Computação Paralela e Distribuída

Profa. Cristina Boeres e Profa. Lúcia Drummond

Rio de Janeiro - março de 2007

Em convênio com:

IC – Instituto de Computação
UFF – Universidade Federal Fluminense

Introdução e Conceitos Básicos

- Por que computação paralela e distribuída
- Computação de Alto Desempenho
- Arquitetura de computadores
- Ambientes de programação paralela
- Modelos de programação paralela

Por que computação paralela e distribuída?

- Sistemas de computadores seqüenciais cada vez mais velozes
  - velocidade de processador
  - memória
  - comunicação com o mundo externo

- Quanto mais se tem, mais se quer.....
  - Demanda computacional está aumentando cada vez mais: visualização, base de dados distribuída, simulações, etc.

- limites em processamento seqüencial
  - velocidade da luz, termodinâmica

Por que computação paralela e distribuída?

- que tal utilizar vários processadores?

- dificuldades encontradas
  - mas como?
  - paralelizar uma solução?

Existem vários desafios em Computação Paralela e Distribuída

Computação de Alto Desempenho

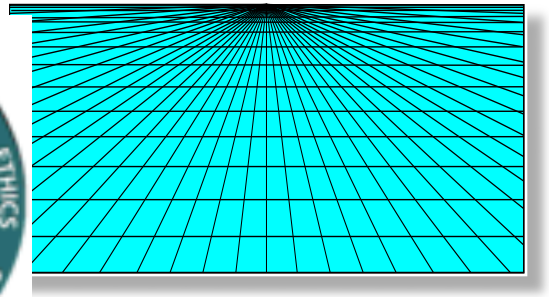
Os *grandes desafios* (Levin 1989):

- química quântica, mecânica estatística e física relativista;
- cosmologia e astrofísica;
- dinâmica e turbulência computacional dos fluídos;
- projeto de materiais e supercondutividade;
- biologia, farmacologia, seqüência de genomas, engenharia genética, dobramento de proteínas, atividade enzimática e modelagem de células;
- medicina, modelagem de órgãos e ossos humanos;
- clima global e modelagem do ambiente

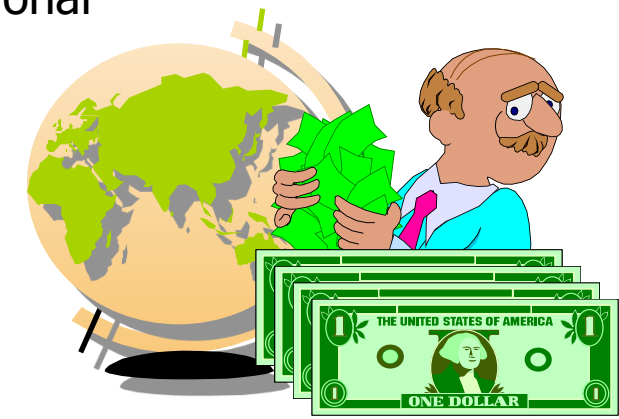
Computação de Alto Desempenho

- utilizando modelagem, simulação e análise computacional

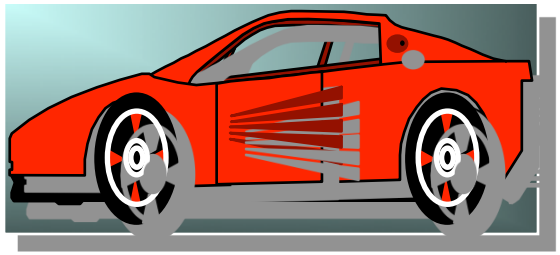
Life Sciences



Aerospace



Internet & Ecommerce



CAD/CAM

Digital Biology

Military Applications

cortesia de RajKumar Buyya

Definindo melhor alguns conceitos

- **Concorrência**
  - termo mais geral, um programa pode ser constituído por mais de um thread/ processo concorrendo por recursos

- **Paralelismo**
  - uma aplicação é executada por um conjunto de processadores em um ambiente único (dedicados)

- **Computação distribuída**
  - aplicações sendo executadas em plataformas distribuídas

Definindo melhor alguns conceitos

Qualquer que seja o conceito, o que queremos?

- estabelecer a solução do problema
- lidar com recursos independentes
- aumentar desempenho e capacidade de memória
- fazer com que usuários e computadores trabalhem em espírito de colaboração

O que paralelizar?

- Concorrência pode estar em diferentes níveis de sistemas computacionais atuais
  - hardware
  - Sistema Operacional
  - Aplicação

- As principais questões que são focadas são
  - Desempenho
  - Corretude
  - possibilidade de explorar o paralelismo

Por que paralelizar?

Aplicação Paralela

- várias tarefas

- vários processadores
  - redução no tempo total de execução

Modelos de Programação Paralela

- Criação e gerenciamento de processos
  - estático ou dinâmico

- Comunicação
  - memória compartilhada
    - visão de um único espaço de endereçamento global
  - memória distribuída
    - troca explícita de mensagens

Modelos de Programação Paralela

- Expressão de Paralelismo: Paradigmas
  - SPMD
  - MPMD

- Metas
  - aumento no desempenho
  - maior eficiência

Objetivos

- Visão geral
  - arquitetura de computadores
  - ambientes de programação paralela
  - modelos de programação paralela

- Motivar $\Rightarrow$ Sistemas de Alto Desempenho

Arquitetura de Computadores

Classificação de Computadores

- Computadores Convencionais
- Memória Centralizada
- Memória Distribuída

Arquitetura de Computadores

- Sistema Paralelo
  - vários processadores
  - vários módulos de memória
  - comunicação: estruturas de interconexão

Plataforma de Execução Paralela

Conectividade ⇒ rede de interconexão

Heterogeneidade ⇒ hardware e software distintos

Compartilhamento ⇒ utilização de recursos

Imagem do sistema ⇒ como usuário o percebe

Escalabilidade ⇒ + nós > desempenho/eficiência

Classificação de Sistemas Paralelos

- Proposta por Flynn
  - quantidade de instruções e dados processados em um determinado momento

SISD (single instruction single data)
- Um contador de programa
- Computadores seqüenciais

SIMD (single instruction multiple data)
- Um contador de programa, uma instrução executada por diversos processadores sobre diferentes dados
- Computadores

Classificação de Sistemas Paralelos

- Proposta por Flynn

MISD (multiple instructions single data)

- Não aplicável

MIMD (multiple instructions multiple data)

- Vários contadores de programa
- Diferentes dados
- Os vários computadores paralelos e distribuídos atuais

Plataforma de Execução Paralela

- Diferentes plataformas do **MIMD** de acordo com os seguintes critérios
  - espaço de endereçamento
  - mecanismo de comunicação

- Podem ser agrupadas em quatro grupos

SMPs (*Symmetric MultiProcessors*)

MPPs (*Massively Parallel Processors*)

Cluster ou NOW*s* (*Network Of Worstations*)

Grades Computacionais

SMPs

- SMPs ou Multiprocessadores

  - único espaço de endereçamento lógico
    - mecanismo de hardware (memória centralizada)

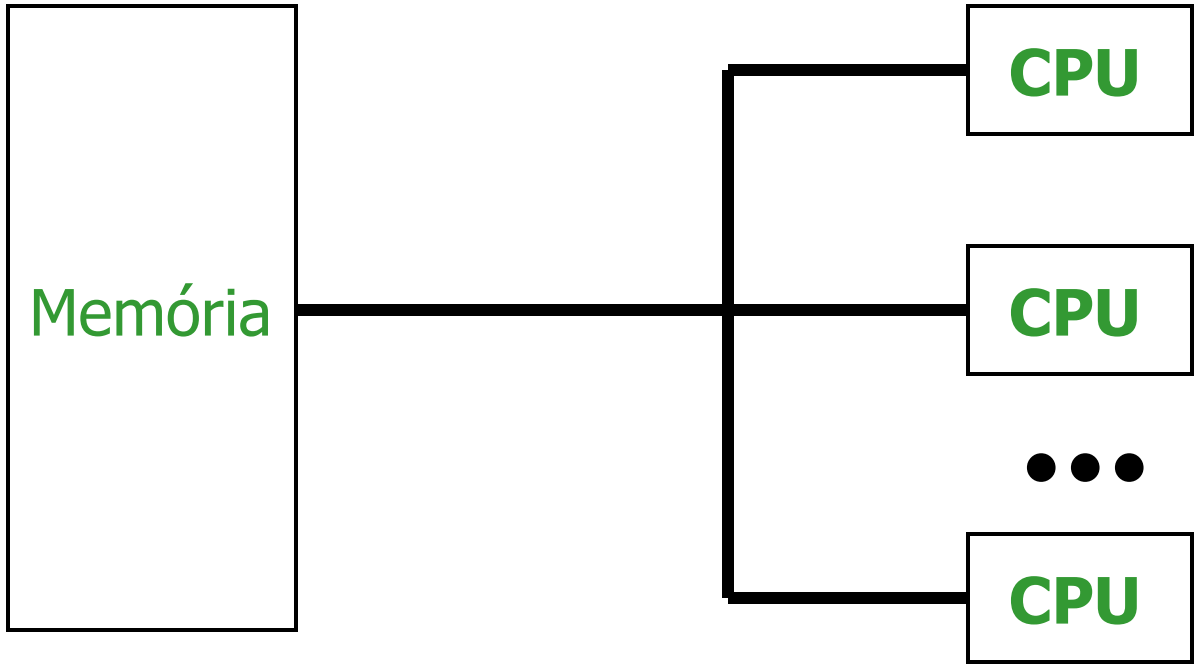
  - comunicação ⇒ espaço de endereçamento compartilhado
    - operações de *loads* e *stores*
      - Acesso a memória é realizada através de leitura (load) e escrita (store), caracterizando desta forma, a comunicação entre processadores

SMPs

- Sistema homogêneo
- Compartilhamento
  - Compartilhamento total da mesma memória
- Uma única cópia do Sistema Operacional
- Imagem única do sistema
- Excelente conectividade
  - fortemente acoplados
- Não escalável
- Exemplos:
  - Sun HPC 10000 (StarFire), SGI Altix, SGI Origin, IBM pSeries, Compac AlphaServer

SMPs

Multiprocessadores



MPPs (Multicomputadores)

- Diferem quanto a implementação física
- Módulos ou elementos de processamento contendo:
  - múltiplos processadores com memória privativa
  - computadores completos
- Espaço de endereçamento
  - não compartilhado - memória distribuída
- Comunicação
  - troca de mensagens
- Rede de interconexão
  - diferentes topologias
- Fracamente acoplados
- Escaláveis

MPPs

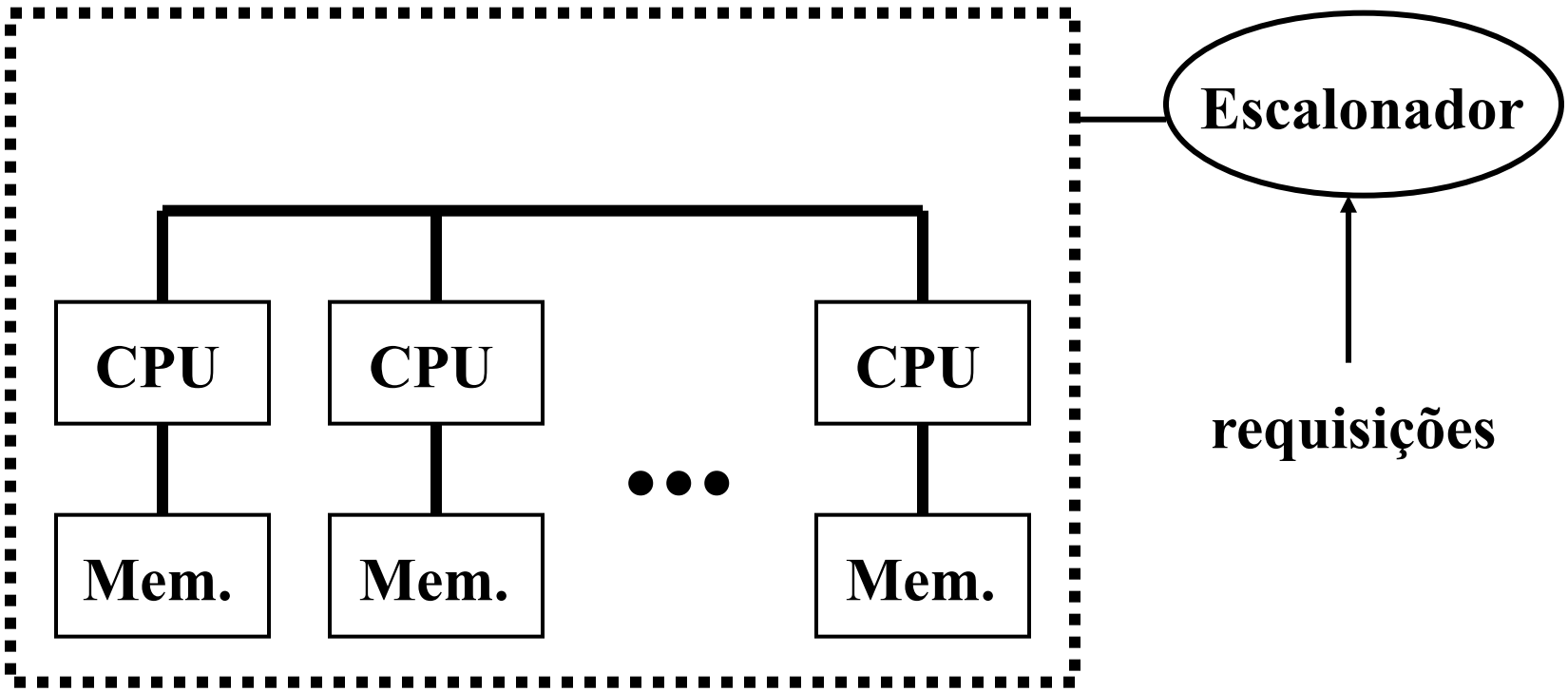
- Sistema homogêneo ou heterogêneo
- Interconexão: redes dedicadas e rápidas
- Cada nó executa sua própria cópia do Sistema Operacional
- Imagem única do sistema
  - visibilidade dos mesmos sistemas de arquivo
- Um escalonador de tarefas
  - partições diferentes para aplicações diferentes

MPPs

- Partições dedicadas a cada aplicação
- Aplicações não compartilham recursos
  - Pode ocorrer que uma aplicação permaneça em estado de espera
- Exemplos:
  - Cray T3E, IBM SP2s, *clusters* montados pelo próprio usuário, com propósito de ser um MPP

MPPs

- **Multicomputadores**



Cluster de computadores ou NOWs

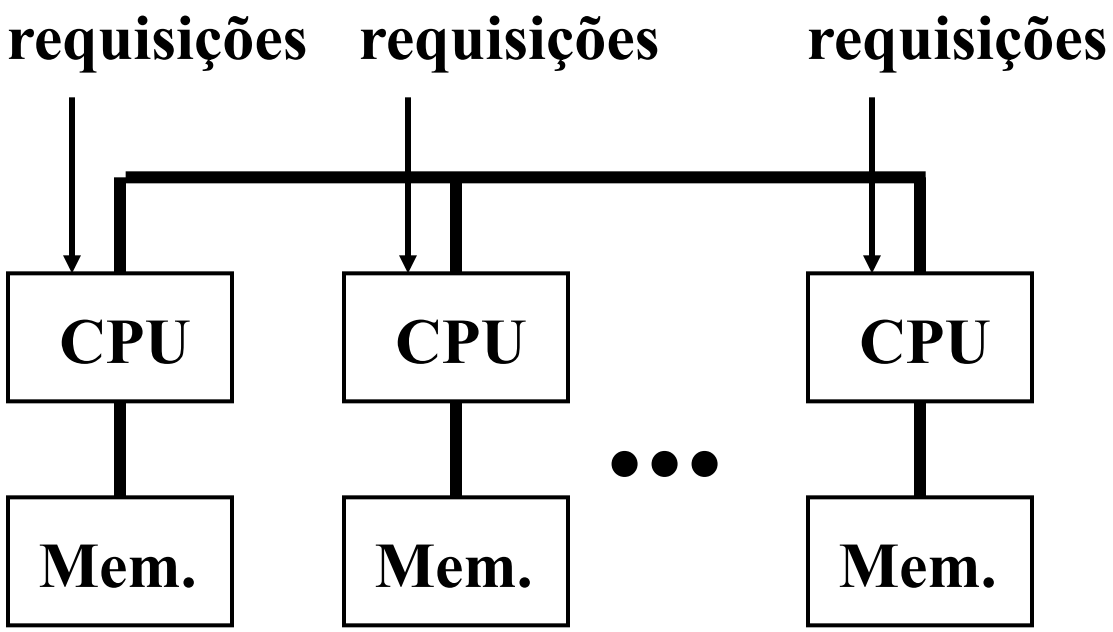
- Conjunto de estações de trabalho ou PCs
- Interconexão: redes locais
- Nós: elementos de processamento = processador + memória
- Diferenças em relação a MPPs:
  - não existe um escalonador centralizado
  - redes de interconexão tendem a ser mais lentas

Cluster de computadores ou NOWs

- Resultado das diferenças:
  - Cada nó tem seu próprio escalonador local
  - Compartilhamento de recursos ⇒ sem partição dedicada a uma aplicação
  - Aplicação ⇒ deve considerar impacto no desempenho
    ⇒ não tem o sistema dedicado

  - Possibilidade de compor um sistema de alto desempenho e um baixo custo (principalmente quando comparados com MPPs).

