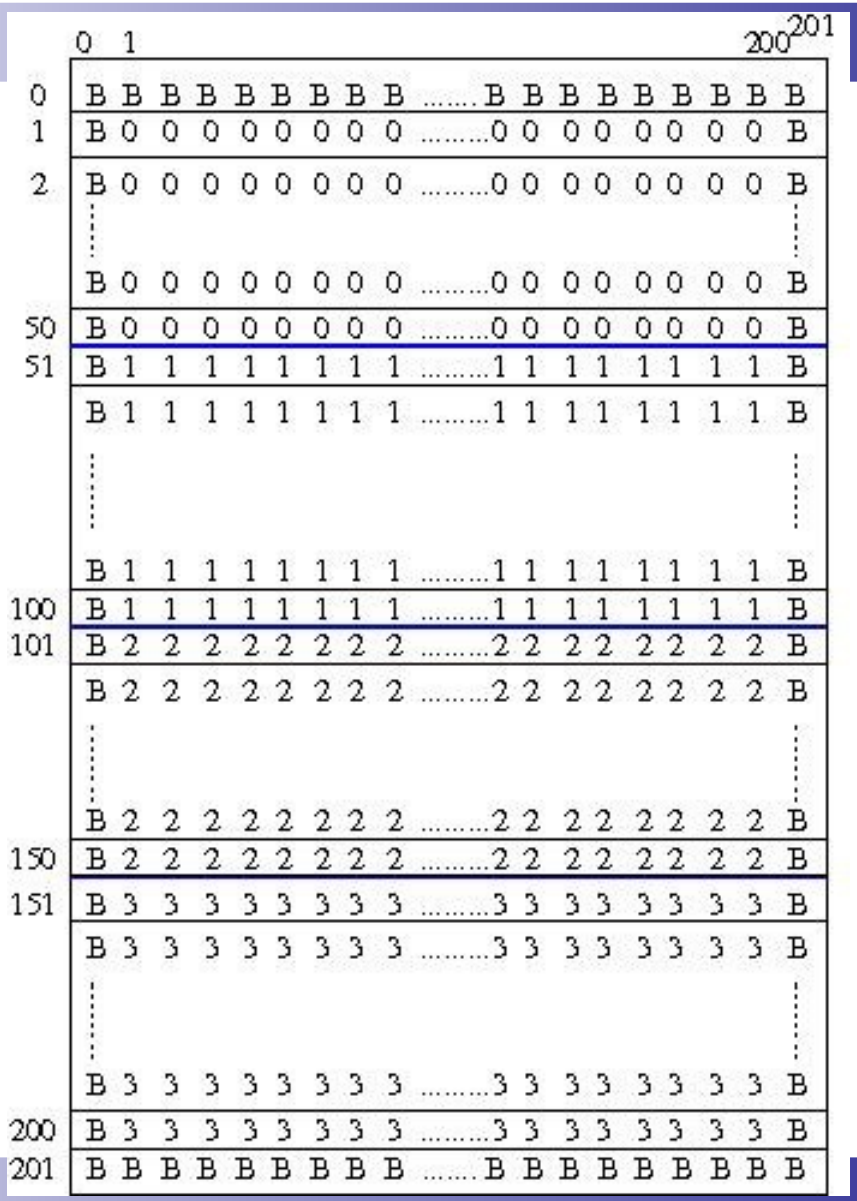
Transferência de calor

A operação em SPMD ocorre pela criação de barreiras de sincronismo entre cada iteração