Cluster ou NOWs



Grades Computacionais (*Computational Grids*)

- Utilização de computadores
  - independentes
  - geograficamente distantes

- Diferenças: *clusters* X grades

  - heterogeneidade de recursos
  - alta dispersão geográfica (escala mundial)
  - compartilhamento
  - múltiplos domínios administrativos
  - controle totalmente distribuído

Grades Computacionais

- Componentes
  - PCs, SMPs, MPPs, *clusters*
  - controlados por diferentes entidades ⇒ diversos domínios administrativos

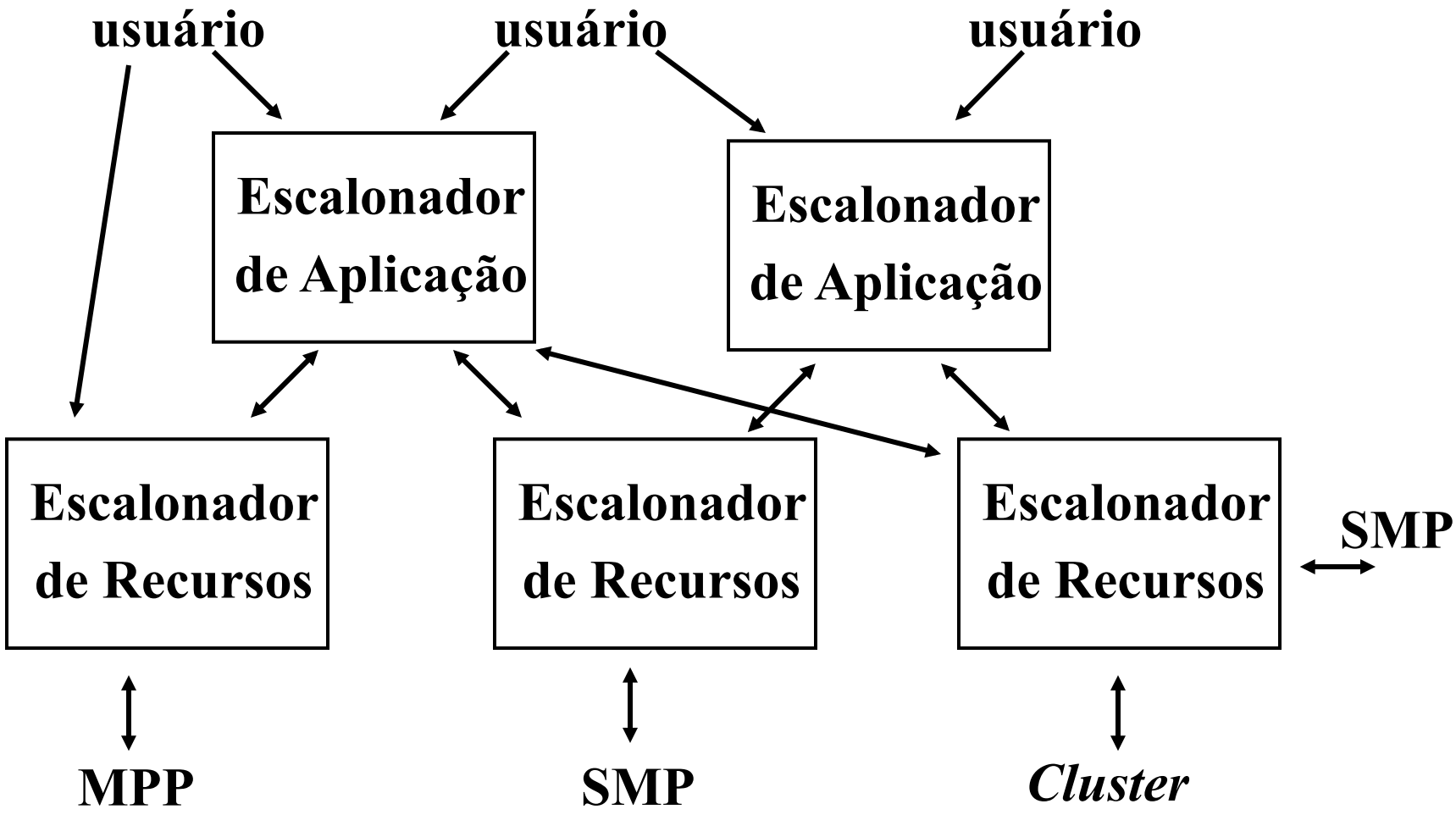
- Não têm uma imagem única do sistema a princípio
  - Vários projetos tem proposto o desenvolvimento de middlewares de gerenciamento ⇒ camada entre a infra-estrutura e as aplicações a serem executadas na grade computacional

- Aplicação deve estar preparada para:
  - Dinamismo
  - Variedade de plataformas
  - Tolerar falhas

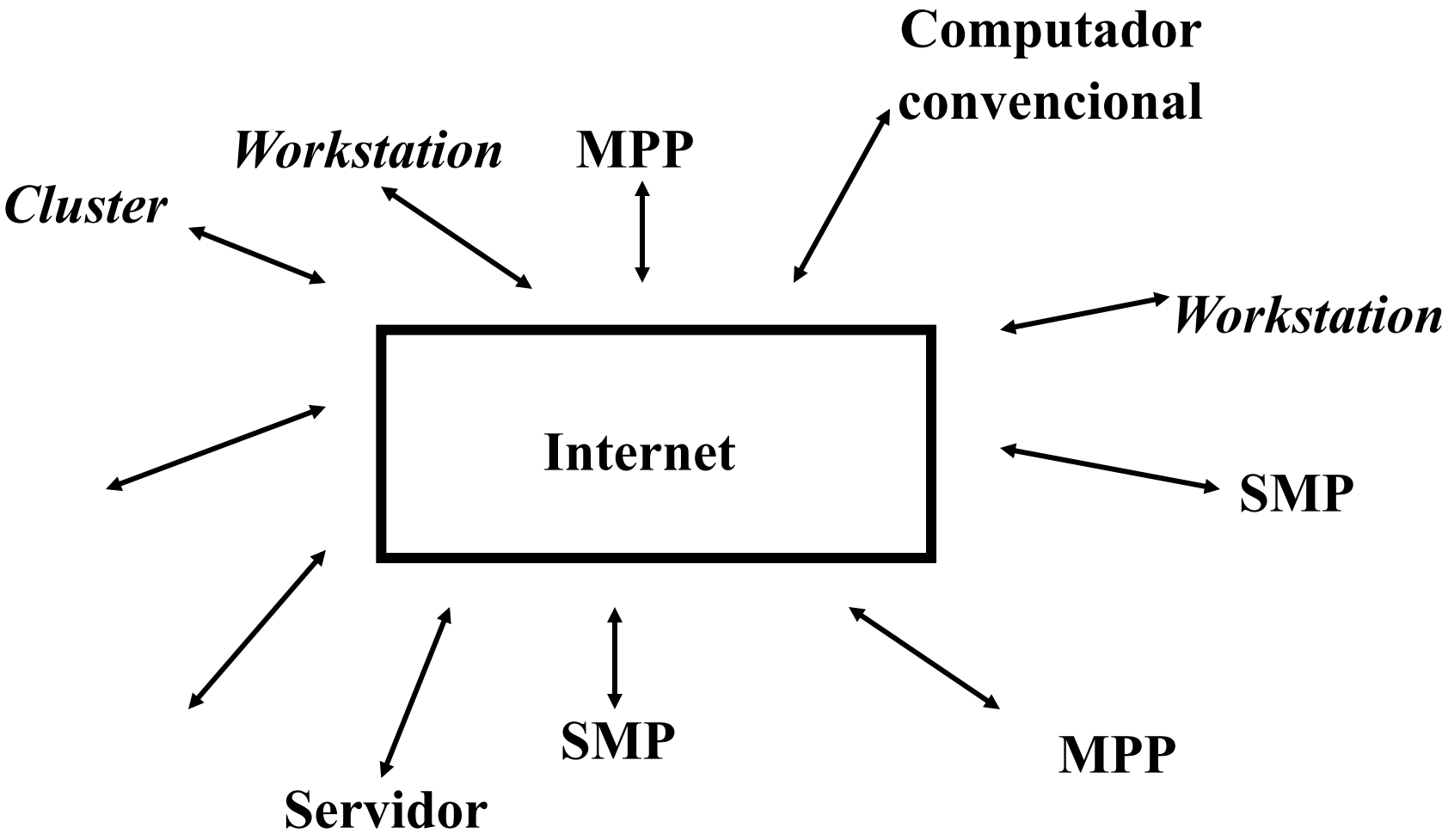
Grades Computacionais

- Sistema não dedicado e diferentes plataformas
  - Usuários da grades devem obter autorização e certificação para acesso aos recursos disponíveis na grade computacional
- Falhas nos recursos tanto de processamento como comunicação são mais freqüentes que as outras plataformas paralelas
  - Mecanismos de tolerância a falhas devem tornar essas flutuações do ambiente transparente ao usuário
- Para utilização eficiente da grade computacional
  - Gerenciamento da execução da aplicação através de políticas de escalonamento da aplicação ou balanceamento de carga
  - Escalonamento durante a execução da aplicação se faz necessário devido as variações de carga dos recursos da grade

Grades Computacionais



Grades Computacionais



Resumo

- **Plataformas de Execução Paralela**

Características	SMPs	MPPs	NOWs	Grids
Conectividade	excelente	muito boa	boa	média/ruim
Heterogeneidade	nula	baixa	média	alta
Compartilhamento	não	não	sim	sim
Imagem do Sistema	única	comum	comum	múltipla
Escalabilidade	10	1.000	1.000	100.000

Top500 Supercomputer (atualizada)

	Site	Computer	Procs	Year	R_{max}	R_{peak}
1	DOE/NNSA/LLNL United States	BlueGene/L - eServer Blue Gene Solution IBM	131072	2005	280600	367000
2	NNSA/Sandia National Laboratories United States	Red Storm - Sandia/ Cray Red Storm, Opteron 2.4 GHz dual Cray Inc.	26544	2006	101400	127411
3	IBM Thomas J. Watson Research Center United States	BGW - eServer Blue Gene Solution IBM	40960	2005	91290	114688
4	DOE/NNSA/LLNL United States	ASC Purple - eServer pSeries p5 575 1.9 GHz IBM	12208	2006	75760	92781
5	Barcelona Supercomputing Center Spain	MareNostrum - BladeCenter JS21 Cluster, PPC 970, 2.3 GHz, Myrinet IBM	10240	2006	62630	94208
6	NNSA/Sandia National Laboratories United States	Thunderbird - PowerEdge 1850, 3.6 GHz, Infiniband Dell	9024	2006	53000	64972.8
7	Commissariat a l'Energie Atomique (CEA) France	Tera00 - NovaScale 5160, Itanium2 1.6 GHz, Quadrics Bull SA	9968	2006	52840	63795.2
8	NASA/Ames Research Center/NAS United States	Columbia - SGI Altix 1.5 GHz, Voltaire Infiniband SGI	10160	2004	51870	60960
9	GSIC Center, Tokyo Institute of Technology Japan	STSUBAME Grid Cluster - Sun Fire x4600 Cluster, Opteron 2.4/2.6 GHz and ClearSpeed Accelerator, Infiniband NEC/Sun	11088	2006	47380	82124.8
10	Oak Ridge National Laboratory United States	Jaguar - Cray XT3, 2.4 GHz Cray Inc.	10424	2006	43480	54204.8

Rmax Maximal LINPACK performance achieved **Rpeak** Theoretical peak performance GFlpos

Top500 Supercomputer (Máquinas Brasileiras)

	Site	Computer	Procs	Year	R_{max}	R_{peak}
273	Petroleum Company (C) Brazil	xSeries Cluster Xeon 3.06 GHz - Gig-E IBM	1024	2004	3755	6266.88
275	PETROBRAS Brazil	Rbwr1 Cluster platform 3000 DL140G3 Xeon 3.06 GHz GigEthernet Hewlett-Packard	1300	2004	3739	7956
363	University of San Paulo Brazil	BladeCenter JS21 Cluster, PPC970, 2.5 GHz, Myrinet IBM	448	2006	3182.38	4480
418	PETROBRAS Brazil	bw7 - Cluster platform 3000 DL140G3 Xeon 3.06 GHz GigEthernet Hewlett-Packard	1008	2004	2992	6169

Rmax Maximal LINPACK performance achieved **Rpeak** Theoretical peak performance GFlpos

Computação em Cluster

- Um conjunto de computadores (PCs)
- não necessariamente iguais → heterogeneidade
- Filosofia de imagem única
- Conectadas por uma rede local

Para atingir tais objetivos, necessidade de uma camada de software ou *middleware*

Computação em Grid

- Computação em Cluster foi estendido para computação ao longo dos sites distribuídos geograficamente conectados por redes metropolitanas

Grid Computing

- Heterogêneos
- Compartilhados
- Aspectos que devem ser tratados
- Segurança
- Falhas de recursos
- Gerenciamento da execução de várias aplicações

Computação em Grid

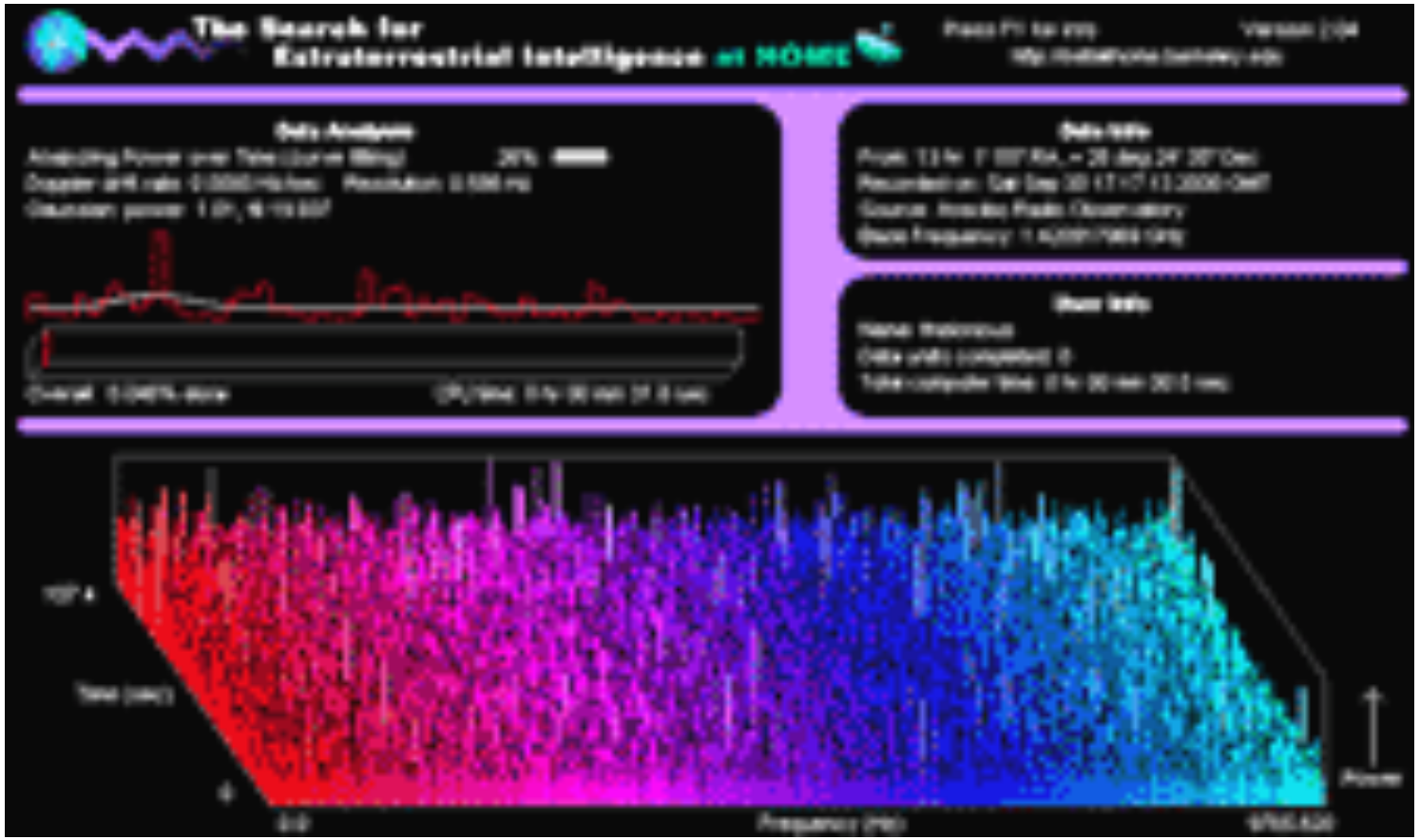
O sonho do cientista (The Grid Vision)

- Computação em Grid adota tanto o nome quanto o conceito semelhantes aqueles da Rede de Potência Elétrica para capturar a noção ou a visão de:
  - Oferecer desempenho computacional eficientemente;
  - De acordo com a demanda;
  - A um custo razoável;
  - Para qualquer um que precisar.

- O sucesso da computação em grid depende da comunidade de pesquisadores
  - A possibilidade de construir tal ambiente (hardware e software)
  - Necessidade de atingir seus objetivos.

Computação em Grid

SETI@home: Search for Extraterrestrial Intelligence at Home



Computação em Grid

- *Grid middlewares*: tem como objetivo facilitar a utilização de um ambiente grid
  - APIs para isolar usuários ou programas da complexidade deste ambiente
  - Gerenciar esses sistemas automaticamente e eficientemente para executar aplicações no ambiente grid (grid-enabled applications)

E as aplicações não habilitadas a execução em ambiente grids?

Computação em Grid

Como o usuário (dono da aplicação) escolhe?

- Vários *middlewares* existem, qual o mais apropriado?
- Vários estão ainda sendo desenvolvidos
- Não há a garantia de suporte
- Pouca comparação entre os *middlewares*, por exemplo, desempenho, grau de intrusão.
- É difícil encontrar grids com o mesmo tipo de software instalado

Programação Distribuída

Introdução e Conceitos Básicos

Conceitos Básicos

- Programa Distribuído: Conjunto de processos que trabalham em conjunto para solucionar um problema e executam em sistemas distribuídos

- Sistema Distribuído:
  - Não há compartilhamento de memória
  - Troca de informação através de troca de mensagens

Conceitos Básicos

- Áreas de demanda:
  - Otimização combinatória
  - Mineração de dados
  - Simulações
  - Meteorologia
  - Bioinformática
  - Computação gráfica

- Computação intensiva !!

Conceitos Básicos

Seqüencial ≠ Distribuído **- Determinação de estado não é trivial**

Seqüencial

if (x=1) then

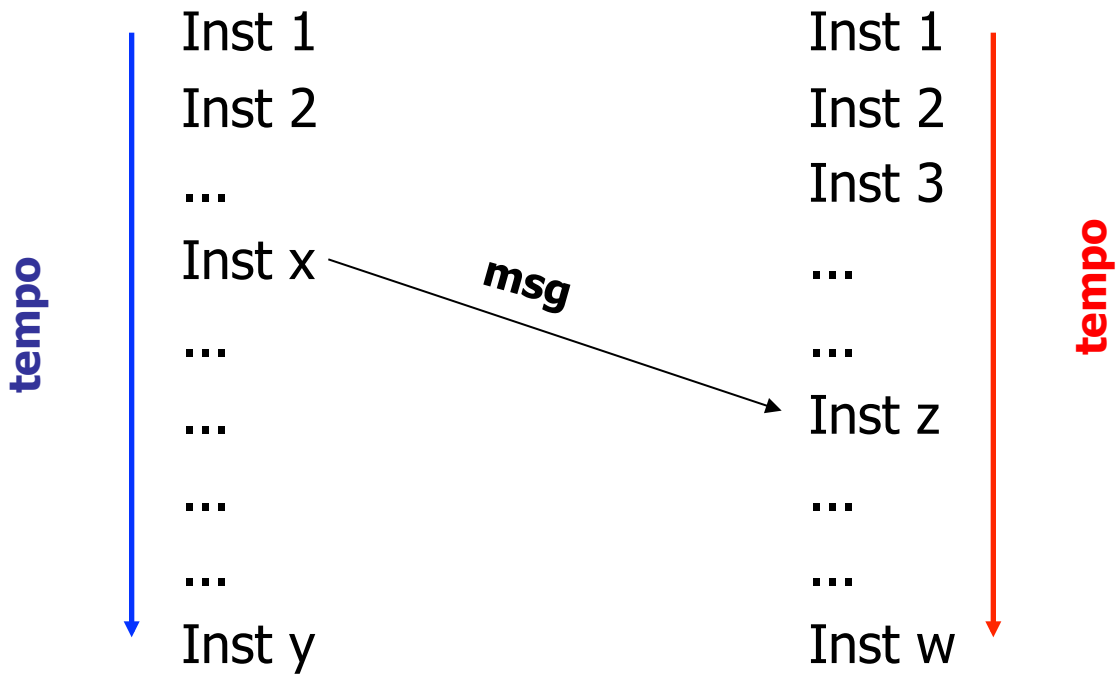
Distribuído

if ($x_{\text{proc a}}$ = 1) and
($y_{\text{proc b}}$ = 2) then

Não é trivial !!!

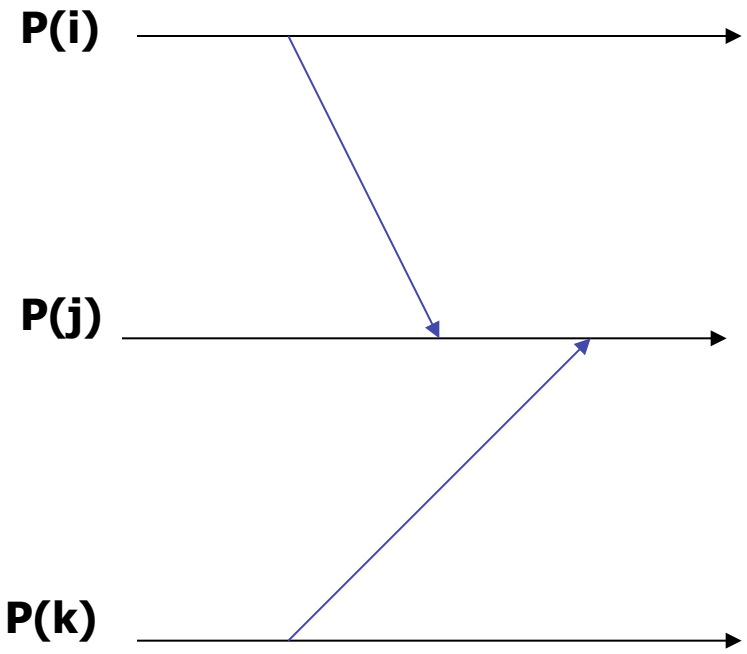
Conceitos Básicos

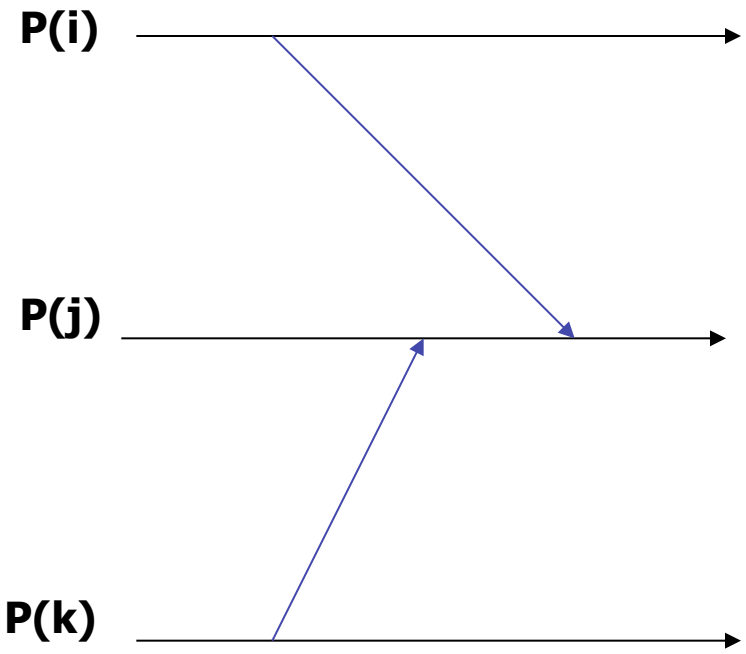
Seqüencial ≠ Distribuído - **Ausência de uma base de tempo global**



Conceitos Básicos

Distribuído ≠ Centralizado - **Não determinístico**





Modelos de Computação

- Assíncrono:
  - Sem base de tempo global
  - Atraso na transmissão das mensagens é finito mas não determinado
  - Desenvolvimento mais complexo
  - Realístico

- Síncrono:
  - Com base de tempo global
  - Comunicação em "pulsos"
  - Desenvolvimento mais simples
  - Menos realístico

Modelo utilizado

- Um programa paralelo distribuído é representado por um grafo não orientado **G = (N,E)** onde:
  - N : conjunto de nós que representam processos
  - n = |N|
  - E : conjunto de arestas que representam canais de comunicação
  - e = |E|

- Para i = 1, 2, 3, ..., n, $p_i \in N$ é um processo que pode se comunicar exclusivamente por troca de mensagens com os processos p_j tal que $(p_i, p_j) \in E$

- Canais de comunicação bidirecionais de capacidade infinita

Avaliação

- CORRETUDE
- Complexidade de mensagens: Número/Tamanho das mensagens enviadas
- Complexidade de tempo:
  - Síncrono: Número de pulsos
  - Assíncrono: Maior cadeia de mensagens com relação de "causalidade"

Introdução ao MPI

Introdução

- O MPI é um padrão para desenvolvimento de aplicações distribuídas
- Disponível na forma de bibliotecas para linguagens C, C++ e Fortran
- SPMD – *Single Program Multiple Data* – todos os processos executam o MESMO programa
- Síntese de diversos sistemas anteriores

Histórico Breve

- Desenvolvido por um fórum aberto internacional composto por representantes da indústria, universidade e entidades governamentais
- Influenciado por outras plataformas e comunidades: Zipcode, Chimp, PVM, Chamaleon e PICL
- Padronização iniciada em abril de 1992

Objetivos

- Eficiência na comunicação
- Ambientes para desenvolvimento e execução heterogêneos
- Fácil aprendizado para atuais programadores de aplicações distribuídas (interface parecida com a do PVM)

Características

- Supõe que a subcamada de comunicação é confiável
- Garante ordem de entrega das mensagens
- Trabalha com o conceito de COMUNICADORES, que definem o universo de processos envolvidos em uma operação de comunicação
- Cada processo ganha uma identificação numérica (*rank*)

Comandos de incialização e finalização

- MPI_Init : Inicializa o ambiente de execução
- MPI_Comm_rank: Determina o *rank* (identificação) do processo no comunicador
- MPI_Comm_size: Determina o número de processos no comunicador
- MPI_Finalize: Termina o ambiente de execução

Comunicação Ponto a Ponto

- O que acontece se um processo tentar receber uma mensagem que ainda não foi enviada?

  - Função BLOQUEANTE de recebimento (MPI_Recv) : bloqueia a aplicação até que o buffer de recepção contenha a mensagem

  - Função NÃO BLOQUEANTE de recebimento (MPI_Irecv) : retorna um *handle request*, que pode ser testado ou usado para ficar em espera pela chegada da mensagem

MPI_Send

int MPI_Send(void* message, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

- **message: endereço inicial da informação a ser enviada**
- **count: número de elementos do tipo especificado a enviar**
- **datatype: MPI_CHAR, MPI_INT, MPI_FLOAT, MPI_BYTE, MPI_LONG, MPI_UNSIGNED_CHAR, etc**
- **dest: rank do processo destino**
- **tag: identificador do tipo da mensagem**
- **comm: especifica o contexto da comunicação e os processos participantes do grupo. O padrão é MPI_COMM_WORLD**

MPI_Recv

```
int MPI_Recv(void* message, int count,
    MPI_Datatype datatype, int source, int tag,
    MPI_Comm comm, MPI_Status* status)
```

- **message**: Endereço inicial do buffer de recepção
- **count**: Número máximo de elementos a serem recebidos
- **datatype**: MPI_CHAR, MPI_INT, MPI_FLOAT, MPI_BYTE, MPI_LONG, MPI_UNSIGNED_CHAR, etc.
- **source**: rank do processo origem (* = MPI_ANY_SOURCE)
- **tag**: tipo de mensagem a receber (* = MPI_ANY_TAG)
- **comm**: comunicador
- **status**: Estrutura com três campos: MPI_SOURCE, MPI_TAG, MPI_ERROR

MPI_Irecv

- Menos utilizada
- Parâmetros iguais ao bloqueante, acrescido de uma estrutura *(request)* que armazena informações que possibilitam o bloqueio posterior do processo usando a função **MPI_Wait(&request, &status)**

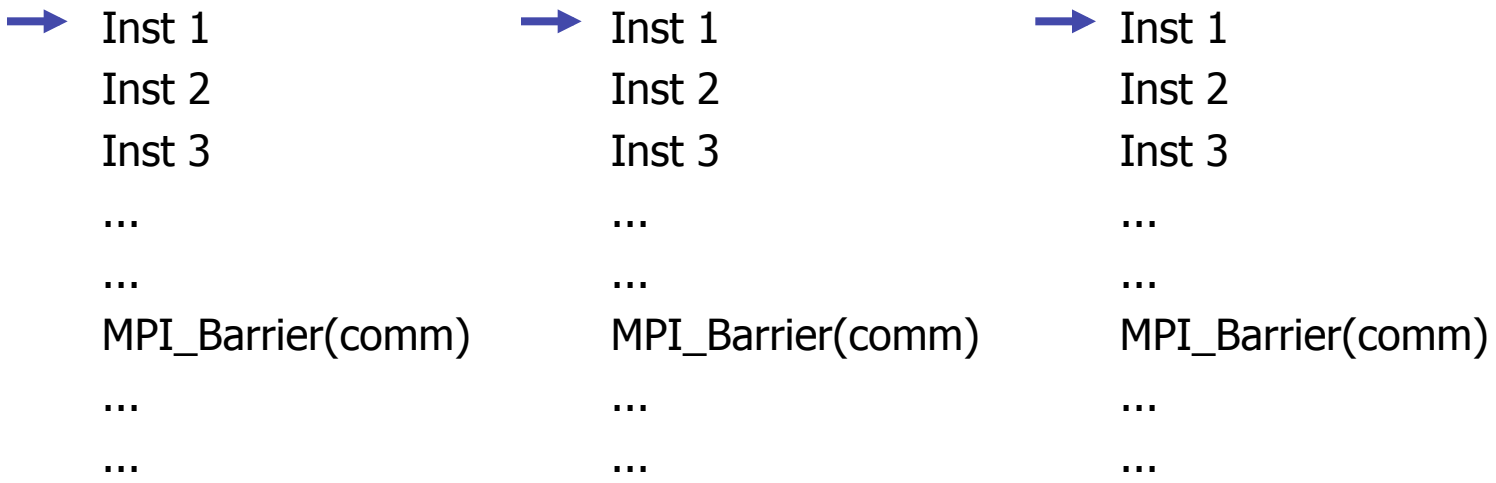
Comunicação Coletiva

- **mais restritivas que as comunicações ponto a ponto:**
  - **quantidade de dados enviados deve casar exatamente com a quantidade de dados especificada pelo receptor**
  - **Apenas a versão bloqueante das funções está disponível**
  - **O argumento *tag* não existe**