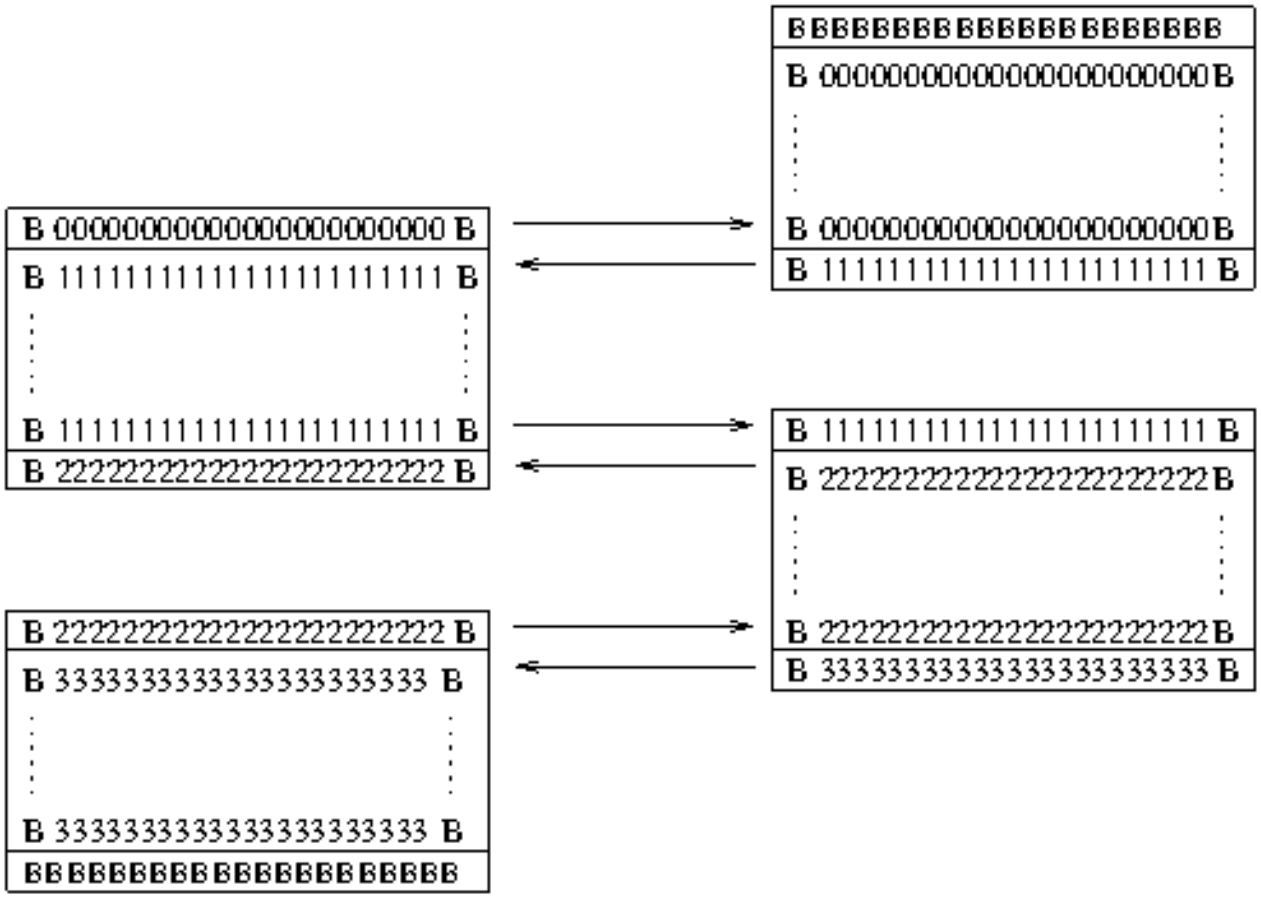
As barreiras podem ser criadas através de primitivas bloqueantes de comunicação, como a função MPI_Sendrecv( )

As informações sobre as linhas compartilhadas são trocadas entre os processos vizinhos a cada iteração