- **Todos os processos participantes da comunicação coletiva chamam a mesma função com argumentos compatíveis**

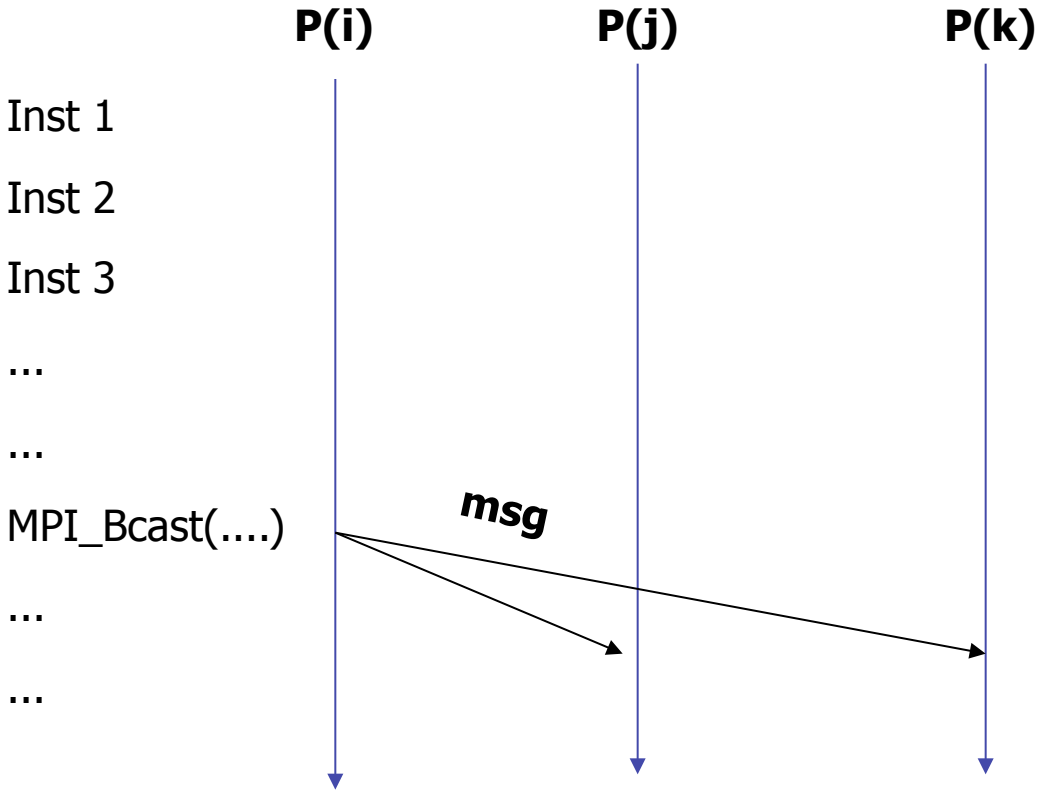
Algumas Funções para Comunicação Coletiva

- **MPI_Barrier: Bloqueia o processo até que todos os processos associados ao comunicador chamem essa função**



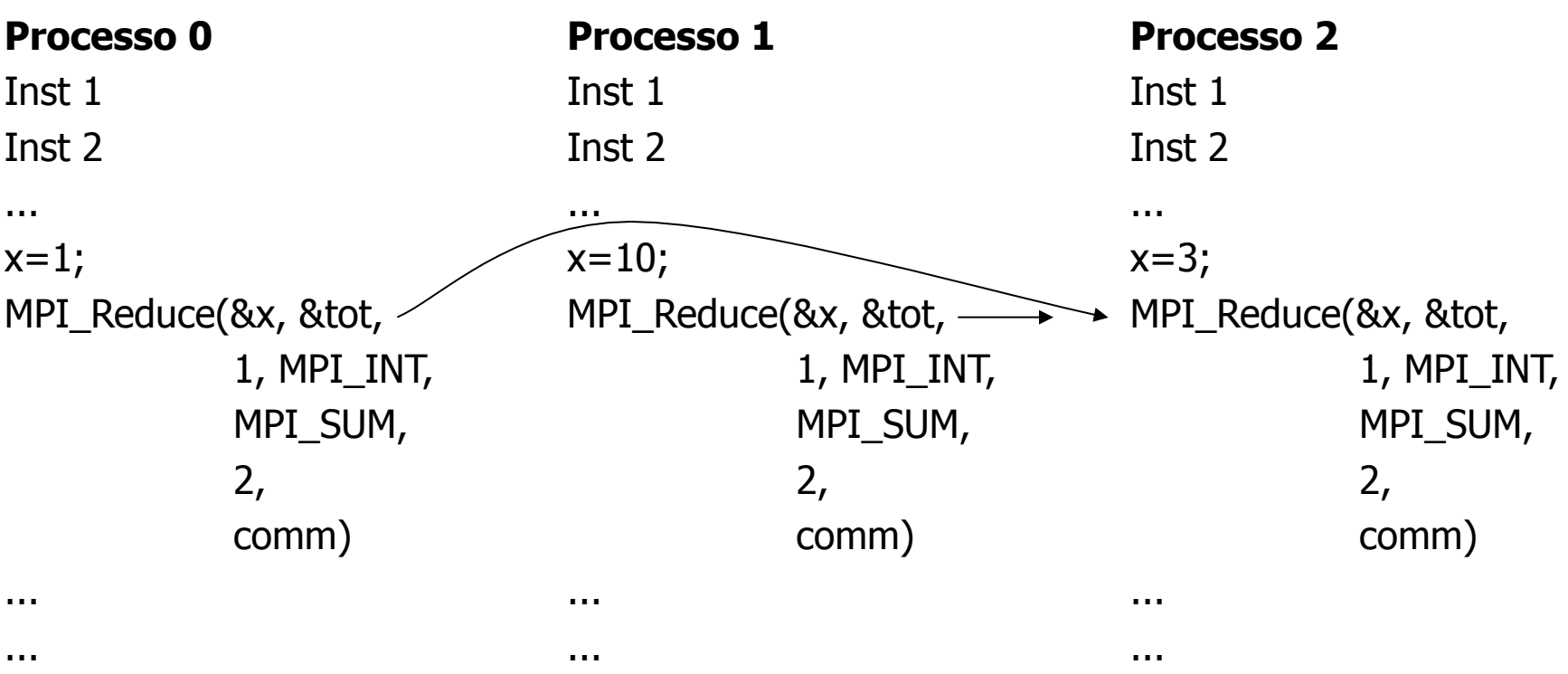
Algumas Funções para Comunicação Coletiva

- **MPI_Bcast: Faz a difusão de uma mensagem do processo raiz para todos os processos associados ao comunicador**



Algumas Funções para Comunicação Coletiva

- **MPI_Reduce: Combina todos os elementos presentes no buffer de cada processo do grupo usando a operação definida como parâmetro e coloca o valor resultante no buffer do processo especificado. O exemplo abaixo soma todas as variáveis "x" armazenando o total na variável "tot" do processo 2.**



Programa Exemplo (1/3)

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

main(int argc, char** argv)
{
    int meu_rank, np, origem, destino, tag=0;
    char msg[100];
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_rank);
    MPI_Comm_size(MPI_COMM_WORLD,&np);
```

Programa Exemplo (2/3)

```
if (meu_rank != 0) {
   sprintf(msg, "Processo %d está vivo!", meu_rank);
   destino = 0;
   MPI_Send(msg,
            strlen(msg)+1,
            MPI_CHAR,
            destino,
            tag,
            MPI_COMM_WORLD);
}
```

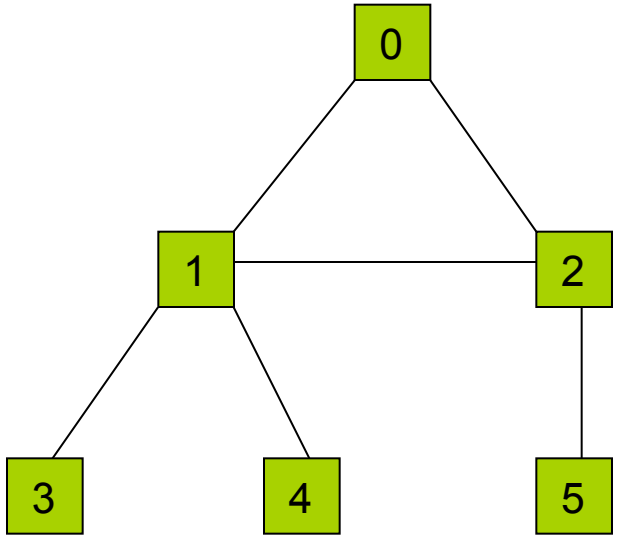
Programa Exemplo (3/3)

```
else {    // if (meu_rank == 0)
   for (origem=1; origem<np; origem++) {
      MPI_Recv(msg,
               100,
               MPI_CHAR,
               origem,
               tag,
               MPI_COMM_WORLD,
               &status);
      printf("%s\n",msg);
   }
}
MPI_Finalize( );
}
```

Alguns Algoritmos Distribuídos

- Propagação de Informação
- Propagação com Realimentação
- Integral Definida
- Conectividade em Grafos
- Distância Mínima

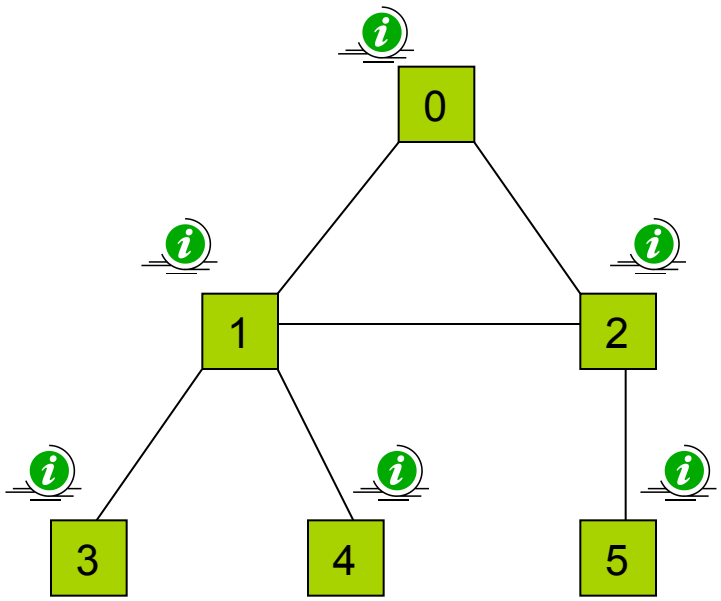
Grafo para testes




```
int matrizVizinhanca[6][6] = {
        {0,1,1,0,0,0},
        {1,0,1,1,1,0},
        {1,1,0,0,0,1},
        {0,1,0,0,0,0},
        {0,1,0,0,0,0},
        {0,0,1,0,0,0}
};
```

Propagação de Informações

Propagação de Informações

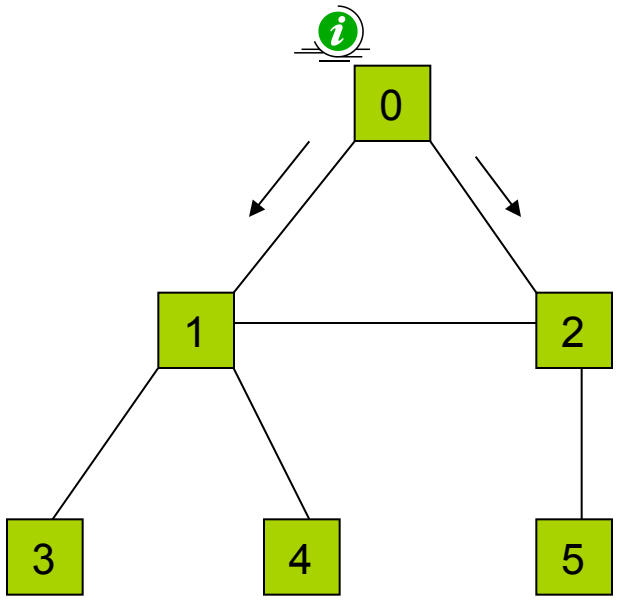


Nó 0 gera uma informação que tem de ser encaminhada a todos os demais.

Propagação de Informações - Algoritmo

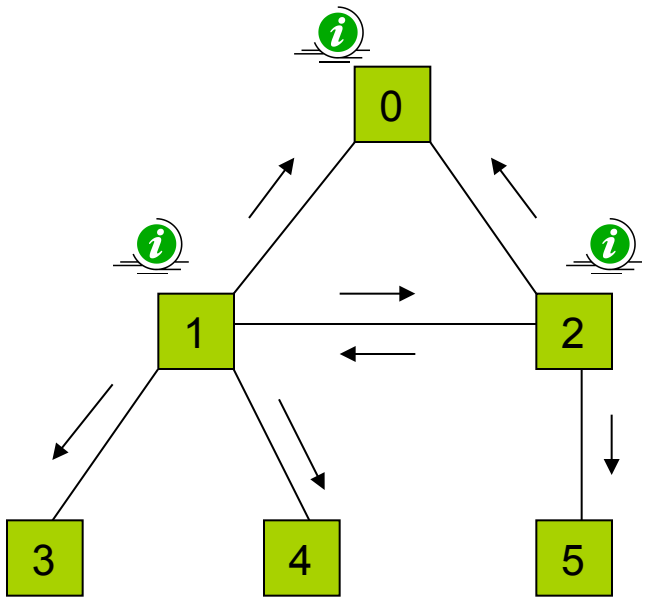
```
Variáveis:
    alcançado = falso

Ação para fonte inicial da informação (inf):
   alcançado := verdadeiro;
   envie inf a todos os vizinhos;

Ao receber inf
   se alcançado = falso
        alcançado := verdadeiro;
        envie inf a todos os vizinhos;
```

Propagação de Informações - Execução



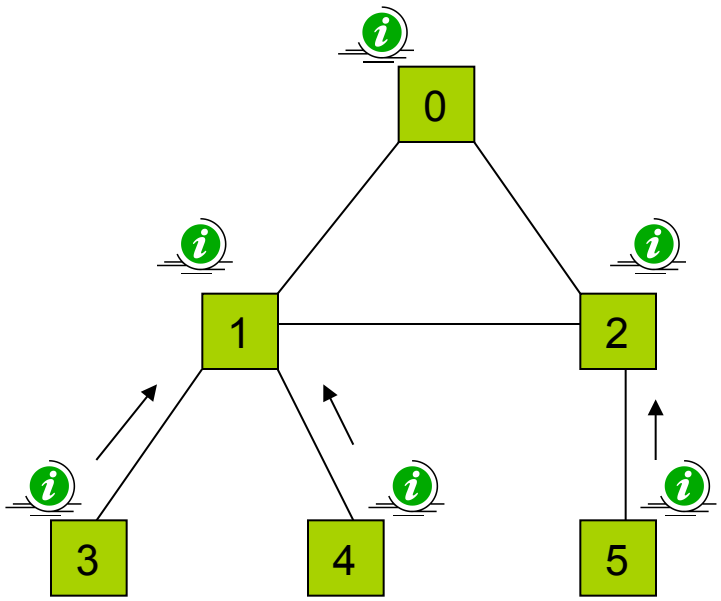
Nó 0 gera uma informação e a encaminha a todos os seus vizinhos

Propagação de Informações - Execução



Nós 1 e 2 recebem a informação e a encaminham a todos os seus vizinhos

Propagação de Informações - Execução

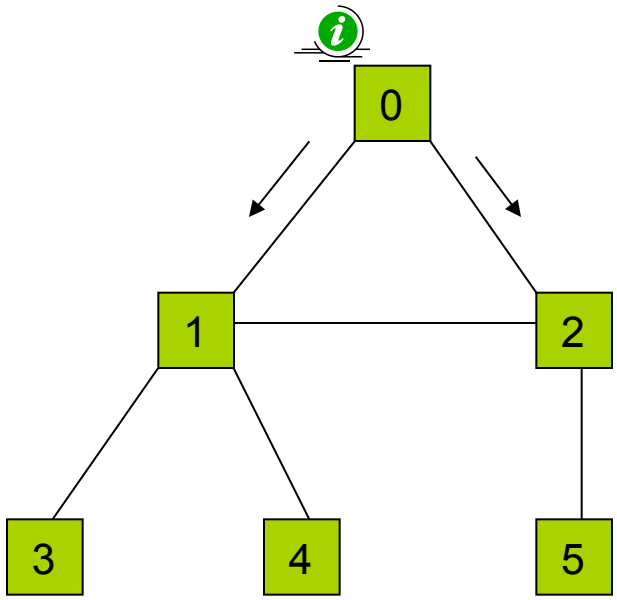


Nós 3, 4 e 5 recebem a informação e a encaminham a todos os seus vizinhos

Propagação de Informações

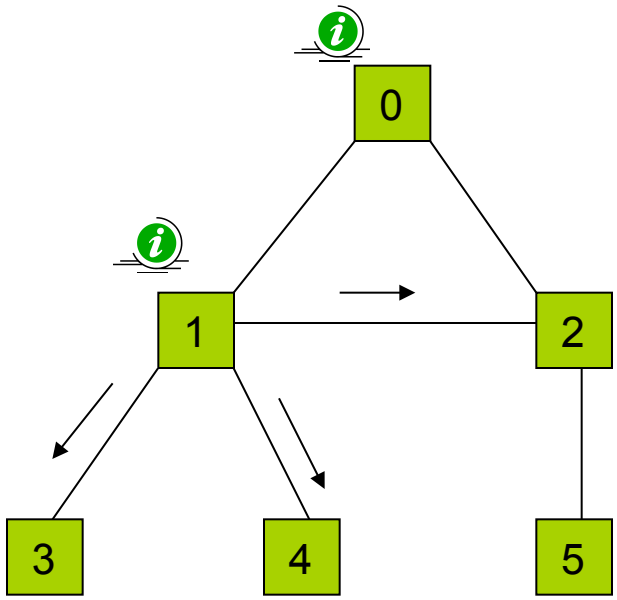
- Será possível aumentar a eficiência do algoritmo?
- Será necessária a propagação para TODOS os vizinhos?
- Inclusive o remetente da mensagem?
- Considere outra possível execução, representada nas próximas telas...

Propagação de Informações - Execução



Nó 0 gera uma informação e a encaminha a todos os seus vizinhos

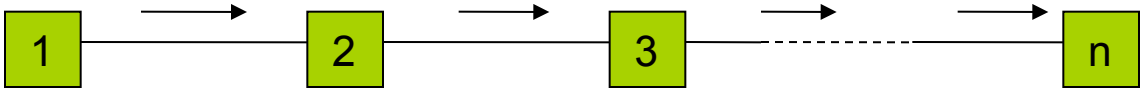
Propagação de Informações - Execução



Nó 1 recebe a informação e a encaminha a todos os seus vizinhos, exceto o nó 0, mas canal de comunicação 0 - 2 está muito lento, e 2 recebe a informação de 1 antes de 0.
Nó 0 poderá receber mensagem de 2 ou não, dependendo da ordem da chegada das mensagens!!

Propagação de Informação - Complexidades

- Mensagens: 2e => **O(e)**
- Tempo: n-1 => **O(n)**



Propagação de Informação - Implementação

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

/*Definir o grafo da aplicação antes de executar*/
int numeroDeTarefas = 6;
int matrizVizinhanca[6][6] = {   {0,1,1,0,0,0},
                                 {1,0,1,1,1,0},
                                 {1,1,0,0,0,1},
                                 {0,1,0,0,0,0},
                                 {0,1,0,0,0,0},
                                 {0,0,1,0,0,0}
                               };
```

Propagação de Informação - Implementação

```
/*retorna o número de vizinhos da tarefa myRank*/
int contaNumeroDeVizinhos(int myRank)
{
   int i;
   int contador = 0;

   for (i = 0; i < numeroDeTarefas; i++)
      if (matrizVizinhanca[myRank][i] == 1)
         contador++;

   return contador;
}
```

Propagação de Informação - Implementação

```
/*programa principal*/
int main(int argc, char** argv)
{
   int i;
   int numeroDeVizinhos;
   int myRank;
   int source;
   int tag = 50;
   char message[100] = "Oi!";
   MPI_Status status;

   //inicialização do MPI
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

   numeroDeVizinhos = contaNumeroDeVizinhos(myRank);
```

Propagação de Informação - Implementação

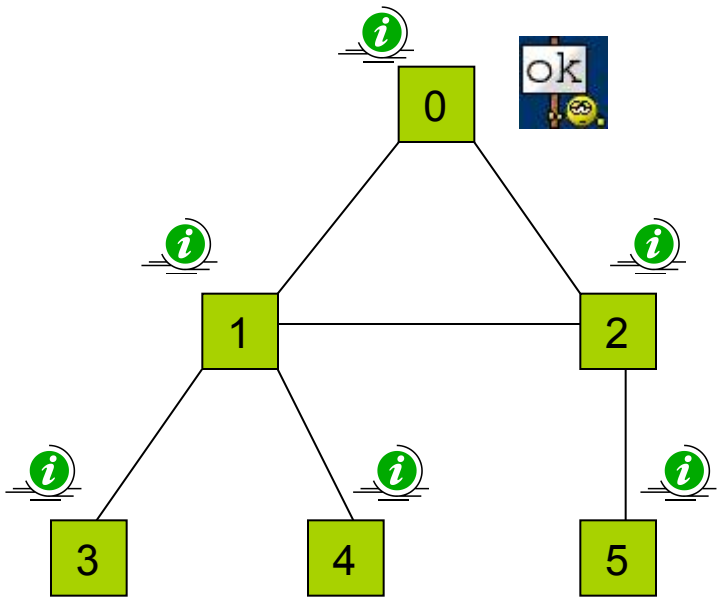
```
if (myRank == 0)       {
   /*enviando para todos os vizinhos de myRank*/
   for (i = 0; i < numeroDeTarefas; i++)
      if (matrizVizinhanca[myRank][i] == 1)
         {
            printf("Enviando mensagem para %d\n", i);
            MPI_Send(message, strlen(message)+1, MPI_CHAR, i, tag,
                     MPI_COMM_WORLD);
         }

   /*recebendo de todos os vizinhos de myRank*/
   for(i = 0; i < numeroDeVizinhos; i++)
        {
          MPI_Recv(message, 100, MPI_CHAR, MPI_ANY_SOURCE, tag,
                   MPI_COMM_WORLD, &status);
          printf("Recebendo msg de %d\n", status.MPI_SOURCE);
         }
}
```

Propagação de Informação - Implementação

```
else
   {
   /*recebendo msg de uma tarefa vizinha qualquer*/
   MPI_Recv(message, 100, MPI_CHAR, MPI_ANY_SOURCE, tag,
                     MPI_COMM_WORLD, &status);
   /*enviando para todos os vizinhos de myRank*/
   for (i = 0; i < numeroDeTarefas; i++)
      if (matrizVizinhanca[myRank][i] == 1)
         MPI_Send(message, strlen(message)+1, MPI_CHAR, i, tag,
                  MPI_COMM_WORLD);
   /*recebendo de todos os vizinhos de myRank menos 1*/
   for(i = 0; i < (numeroDeVizinhos - 1); i++)
      MPI_Recv(message, 100, MPI_CHAR, MPI_ANY_SOURCE, tag,
               MPI_COMM_WORLD, &status);
   }

   // Finalização do MPI
   MPI_Finalize();
}
```

Propagação de Informações com Realimentação

Propagação de Informações com Realimentação

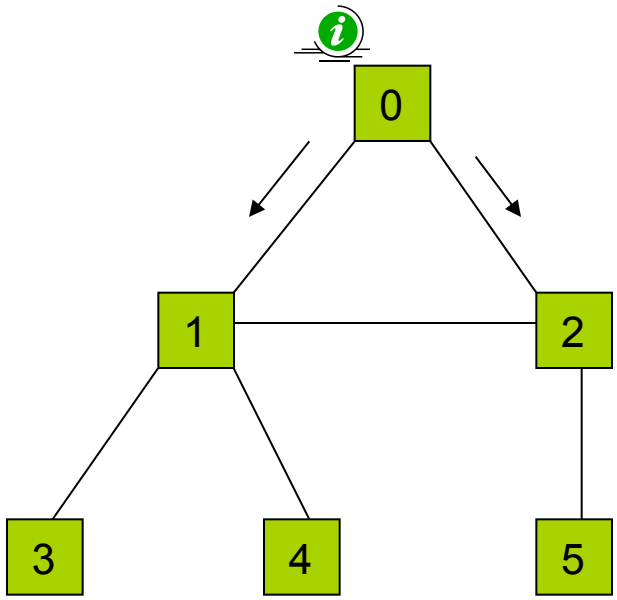


Nó 0 gera uma informação que tem de ser encaminhada a todos os demais e recebe uma confirmação quando a informação já estiver completamente disseminada.

Propagação de Informações com Realimentação - Algoritmo

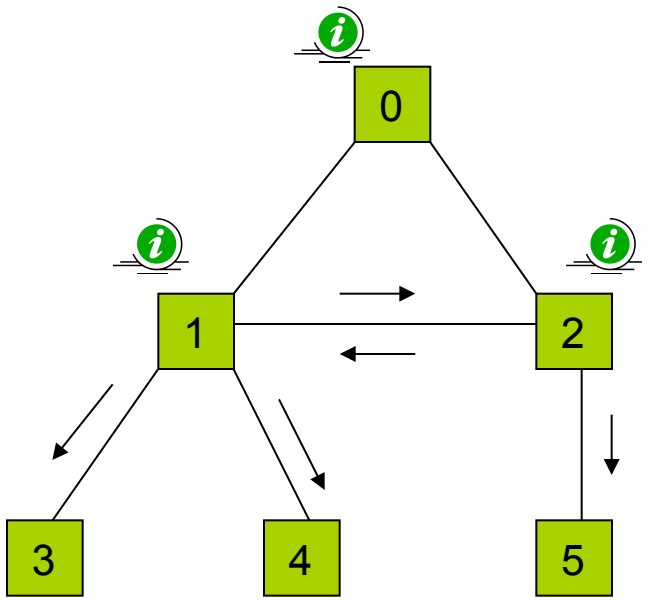
```
Variáveis
   pai := nil;
   count := 0;
   alcançado := falso;

Ação para fonte inicial da informação (inf):
   alcançado := verdadeiro;
   envie inf a todos os vizinhos;

Ao receber inf:
    count := count +1;
    se alcançado = falso
         alcançado := verdadeiro;
         pai := origem(inf);
         envie inf a todos os vizinhos exceto pai;
    se count = |vizinhos| e pai ≠ nil
         envie inf para pai;
```

Propagação de Informações com Realimentação - Execução



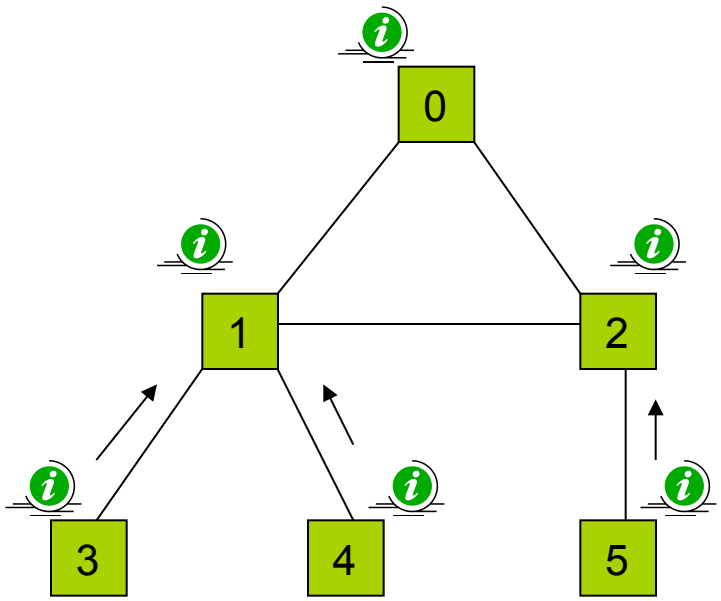
Nó 0 gera uma informação e a encaminha a todos os seus vizinhos

Propagação de Informações com Realimentação - Execução



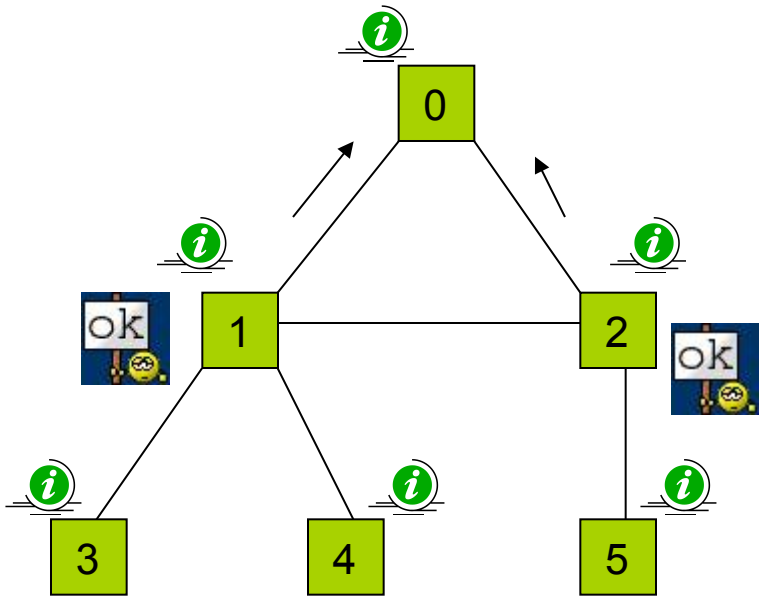
Nós 1 e 2 recebem a informação e a encaminham a todos os seus vizinhos, exceto pai

Propagação de Informações com Realimentação - Execução



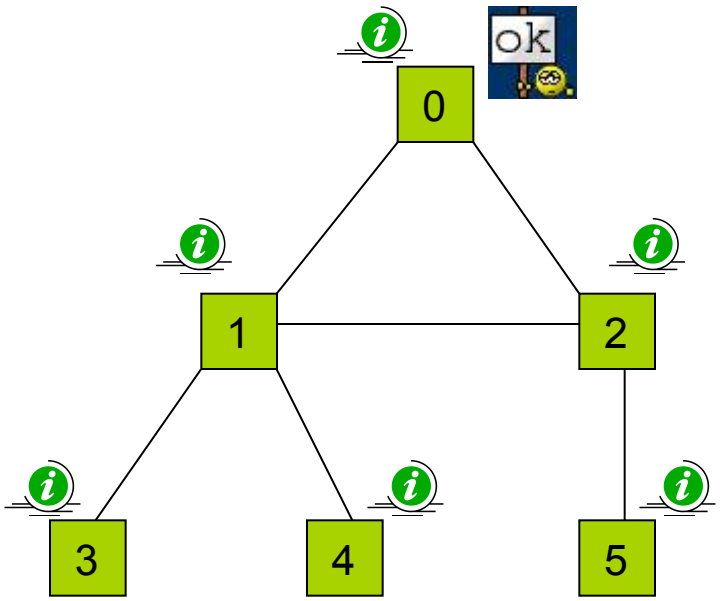
Nós 3, 4 e 5 recebem a informação e como são folhas, a retornam para seus pais.