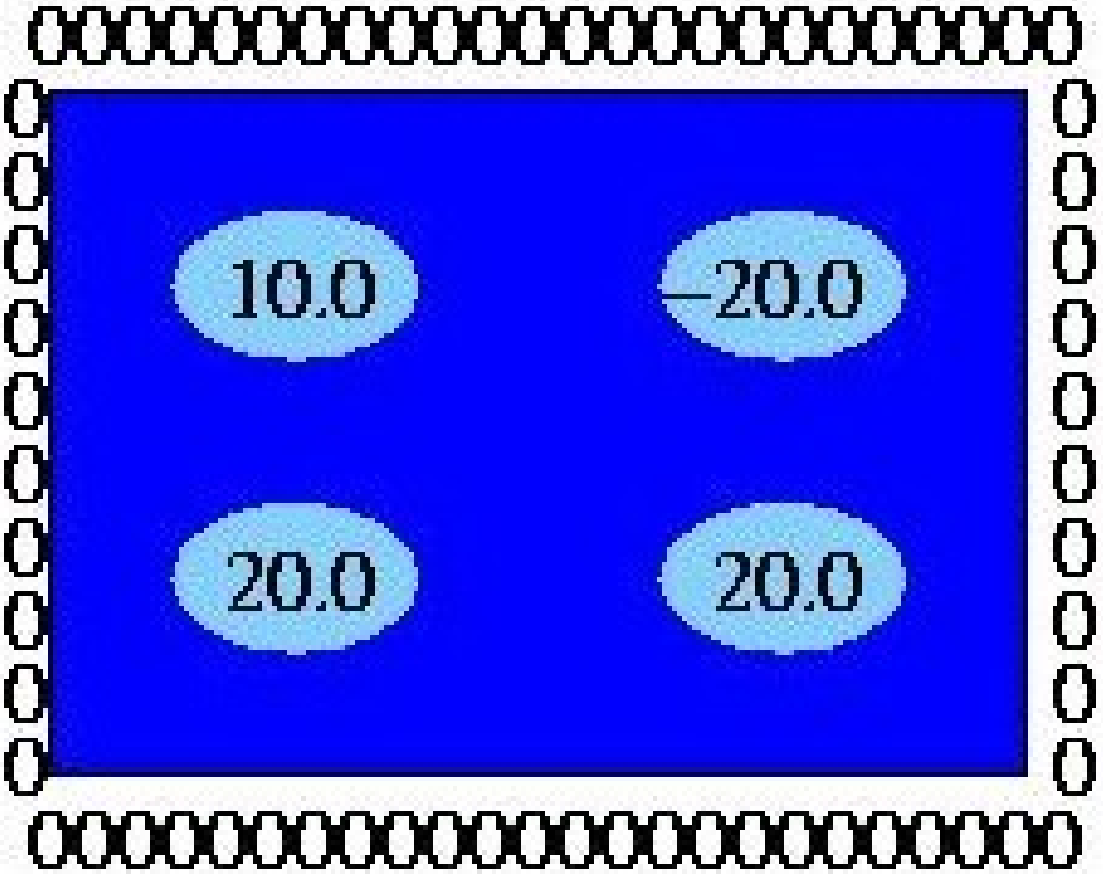
Transferência de calor



Transferência de calor



Exemplo de chapa



OBSERVAÇÕES

Para os dois programas se usou a implementação open-mpi

A compilação é feita usando o comando

```
mpicc -o arqexec arqfonte.c
```

A execução é feita usando o comando

```
mpirun -np numproc arqexec
```

Versão 1

Nesta versão usaremos as primitivas simples de comunicação, ou seja, MPI_Send, MPI_Recv e MPI_Sendrecv

Usaremos também o modelo SPMD puro, típico do MPI

Código.....


```
#include "stdio.h"
#include <mpi.h>

#define maxtime 100000
/* Simulação é executada com MINPROC processos, podendo
* ser igual a 1 para testes. Para uma matriz maior é
* preciso ajustar MINPROC p/ o numero real de nós */
#define MINPROC 4
#define cols 200
#define totrows 200
#define rows totrows/MINPROC + 2

double black[rows+2][cols+2];
```

Código.....


```
void storeconst(int s, int e, int row, int col, double val)
{ if (row >= s && row <= e)
      black[row-s][col] = val;   }

int main(int argc, char **argv)
{double red[rows+2][cols+2];
 int s, e, mylen, r, c, tick;
 int inum, nproc, dims, coords, ndim = 1;
 int periods, reorder;// usadas como booleanas
 int upproc, downproc;
 int comm1d;
 MPI_Status status;
```

Código.....


```
// Chamando MPI_INIT
   MPI_Init (&argc, &argv);
// Recebendo o número de processos em paralelo
   MPI_Comm_size (MPI_COMM_WORLD, &nproc);

   dims = nproc;
   periods = 0; // significa um valor falso
   reorder = 1; // significa um valor verdade
```

Código.....


```
// cria grupo cartesiano unidimensional de processos
   MPI_Cart_create (MPI_COMM_WORLD, ndim, &dims,
                    &periods, reorder, &comm1d);
// Pega valor de rank dentro do grupo de comunicadores
   MPI_Comm_rank (comm1d, &inum);

// Pega a topologia do grupo COMM1D e o lugar
// desse processo dentro dessa topologia
   MPI_Cart_get (comm1d, ndim, &dims, &periods, &coords);
```

Código.....


```
/* Retorna os vizinhos de distância 1 acima e abaixo,    *
 * sendo que os das extremidades terão apenas um deles */
   MPI_Cart_shift (comm1d, 0, 1, &upproc, &downproc);
   s = totrows/nproc*inum;
   e = totrows/nproc*(inum+1);
   mylen = (e - s) + 1;
   if (mylen > rows)
     { printf("No space, need %d, have %d\n", mylen, rows);
       printf("%d %d %d %d %d\n",totrows,nproc,inum,s,e);
     }
   printf ("%d %d %d %d %d %d %d \n", inum, nproc, coords,
                                upproc, downproc, s, e);
```