Propagação de Informações com Realimentação - Execução



Nós 1 e 2 receberam o retorno de seus filhos e podem informar ao seus pais

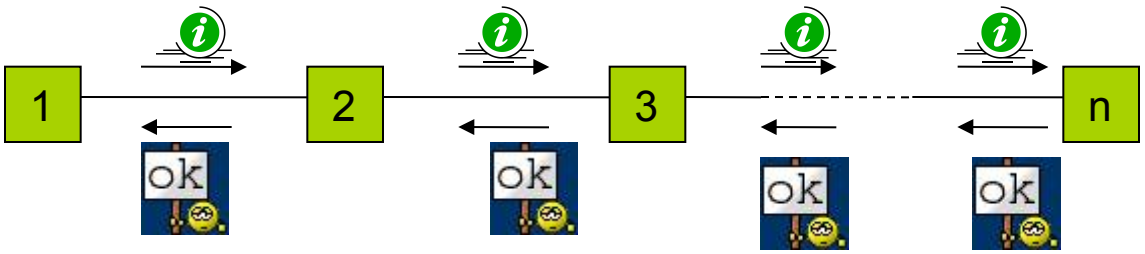
Propagação de Informações com Realimentação - Execução



Nó 0 recebeu o retorno de todos os seus filhos e pode concluir que todos os nós do sistema já receberam a informação.

Propagação de Informações com Realimentação - Complexidade

- Mensagens: 2e => **O(e)**
- Tempo: 2(n-1) => **O(n)**



Propagação de Informações com Realimentação - Implementação

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

/*Definir o grafo da aplicação antes de executar*/
int numeroDeTarefas = 6;
int matrizVizinhanca[6][6] = {    {0,1,1,0,0,0},
                                  {1,0,1,1,1,0},
                                  {1,1,0,0,0,1},
                                  {0,1,0,0,0,0},
                                  {0,1,0,0,0,0},
                                  {0,0,1,0,0,0}
                              };
```

Propagação de Informações com Realimentação - Implementação

```
/*retorna o número de vizinhos da tarefa myRank*/
int contaNumeroDeVizinhos(int myRank)
{
   int i;
   int contador = 0;

   for (i = 0; i < numeroDeTarefas; i++)
      if (matrizVizinhanca[myRank][i] == 1)
         contador++;

   return contador;
}
```

Propagação de Informações com Realimentação - Implementação

```
/*programa principal*/
int main(int argc, char** argv)
{
   int i;
   int numeroDeVizinhos;
   int myRank;
   int source;
   int tag = 50;
   int pai;
   char message[100] = "Oi!";
   MPI_Status status;

   //inicialização do MPI
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

   numeroDeVizinhos = contaNumeroDeVizinhos(myRank);
```

Propagação de Informações com Realimentação - Implementação

```
if (myRank == 0)
   {
      /*enviando para todos os vizinhos de myRank*/
      for (i = 0; i < numeroDeTarefas; i++)
         if (matrizVizinhanca[myRank][i] == 1)
         {
            printf("Enviando mensagem para %d\n", i);
            MPI_Send(message, strlen(message)+1, MPI_CHAR, i, tag,
                     MPI_COMM_WORLD);
         }

      /*recebendo de todos os vizinhos de myRank*/
      for(i = 0; i < numeroDeVizinhos; i++)
        {
           MPI_Recv(message, 100, MPI_CHAR, MPI_ANY_SOURCE, tag,
                    MPI_COMM_WORLD, &status);
          printf("Confirmação do filho %d\n", status.MPI_SOURCE);
        }
   }
```

Propagação de Informações com Realimentação - Implementação

```
else
   {
   /*recebendo msg de uma tarefa vizinha qualquer*/
   MPI_Recv(message, 100, MPI_CHAR, MPI_ANY_SOURCE, tag,
            MPI_COMM_WORLD, &status);
   pai = status.MPI_SOURCE;

   /*enviando para todos os vizinhos de myRank menos seu pai*/
   for (i = 0; i < numeroDeTarefas; i++)
      if ( (matrizVizinhanca[myRank][i] == 1) && (i != pai) )
         MPI_Send(message, strlen(message)+1, MPI_CHAR, i, tag,
                  MPI_COMM_WORLD);
   /*recebendo de todos os vizinhos de myRank menos 1*/
   for(i = 0; i < (numeroDeVizinhos - 1); i++)
      MPI_Recv(message, 100, MPI_CHAR, MPI_ANY_SOURCE, tag,
               MPI_COMM_WORLD, &status);
   MPI_Send(message, strlen(message)+1, MPI_CHAR, pai, tag,
            MPI_COMM_WORLD);
   }
```

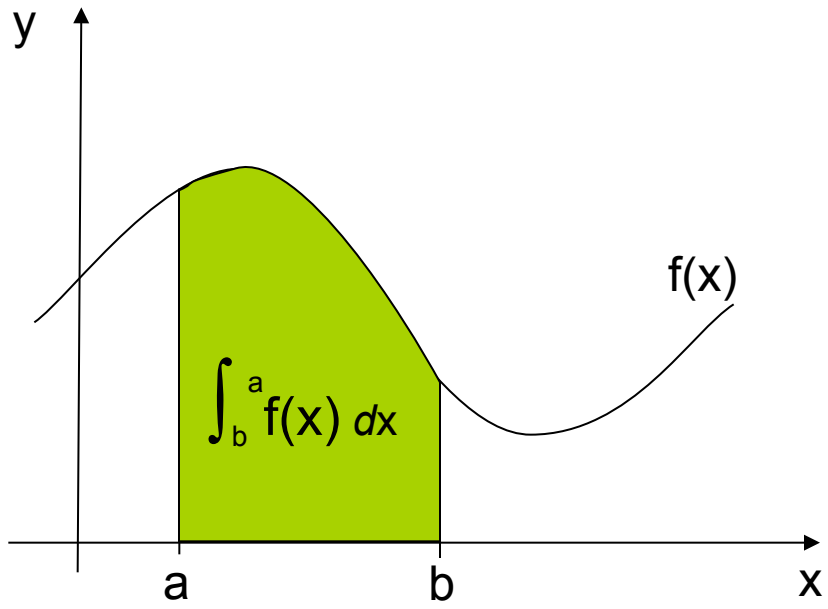
Propagação de Informações com Realimentação - Implementação

```
    //Finalização do MPI
   MPI_Finalize();
}
```

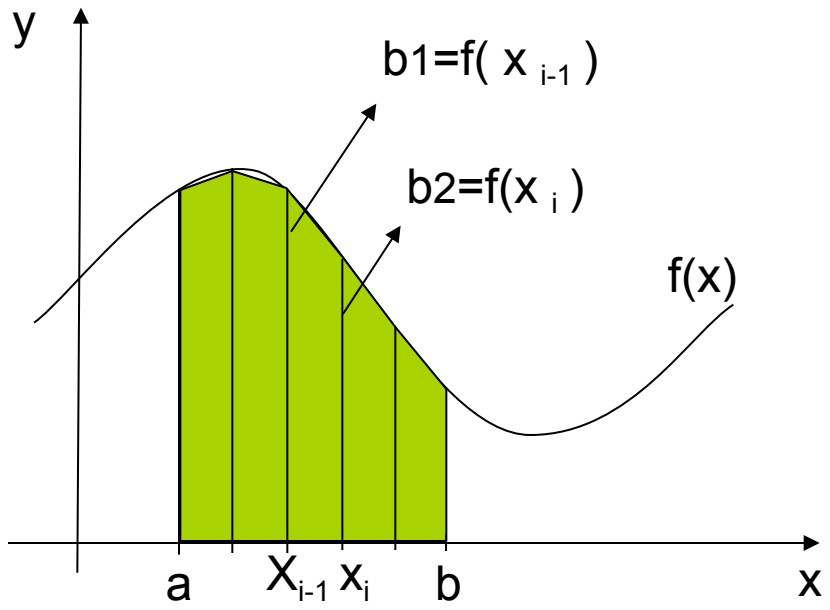
Integral Definida

Método do Trapezóide

Integral Definida



Integral Definida – Método do Trapezóide



Área do trapézio = **h (b1+b2) / 2**

h=(b-a)/n (constante)

Área do i-ésimo trapésio:
(h/2)[f(x_{i-1}) + f(x_i)]

$$\int_b^a f(x)\,dx =$$

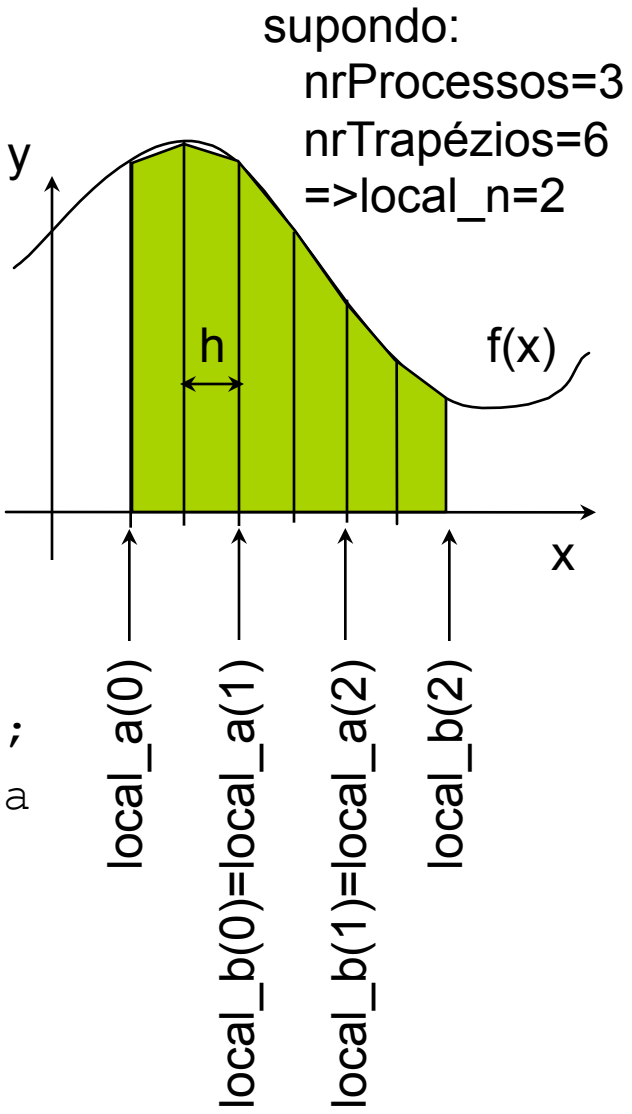
$(h/2)[f(x_0)+f(x_1)] + (h/2)[f(x_1)+f(x_2)] + (h/2)[f(x_2)+f(x_3)] + \ldots + (h/2)[f(x_{n0})+f(x_n)] =$

$(h/2)[\, f(x_0) + 2\, f(x_1) + 2\, f(x_2) + 2\, f(x_3) + \ldots + 2\, f(x_{n-1}) + f(x_n)\,] =$

$h\,[\, f(x_0)/2 + f(x_n)/2 + f(x_1) + f(x_2) + f(x_3) + \ldots + f(x_{n-1})\,]$

Integral Definida - Algoritmo

```
Variáveis:
   h := (b-a)/nrTrapézios;
   local_n := nrTrapézios / nrProcessos;
   local_a := a + my_rank * local_n * h;
   local_b := local_a + local_n * h;

Ação inicial para todos os nós:
   x := local_a;
   integral :=(f(local_a)+f(local_b))/2.0;
   para i variando de 1 até local_n-1 faça
      x := x + h;
      integral := integral + f(x);
   integral := integral * h;
```



Integral Definida - Algoritmo

```
se my_rank = 0
   total := integral;
   para i variando de 1 até nrProcessos – 1
      receba valor na variável integral;
      total = total + integral;
   imprima "Valor calculado: " + total;
senão
   envie integral para nó 0;
```

Integral Definida - Implementação

```
#include <stdio.h>
#include <mpi.h>
main(int argc, char** argv) {
   int my_rank;
   int p;                      //  número de processos
   float a=0.0, b=1.0; //  intervalo a calcular
   int n=1024;          //  número de trapezóides
   float h;                   //  base do trapezóide
   float local_a, local_b;    //  intervalo local
   int local_n;               //  número de trapezóides local
   float integral;            //  integral no meu intervalo
   float total;               //  integral total
   int source;          //  remetente da integral
   int dest=0;          //  destino das integrais (nó 0)
   int tag=200;         //  tipo de mensagem (único)
   MPI_Status status;
```

Integral Definida - Implementação

```
float calcula(float local_a, float local_b,
              int local_n, float h);

 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
 MPI_Comm_size(MPI_COMM_WORLD, &p);

 h = (b-a) / n;
 local_n = n / p;
 local_a = a + my_rank * local_n * h;
 local_b = local_a + local_n * h;

 integral = calcula(local_a, local_b, local_n, h);
```

Integral Definida - Implementação

```
   if(my_rank == 0) {
       total = integral;
       for(source=1; source<p; source++) {
            MPI_Recv(&integral, 1, MPI_FLOAT, source, tag,
                                MPI_COMM_WORLD, &status);
            total +=integral;
        }
    } else
            MPI_Send(&integral, 1, MPI_FLOAT, dest,
                     tag, MPI_COMM_WORLD);

    if(my_rank == 0) printf("Resultado: %f\n", total);
    MPI_Finalize();
}
```

Integral Definida - Implementação

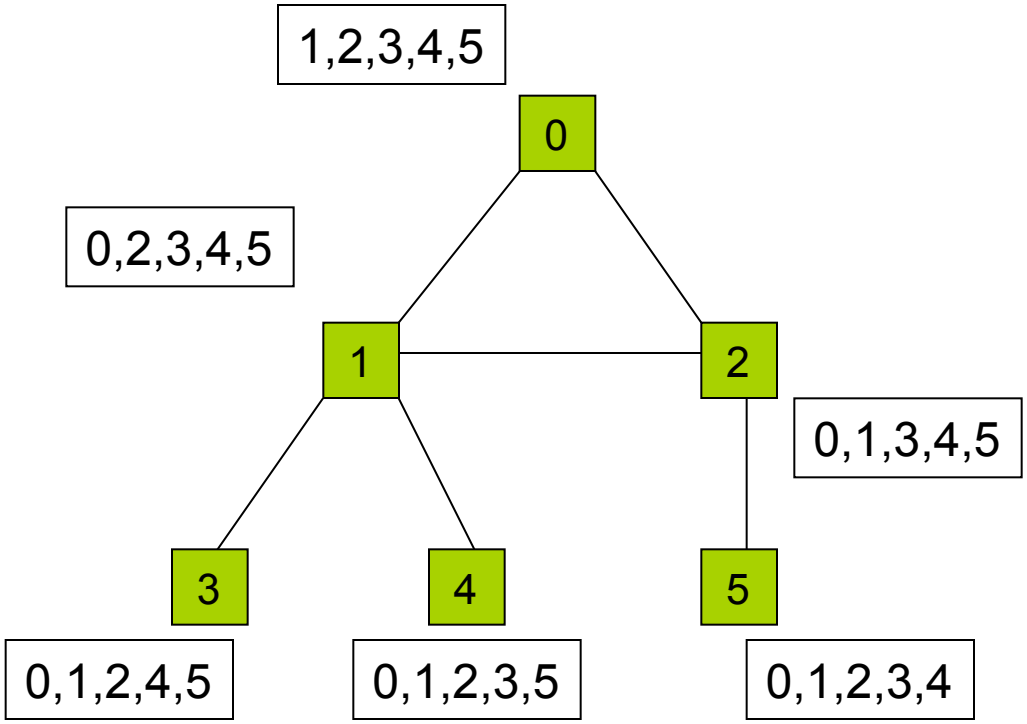
```
float calcula(float local_a, float local_b,
              int local_n, float h) {
    float integral;
    float x, i;

    float f(float x);  // função a integrar

    integral = ( f(local_a) + f(local_b) ) /2.0;

    x = local_a;
    for( i=1; i<=local_n0; i++) {
         x += h;
         integral += f(x);
    }
    integral *= h;
    return integral;
 }
```

Integral Definida – Implementação