Código.....


```
// Começa frio (todos os pontos em 0◦C)
   for (r=0; r <= mylen+1; r++)
     for (c=0; c <= cols+1; c++)
            black[r][c] = 0.0;
// Inicia a execução das iterações
   for (tick = 1; tick <= maxtime; tick++)  {
// Inicia valores das fontes constantes de calor
      storeconst(s, e, totrows/3, cols/3, 10.0);
      storeconst(s, e, 2*totrows/3, cols/3, 20.0);
      storeconst(s, e, totrows/3, 2*cols/3, -20.0);
      storeconst(s, e, 2*totrows/3, 2*cols/3, 20.0);
```

Código.....


```
// Envia para vizinho de cima e recebe do vizinho abaixo
    MPI_Send(&black[1][0], cols, MPI_DOUBLE, upproc,
                                          1, comm1d);
    MPI_Recv(&black[mylen+1][0], cols, MPI_DOUBLE,
                        downproc, 1, comm1d, &status);

// Envia para baixo e recebe de cima num único comando
    MPI_Sendrecv (&black[mylen][0], cols, MPI_DOUBLE,
          downproc, 2, &black[0][0], cols, MPI_DOUBLE,
                      upproc, 2, MPI_COMM_WORLD, &status);
```

Código.....


```
/* Calcula-se as temperaturas nessa iteração */
    for (r=1; r <= mylen; r++)
      for (c=1; c <= cols; c++)
         red[r][c] = ( black[r][c] +
                       black[r][c-1] +
                       black[r-1][c] +
                       black[r+1][c] +
                       black[r][c+1] )
                                / 5.0;
```

Código.....


```
/* Copia a matriz – Normalmente faríamos a
  fase vermelha do laço, mas assim diminuímos
  o tamanho do exemplo */
      for (r=1; r <= mylen; r++)
        for (c=1; c <= cols; c++)
           black[r][c] = red[r][c];
 } /* Final do laço principal
  "for (tick = 1; tick <= maxtime; tick++)"
  */
```

Código.....


```
// Imprime parte da matriz do processo 1
 if (inum==1)
    { for (r=12; r <= 22; r++) {
        for (c=62; c <= 70; c++)
          printf("%lf ",black[r][c]);
      puts("  "); }
    }
  MPI_Finalize();
}
```

Versão 1

No código apresentado ainda falta fazer a junção das matrizes parciais

Isso pode ser feito fazendo com que todos os processos enviem a sua matriz para o processo 0 (em vez de se imprimir parte de uma das matrizes)

Versão 2

Nesta segunda versão faremos uso de primitivas de comunicação em grupos, como MPI_Bcast

Em particular, nesse caso teremos uma estrutura de programação tipo Mestre-Escravo

Código.....


```
#include "stdio.h"
#include <mpi.h>

#define MINPROC 4
#define cols 200
#define rows 200
#define maxtime 100000

int main(int argc, char **argv)
{double red[rows+2][cols+2], black[rows+2][cols+2];
 int s, e, ls, le, mylen, i, r, c, tick;
 int inum, nproc, src, dest, tag;
 MPI_Status status;
```

Código.....


```
   puts ("Chamando MPI_INIT");
   MPI_Init (&argc, &argv);
   MPI_Comm_size (MPI_COMM_WORLD, &nproc);
   MPI_Comm_rank (MPI_COMM_WORLD, &inum);
   s = 1 + inum*(rows/nproc);
   e = s + (rows/nproc) - 1;

// Começa frio (todos os pontos em 0◦C)
   if (inum==0)
     for (r=0; r <= mylen+1; r++)
        for (c=0; c <= cols+1; c++)
            black[r][c] = 0.0;
```

Código.....


```
// Inicia a execução das iterações
   for (tick = 1; tick <= maxtime; tick++)   {

// Inicia valores das fontes constantes de calor
      black[rows/3][cols/3] = 10.0;
      black[2*rows/3][cols/3] = 20.0;
      black[rows/3][2*cols/3] = -20.0;
      black[2*rows/3][2*cols/3] = 20.0;

// Faz o broadcast da matriz aos demais processos
      MPI_Bcast(black, (rows+2)*(cols+2), MPI_DOUBLE,
                                          0, MPI_COMM_WORLD);
```

Código.....


```
// Calcula uma iteração do fluxo para sua parte da matriz
   for (r=s; r<=e; r++)
     for (c=1; c<=rows; c++)
       red[r][c] = ( black[r][c] +
                     black[r][c-1] +
                     black[r-1][c] +
                     black[r+1][c] +
                     black[r][c+1] )
                               / 5.0;
```

Código.....


```
// Processo mestre recolhe os resultados na matriz BLACK
 if (inum == 0)
  { for (r=s; r<=e; r++)
      for (c=1; c<=rows; c++)
          black[r][c] = red[r][c];
    for (i=1; i<=nproc-1; i++)
     { ls = 1 + i * (rows/nproc);
       le = ls + (rows/nproc) - 1;
       mylen = (le - ls) + 1;       src = i;       tag = 0;
       MPI_Recv(&black[ls][0], mylen*(cols+2), MPI_DOUBLE,
                src, tag, MPI_COMM_WORLD, &status);
     }
  } // final do "then" para "if (inum==0)" ;
```

Código.....


```
   else  // demais processos enviam sua parte
     { mylen = (e -s) + 1;
       dest = 0;
       tag = 0;
       MPI_Send (&red[s][0], mylen*(cols+2), MPI_DOUBLE,
                                 dest, tag, MPI_COMM_WORLD);
     }
  }  /* Final do laço principal "for (tick = 1; tick <=
maxtime; tick++)" */
```

Código.....


```
    if (inum==0) // imprime parte da matriz
    { for (r=62; r <= 72; r++) {
        for (c=62; c <= 70; c++)
          printf("%lf ",black[r][c]);
      puts("  "); }
    }

  MPI_Finalize();
}
```

Comparação entre versões

A versão mestre-escravo é um pouco mais intuitiva (alguém controla os demais)

A versão SPMD é mais eficiente por diminuir o tamanho das mensagens

Nas medições feitas a versão SPMD foi pelo menos três vezes mais rápida (menos iterações)

Medições? O que é isso?

O objetivo de computação paralela é acelerar a execução de programas

Isso implica em saber se a aceleração é eficiente ou não, ou saber o desempenho do programa

A determinação do desempenho envolve técnicas de medição (benchmarking)

Desempenho em sistemas paralelos

Envolve um conjunto de medidas diferente daquele que se entende por desempenho em sistemas sequenciais

Em sistemas paralelos as principais medidas são:

Ganho de velocidade (*speedup*)

Capacidade de crescimento (*scalability*)

Escalabilidade

Mede o quanto um problema ou sistema pode crescer sem prejuízo significativo de desempenho

Depende de características do ambiente (hardware e software), como:

- Granulação (ou granulosidade)
- Custo de comunicação
- Necessidade de comunicação

Escalabilidade

Por medir a capacidade de crescimento é de especial interesse a quem paga pelo ambiente paralelo

Não é uma indicação de ganho máximo de velocidade, mas sim de quando esse ganho passa a crescer menos que o necessário (ou desejável)

Ganho de velocidade (speedup)

Temos vários tipos de *speedup*:

- Teórico
  - tamanho fixo
  - tempo fixo
- Medido
  - pelo sequencial
  - pelo paralelo monoprocessado