```
float f(float x) {

   float fx; //  valor de retorno

   //  esta é a função a integrar
   //  exemplo: função quadrática
   fx = x * x;

   return fx;
}
```

Conectividade em Grafos

Conectividade em Grafos



O problema trata da descoberta, por cada nó, das identificações de todos os outros nós aos quais está conectado.

Conectividade em Grafos - Algoritmo

```
Variáveis:
   initiate = false;
   para todos os nós k de N;
        parent(k) := nil;
        count(k) := 0;
        reached(k) := false;

Ação se n ∈ N(0):
   initiate := true;
   reached(id) := true;
   envie id para todos os vizinhos;
```

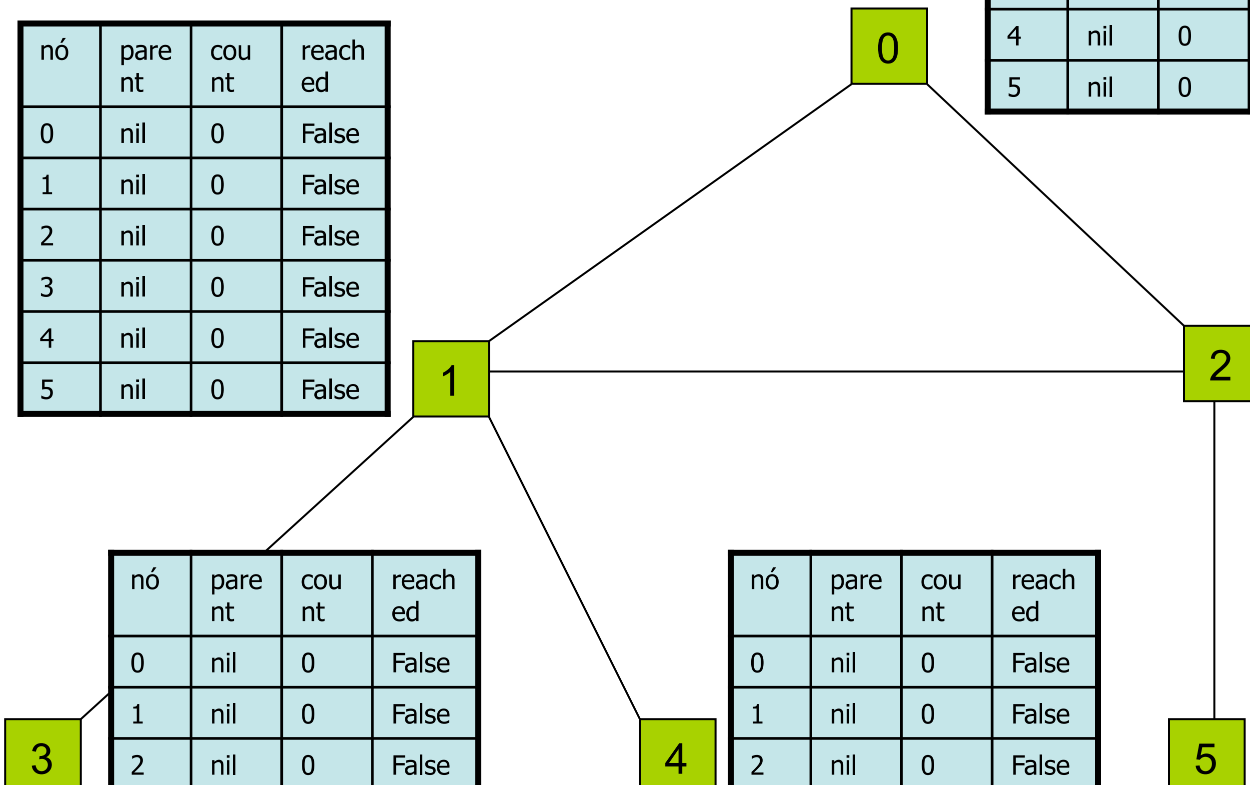
Conectividade em Grafos - Algoritmo

```
Input: id(k) recebido do nó j
   se initiated = false
        initiated := true;
        reached(id) := true;
        envie id para todos os vizinhos;
```

Conectividade em Grafos - Algoritmo

```
count(k) := count(k) + 1;
se reached(k) = false
     reached(k) := true;
     parent(k) := j;
     para todos os vizinhos exceto parent(k)
          envie id(k);