Speedups teóricos

Lei de Amdahl, que trata o problema de tamanho fixo

Lei de Gustafson, que trata o problema de tempo fixo

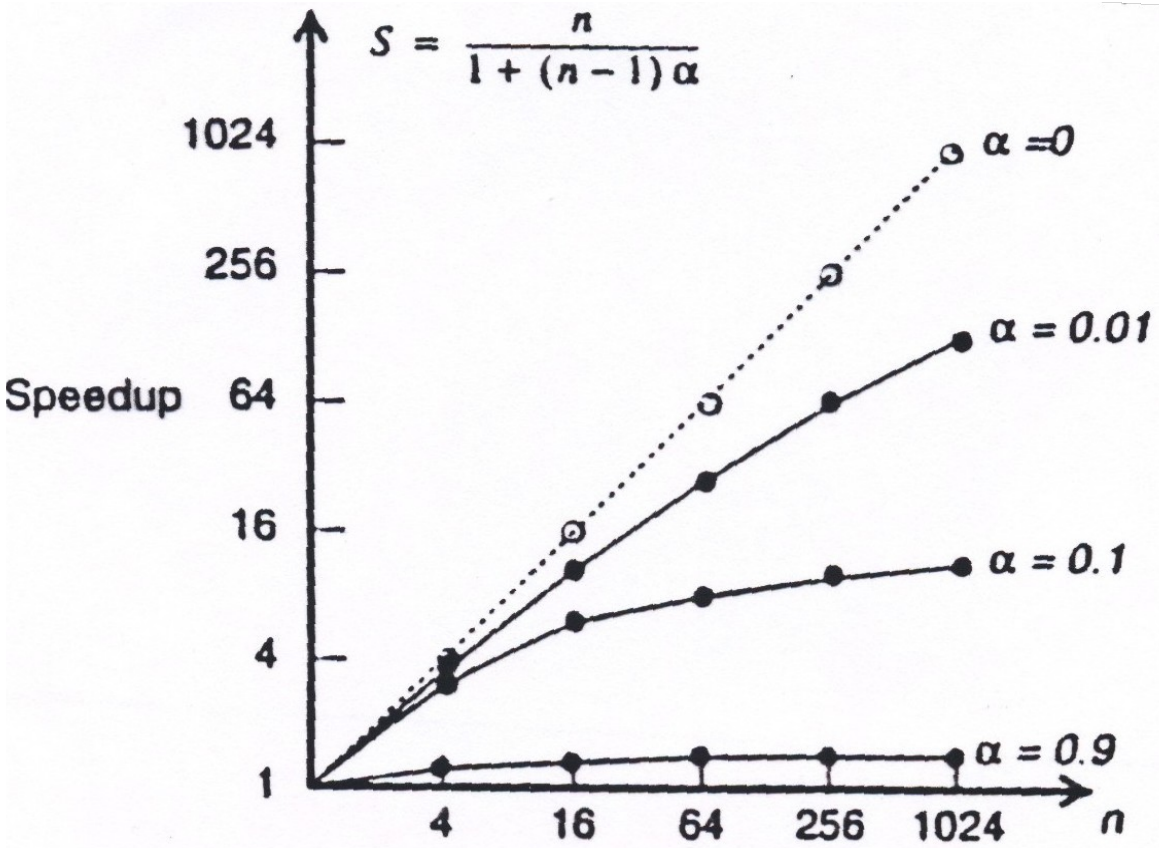
Lei de Amdahl

Define α como sendo a probabilidade de processamento sequencial

Então, o *speedup* para N elementos paralelos resulta em:

$$S_n = \frac{n}{1 + (n-1).\alpha}$$

Curvas de speedup



Problemas com medição

Antes de medir é preciso definir várias coisas, ou melhor dizendo, responder às seguintes questões:

O que medir?

Como medir?

Por que medir?

Problemas da medição

A solução das questões sobre o que e como medir partem da resposta sobre porque medir.

Identificar precisamente o motivo de se estar medindo ajuda a identificação dos outros problemas (*não necessariamente sua solução*).

Por que medir?

Respostas possíveis:

- Medida do desempenho global
- Informações para melhorar o desempenho global
- Informações sobre partes específicas do sistema
- etc.

O que medir?

Partindo da escolha anterior, a resposta pode ficar entre:

- Tempo de execução
- Tempo de CPU
- Tempo gasto em certas atividades
- Ganhos de velocidade em sistemas paralelos
- Eficiência na utilização de recursos
- etc., etc., etc.

Como medir?

Monitoração por hardware

- Uso de analisadores lógicos e outros equipamentos

Monitoração por software

- Uso de interrupções do sistema operacional

Modificação de código

- Inserção de código, **fonte, objeto ou executável**, para medição

Monitoração X Modificação

Monitoração

- é mais precisa
- exige conhecimentos sobre hardware e sistema operacional

Modificação de código

- é feita com o uso de ferramentas prontas
- usa técnicas como *profiling* e extração de eventos

Profiling / extração de eventos

Várias ferramentas existentes para programas sequenciais

prof, gprof, pixie, tcov, etc.

Problemas:

Precisão da amostragem

Não tratam paralelismo

Profiling / extração de eventos

Para sistemas paralelos existem ferramentas mais complexas, como

- *Tau*
- *Totalview*
- *Vampir*
- *Ferramentas de fabricantes*

Modificação de objeto/executável


```
main() {
   int l;
   for (l=0; l < 10000; l++) {
      if (l%2 == 0) foo();
      bar();
      baz();
   }
}
```

Modificação de objeto/executável


```
foo(){
  int j;
  for (j=0; j<200000; j++);
}
```

```
bar(){
  int i;
  for (i=0; i<200000; i++);
}
```

Modificação de objeto/executável


```
baz(){
  int k;
  for (k=0; k<300000; k++);
}
```

```
$[1] gcc -pg loops.c -o loops
$[2] ./loops
$[3] gprof > loops.prof
$[4] more loops.prof
```

Modificação de objeto/executável

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
53.89	0.08	0.08	10000	8.08	8.08	baz
33.68	0.13	0.05	10000	5.05	5.05	bar
13.47	0.15	0.02	5000	4.04	4.04	foo

Modificação de objeto/executável


```
index  % time    self  children    called        name
                                                 <spontaneous>
[1]    100.0    0.00    0.15                     main [1]
                0.08    0.00   10000/10000           baz [2]
                0.05    0.00   10000/10000           bar [3]
                0.02    0.00    5000/5000            foo [4]
-----------------------------------------------
                0.08    0.00   10000/10000           main [1]
[2]     53.3    0.08    0.00   10000                 baz [2]
-----------------------------------------------
                0.05    0.00   10000/10000           main [1]
[3]     33.3    0.05    0.00   10000                 bar [3]
-----------------------------------------------
                0.02    0.00    5000/5000            main [1]
[4]     13.3    0.02    0.00    5000                 foo [4]
-----------------------------------------------
```

This table describes the call tree of the program, and was sorted by the total amount of time spent in each function and its children.

Modificação de código fonte

O programador insere chamadas de funções para contagem de tempo (instrumentação) dentro de seu código.

A medição, portanto, depende da existência de acesso ao código fonte do programa.

Problemas de precisão e amostragem.

Modificação de código fonte


```
#include <time.h>
#include <sys/times.h>
#include “stdio.h”

float etime ()
{struct tms local;
 times (&local);
 return ((float)local.tms_utime/100.0 +
  (float)local.tms_stime/100.0);
}
```

Modificação de código fonte


```
main()
{float Duration;
……
    Duration = etimes();
       do_whatever_has_to_be_measured();
    Duration = etimes() - Duration;
    printf ("Work took %f seconds\n",Duration);
}
```

Modificação de código fonte

Com MPI é possível usar MPI_Wtime para obter tempos de execução em trechos específicos do programa, como em:

```
double start, finish;
    MPI_Barrier(comm);
    start = MPI_Wtime( );
        …
    MPI_Barrier(comm);
    Finish = MPI_Wtime( );
    If (myrank == 0) printf (“spent %e sec\n”, finish - start );
```

Modificação do executável

Pode ser feita por instrumentação *off-line*, como fazem *prof* e similares

Ou por instrumentação dinâmica, como faz o *Paradyn* (através do *Dyninst*)

Também pode ser feito por extração de características do programa, que então são simuladas *off-line*

Problemas de benchmarks

Definição da carga de trabalho

Qual será a carga, como ela será aplicada ao sistema, que padrão estatístico ela terá?

Definição dos padrões de medida

O que será medido, qual o arranjo de memórias, processadores, rede de comunicação, compiladores, etc.

E o que fazer com as medidas?

Com as medidas em mãos passa-se para a fase de otimização.

Para essa fase podem ser usadas algumas ferramentas automáticas e, principalmente, alterações de estilo de programação.

Gargalos de software

Os pontos críticos para a melhoria de desempenho são chamados gargalos.

Podem ser de três tipos:

- Linguagem (ou estruturas utilizadas)
- Funções
- Blocos de repetição

Gargalos de linguagem

A escolha da linguagem de programação deve levar em consideração:

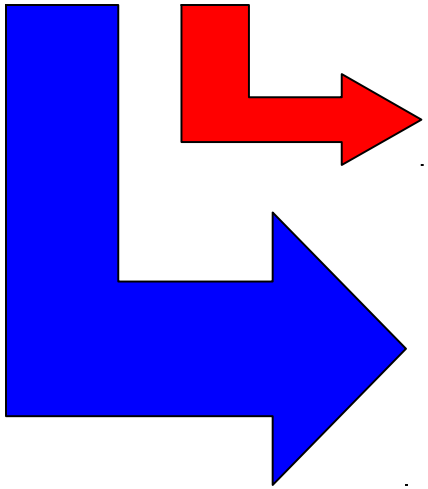
Favorecimento de estruturas simples e estáticas

Não uso de ponteiros e estruturas dinâmicas

Gargalos de linguagem

Um exemplo:

Montagem de um grafo de +/- 70 mil vértices



2 horas com ponteiros

Menos que 30 segundos com vetores

Gargalos em funções

Programação estruturada é elegante e deve ser sempre buscada.

entretanto.....

Sobrecarrega o sistema com excessos de modularização e estruturação do código fonte

Gargalos em funções

Para obter desempenho deve-se evitar o uso de funções curtas (muito antigamente economizava-se memória com elas).

Deve-se usar alinhamento de funções (*function inlining*) no lugar de suas chamadas.

Gargalos em blocos de repetição

São a maior fonte para otimizações, envolvendo:

- Desdobramento de código
- Remoção de testes desnecessários
- Inversão de aninhamentos
- Fissão ou fusão de laços

Desdobramento de laços

Desdobramentos

```
DO I=1,N
    A(I) = A(I)+ B(I)* C
ENDDO
```

```
DO I=1,N,4
    K=I+1
  ….
    A(I) = A(I)+ B(I)* C
    A(k) = A(k)+ B(k)* C
    A(L) = A(L)+ B(L)* C
    A(M) = A(M)+ B(M)* C
ENDDO
```

Condições de paralelismo

Embora a solução para a computação de alto desempenho seja o paralelismo, nem tudo pode ser paralelizado e mesmo quando isso é possível, existem restrições na forma em que isso ocorre.

Essas restrições estão bem definidas através das chamadas condições de paralelismo.

Condições de paralelismo

São três:

Dependência de dados

(ou Condições de Bernstein)

Dependência de controle

Dependência de recursos

Dependência de recursos

É bastante natural, restringindo o paralelismo ao volume de recursos que podem ser simultaneamente utilizados.

Trata de recursos como registradores, memória, canais de conexão, acesso a arquivos, etc.

Dependência de controle

Ocorre quando as instruções que devem ser executadas (e portanto paralelizadas) dependem de resultados que serão conhecidos apenas em tem po de execução

Não se pode contornar essa dependência quando ela ocorre.

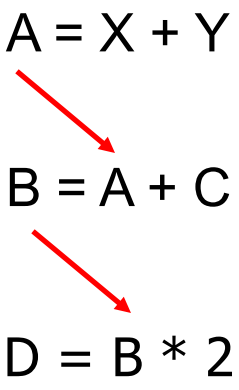
Dependência de dados

Assume três formas:

- Fluxo
- Antidependência
- Saída

Dependência de fluxo de dados

Ocorre quando o resultado (saída) de um grupo de instruções (S1) é necessário (entrada) para a execução de um segundo grupo de instruções (S2)



Antidependência

Ocorre quando a saída de um grupo de instruções (S2) for entrada para um grupo de instruções (S1), que o anteceda numa execução sequencial.

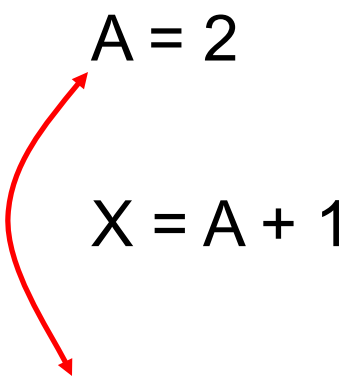
A = 2

X = A + 1

A = 5

Dependência de saída de dados

Ocorre quando uma das saídas de dois grupos de instruções (S1 e S2) forem comuns.



Condições de Bernstein

Formalizam as dependências de fluxo de dados.

Algumas definições:

Processo $\Rightarrow$ fragmento de um programa

Conjunto de entrada (I_i) $\Rightarrow$ entradas necessárias para executar P_i

Conjunto de saída (O_i) $\Rightarrow$ saídas produzidas pela execução de P_i

Condições de Bernstein

Dois processos podem executar em paralelo (Pi // Pk ; i<k) se:

$I_i \cap O_k = \varnothing \Rightarrow$ antidependência

$I_k \cap O_i = \varnothing \Rightarrow$ fluxo

$O_i \cap O_k = \varnothing \Rightarrow$ saída

Condições de Bernstein

Ampliando-se tais condições para *n* processos temos:

$P_1 // P_2 // P_3 // ... // P_n$ se e somente se:

$P_i // P_j$ para todo $i,j = 1...n$ e $i \neq j$

Voltando ao nosso exemplo

O que pode ser modificado para melhorar sua execução?

Como será mais conveniente particionar os elementos da matriz?

Qual seria o número adequado de processos em paralelo?

Isso muda se mudarmos o grau de discretização da chapa?

Voltando ao nosso exemplo

E o que acontece se precisarmos de mais precisão nos resultados?

Como, por exemplo, calcular π com 500 dígitos de precisão?