se count(k) = nr de vizinhos de i
     se parent(k) ≠ nil
          envie id(k) para parent(k)
```

Conectividade em Grafos - Execução



Node 0:

nó	parent	count	reached
0	nil	0	False
1	nil	0	False
2	nil	0	False
3	nil	0	False
4	nil	0	False
5	nil	0	False

Node 1:

nó	parent	count	reached
0	nil	0	False
1	nil	0	False
2	nil	0	False
3	nil	0	False
4	nil	0	False
5	nil	0	False

Node 2:

nó	parent	count	reached
0	nil	0	False
1	nil	0	False
2	nil	0	False
3	nil	0	False
4	nil	0	False
5	nil	0	False

Node 3:

nó	parent	count	reached
0	nil	0	False
1	nil	0	False
2	nil	0	False
3	nil	0	False
4	nil	0	False
5	nil	0	False

Node 4:

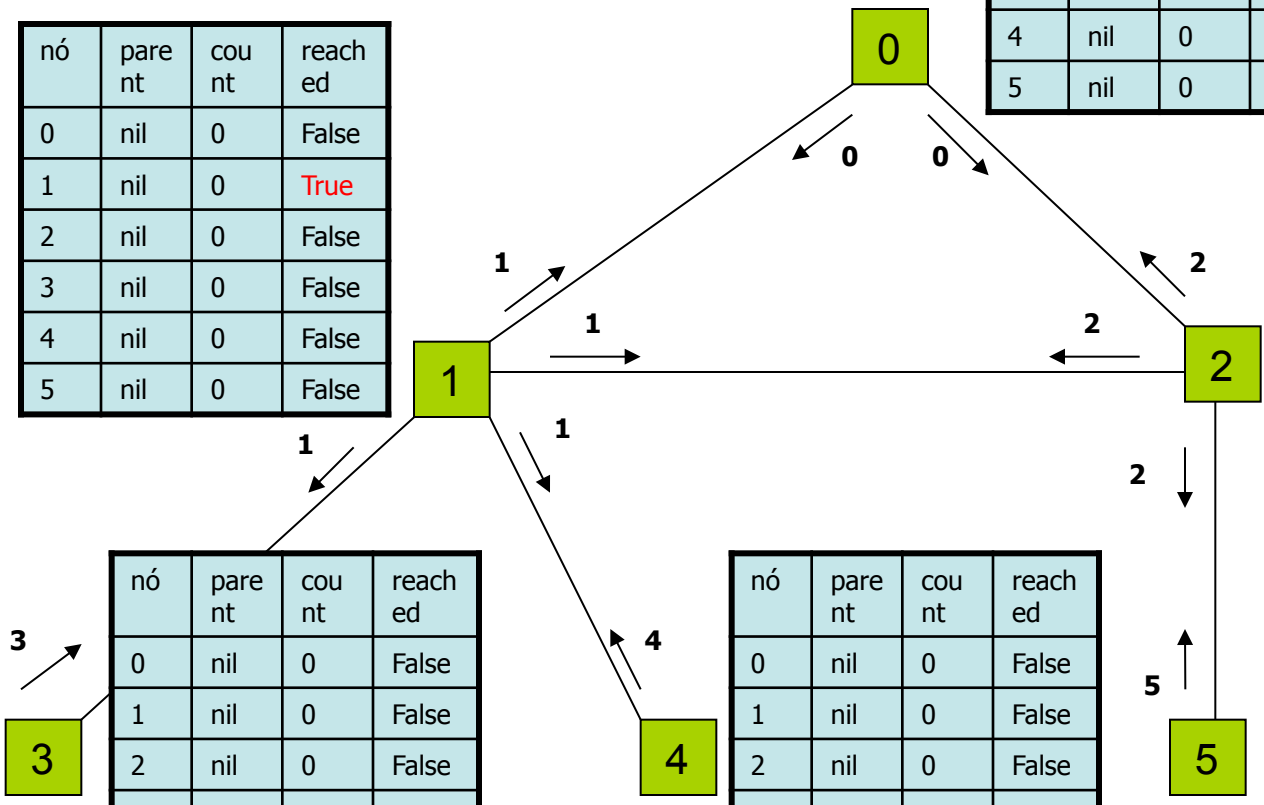
nó	parent	count	reached
0	nil	0	False
1	nil	0	False
2	nil	0	False
3	nil	0	False
4	nil	0	False
5	nil	0	False

Node 5:

nó	parent	count	reached
0	nil	0	False
1	nil	0	False
2	nil	0	False
3	nil	0	False
4	nil	0	False
5	nil	0	False

Conectividade em Grafos - Execução

nó	parent	count	reached
0	nil	0	True
1	nil	0	False
2	nil	0	False
3	nil	0	False
4	nil	0	False
5	nil	0	False



nó	parent	count	reached
0	nil	0	False
1	nil	0	True
2	nil	0	False
3	nil	0	False
4	nil	0	False
5	nil	0	False

nó	parent	count	reached
0	nil	0	False
1	nil	0	False
2	nil	0	True
3	nil	0	False
4	nil	0	False
5	nil	0	False

nó	parent	count	reached
0	nil	0	False
1	nil	0	False
2	nil	0	False
3	nil	0	True
4	nil	0	False
5	nil	0	False

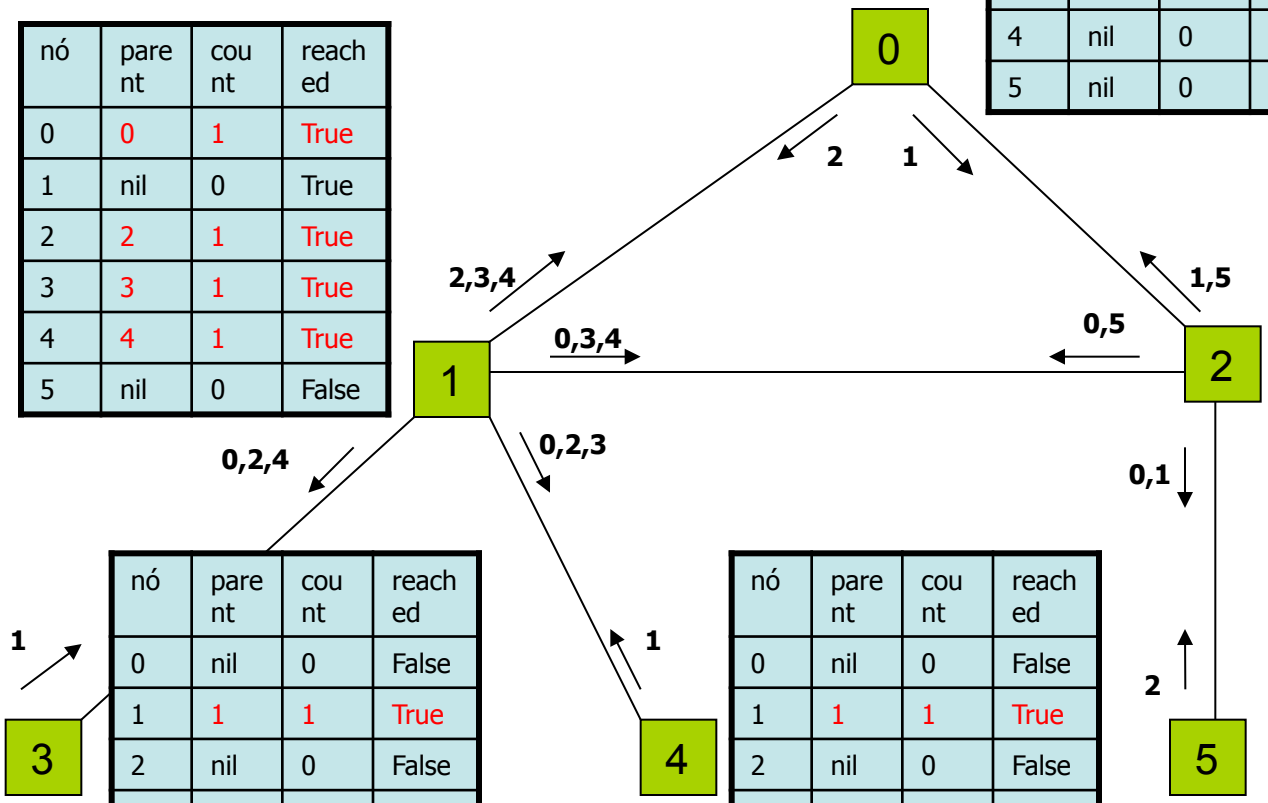
nó	parent	count	reached
0	nil	0	False
1	nil	0	False
2	nil	0	False
3	nil	0	False
4	nil	0	True
5	nil	0	False

nó	parent	count	reached
0	nil	0	False
1	nil	0	False
2	nil	0	False
3	nil	0	False
4	nil	0	False
5	nil	0	True

Conectividade em Grafos - Execução

nó	parent	count	reached
0	nil	0	True
1	1	1	True
2	2	1	True
3	nil	0	False
4	nil	0	False
5	nil	0	False

nó	parent	count	reached
0	0	1	True
1	nil	0	True
2	2	1	True
3	3	1	True
4	4	1	True
5	nil	0	False



nó	parent	count	reached
0	0	1	True
1	1	1	True
2	nil	0	True
3	nil	0	False
4	nil	0	False
5	5	1	True

nó	parent	count	reached
0	nil	0	False
1	1	1	True
2	nil	0	False
3	nil	0	True
4	nil	0	False
5	nil	0	False

nó	parent	count	reached
0	nil	0	False
1	1	1	True
2	nil	0	False
3	nil	0	False
4	nil	0	True
5	nil	0	False

nó	parent	count	reached
0	nil	0	False
1	nil	0	False
2	2	1	True
3	nil	0	False
4	nil	0	False
5	nil	0	True

Conectividade em Grafos - Execução

nó	parent	count	reached
0	nil	0	True
1	1	2	True
2	2	2	True
3	1	1	True
4	1	1	True
5	2	1	True

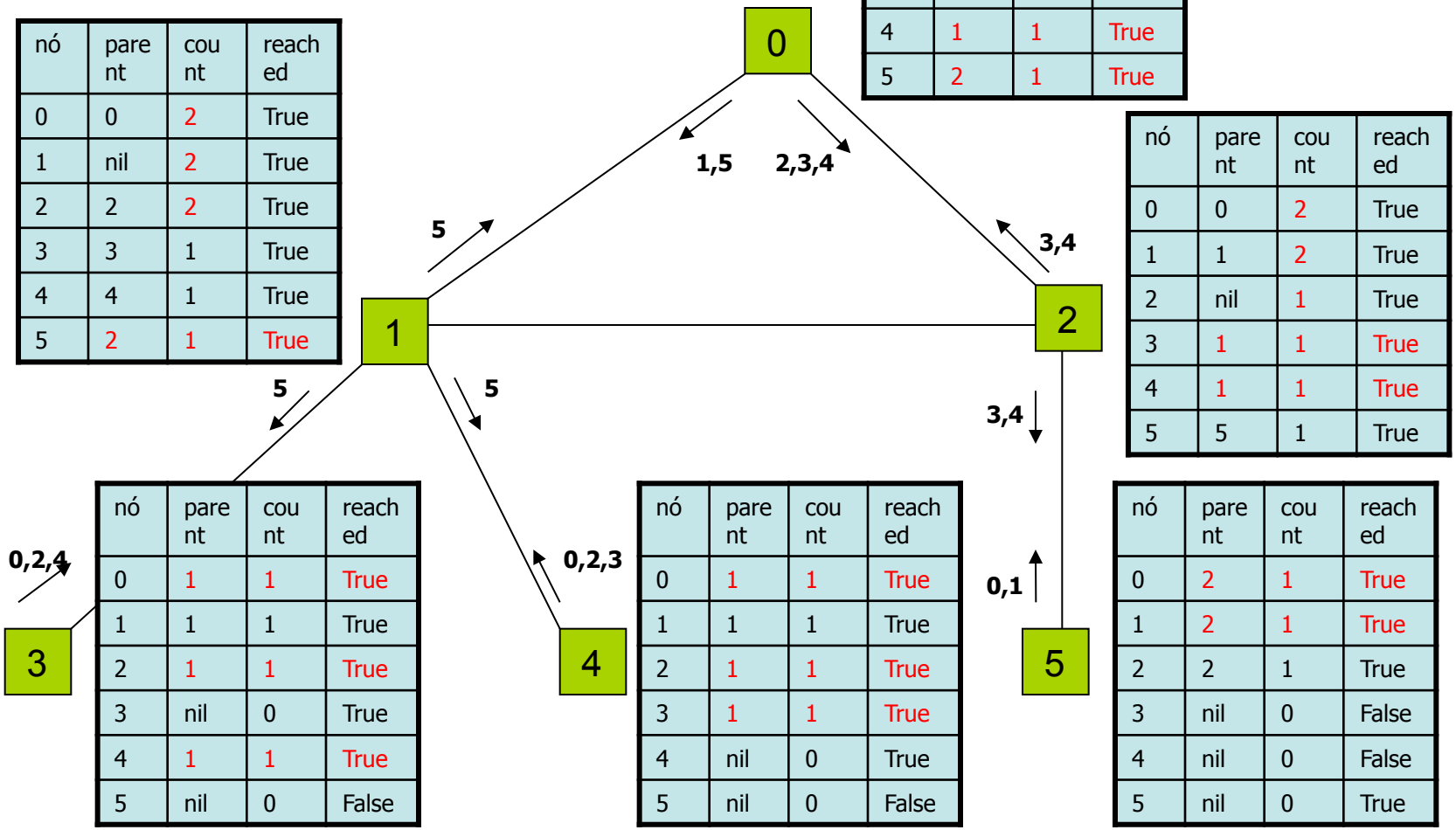
nó	parent	count	reached
0	0	2	True
1	nil	2	True
2	2	2	True
3	3	1	True
4	4	1	True
5	2	1	True

nó	parent	count	reached
0	0	2	True
1	1	2	True
2	nil	1	True
3	1	1	True
4	1	1	True
5	5	1	True

nó	parent	count	reached
0	1	1	True
1	1	1	True
2	1	1	True
3	nil	0	True
4	1	1	True
5	nil	0	False

nó	parent	count	reached
0	1	1	True
1	1	1	True
2	1	1	True
3	1	1	True
4	nil	0	True
5	nil	0	False

nó	parent	count	reached
0	2	1	True
1	2	1	True
2	2	1	True
3	nil	0	False
4	nil	0	False
5	nil	0	True

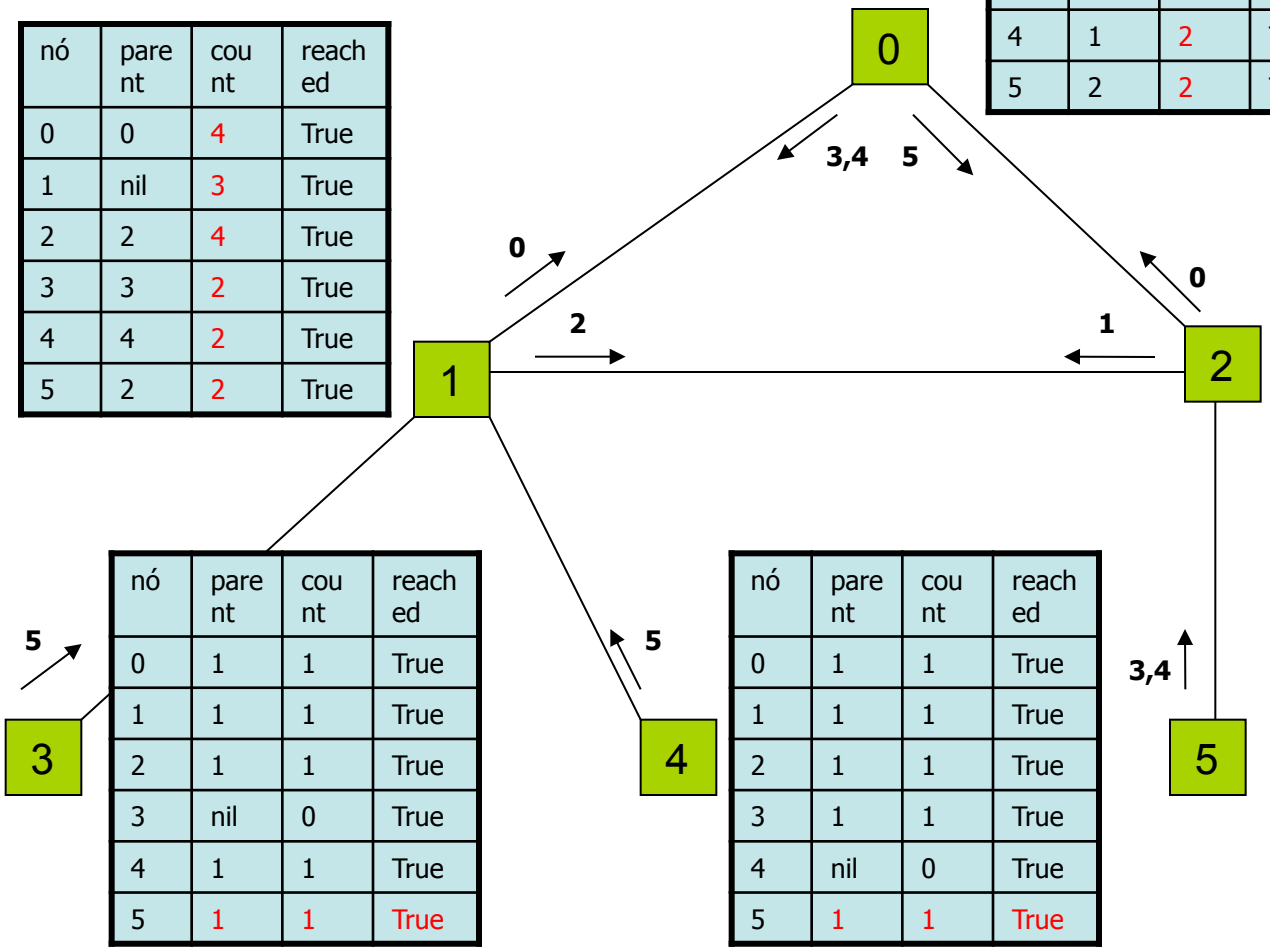


Conectividade em Grafos - Execução

nó	parent	count	reached
0	nil	0	True
1	1	2	True
2	2	2	True
3	1	2	True
4	1	2	True
5	2	2	True

nó	parent	count	reached
0	0	4	True
1	nil	3	True
2	2	4	True
3	3	2	True
4	4	2	True
5	2	2	True

nó	parent	count	reached
0	0	3	True
1	1	3	True
2	nil	2	True
3	1	2	True
4	1	2	True
5	5	1	True



nó	parent	count	reached
0	1	1	True
1	1	1	True
2	1	1	True
3	nil	0	True
4	1	1	True
5	1	1	True

nó	parent	count	reached
0	1	1	True
1	1	1	True
2	1	1	True
3	1	1	True
4	nil	0	True
5	1	1	True

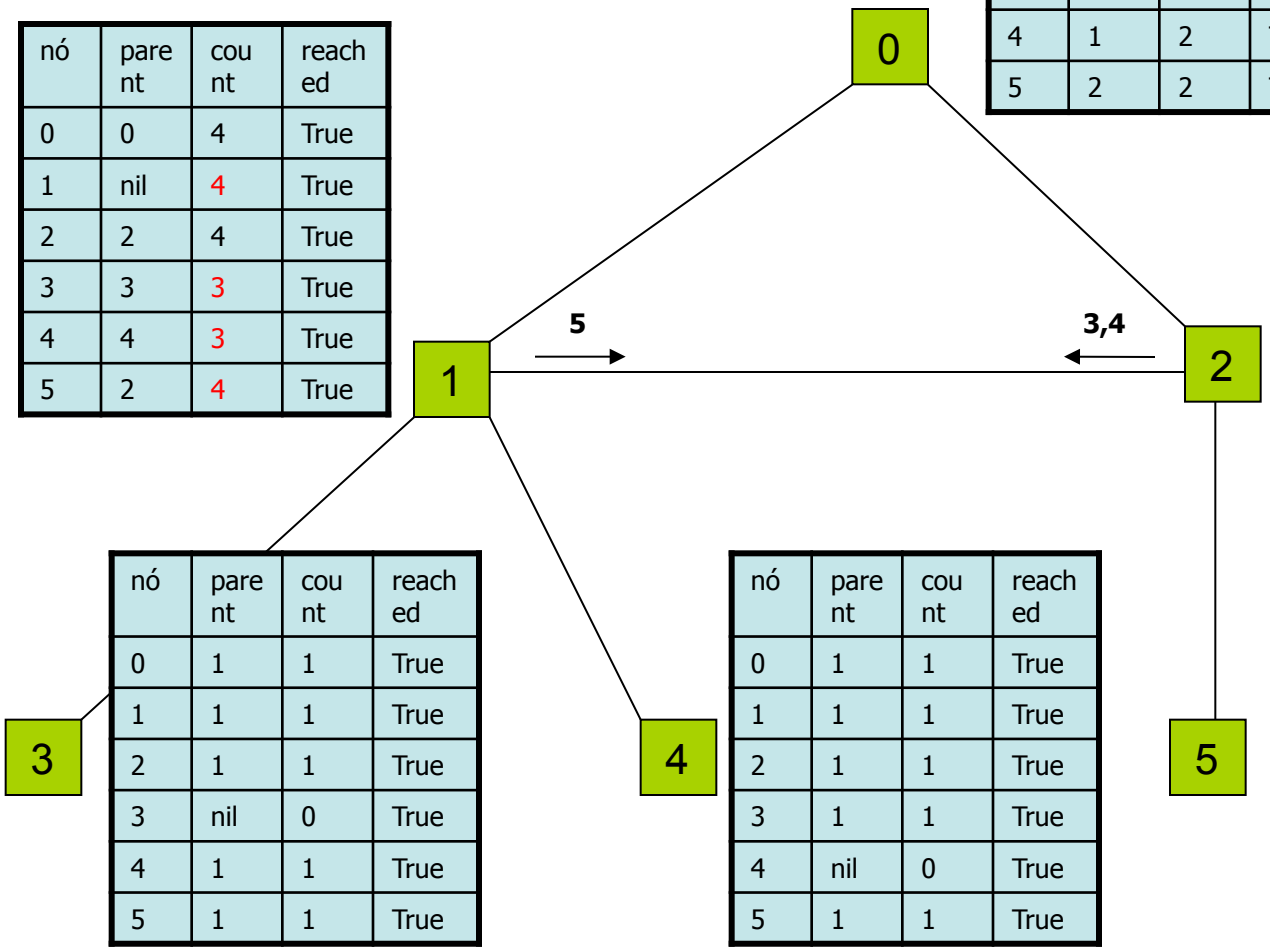
nó	parent	count	reached
0	2	1	True
1	2	1	True
2	2	1	True
3	2	1	True
4	2	1	True
5	nil	0	True

Conectividade em Grafos - Execução

nó	parent	count	reached
0	0	4	True
1	nil	4	True
2	2	4	True
3	3	3	True
4	4	3	True
5	2	4	True

nó	parent	count	reached
0	nil	2	True
1	1	2	True
2	2	2	True
3	1	2	True
4	1	2	True
5	2	2	True

nó	parent	count	reached
0	0	3	True
1	1	3	True
2	nil	3	True
3	1	3	True
4	1	3	True
5	5	2	True



nó	parent	count	reached
0	1	1	True
1	1	1	True
2	1	1	True
3	nil	0	True
4	1	1	True
5	1	1	True

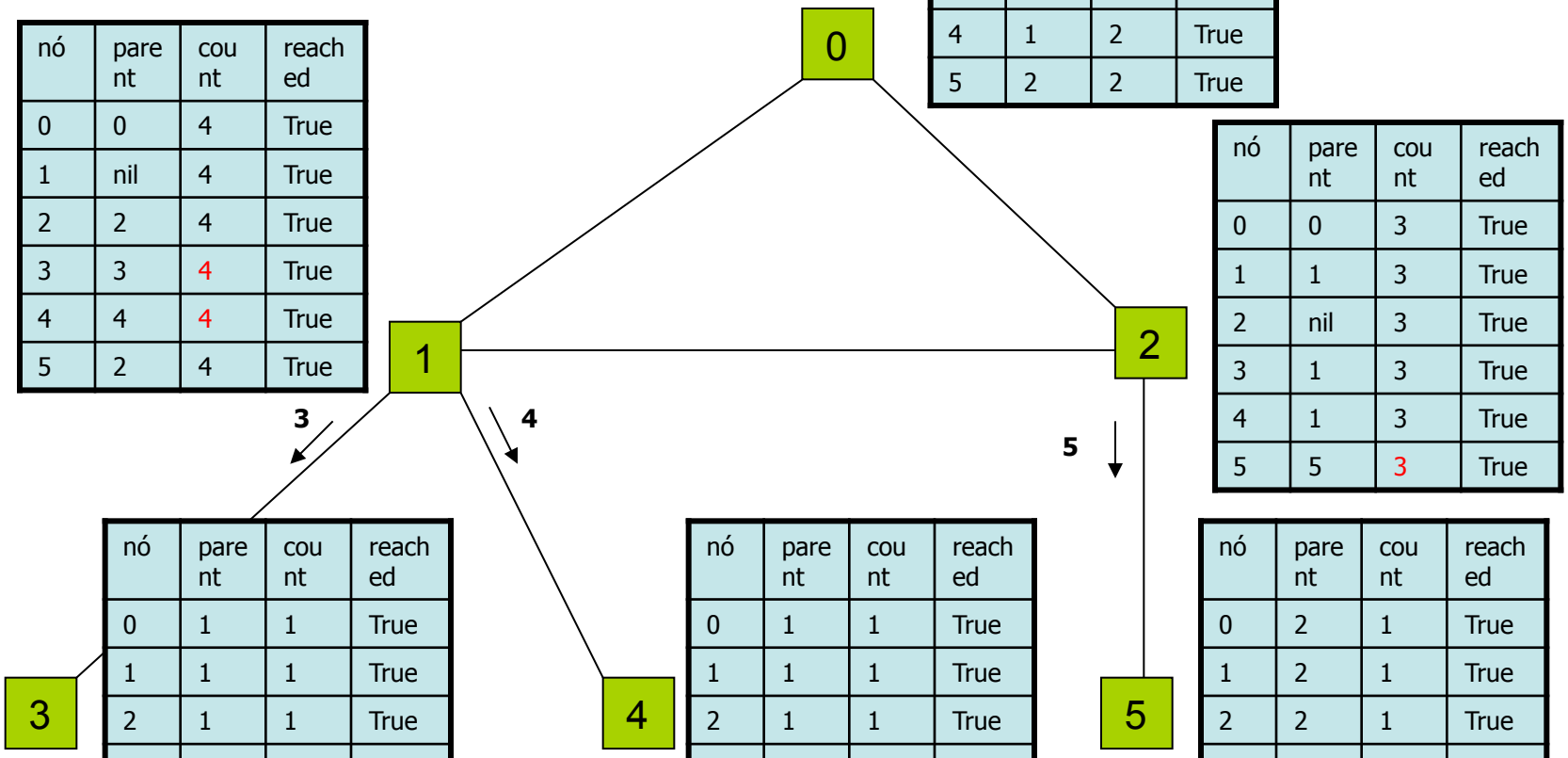
nó	parent	count	reached
0	1	1	True
1	1	1	True
2	1	1	True
3	1	1	True
4	nil	0	True
5	1	1	True

nó	parent	count	reached
0	2	1	True
1	2	1	True
2	2	1	True
3	2	1	True
4	2	1	True
5	nil	0	True

Conectividade em Grafos - Execução

nó	parent	count	reached
0	nil	2	True
1	1	2	True
2	2	2	True
3	1	2	True
4	1	2	True
5	2	2	True

nó	parent	count	reached
0	0	4	True
1	nil	4	True
2	2	4	True
3	3	4	True
4	4	4	True
5	2	4	True



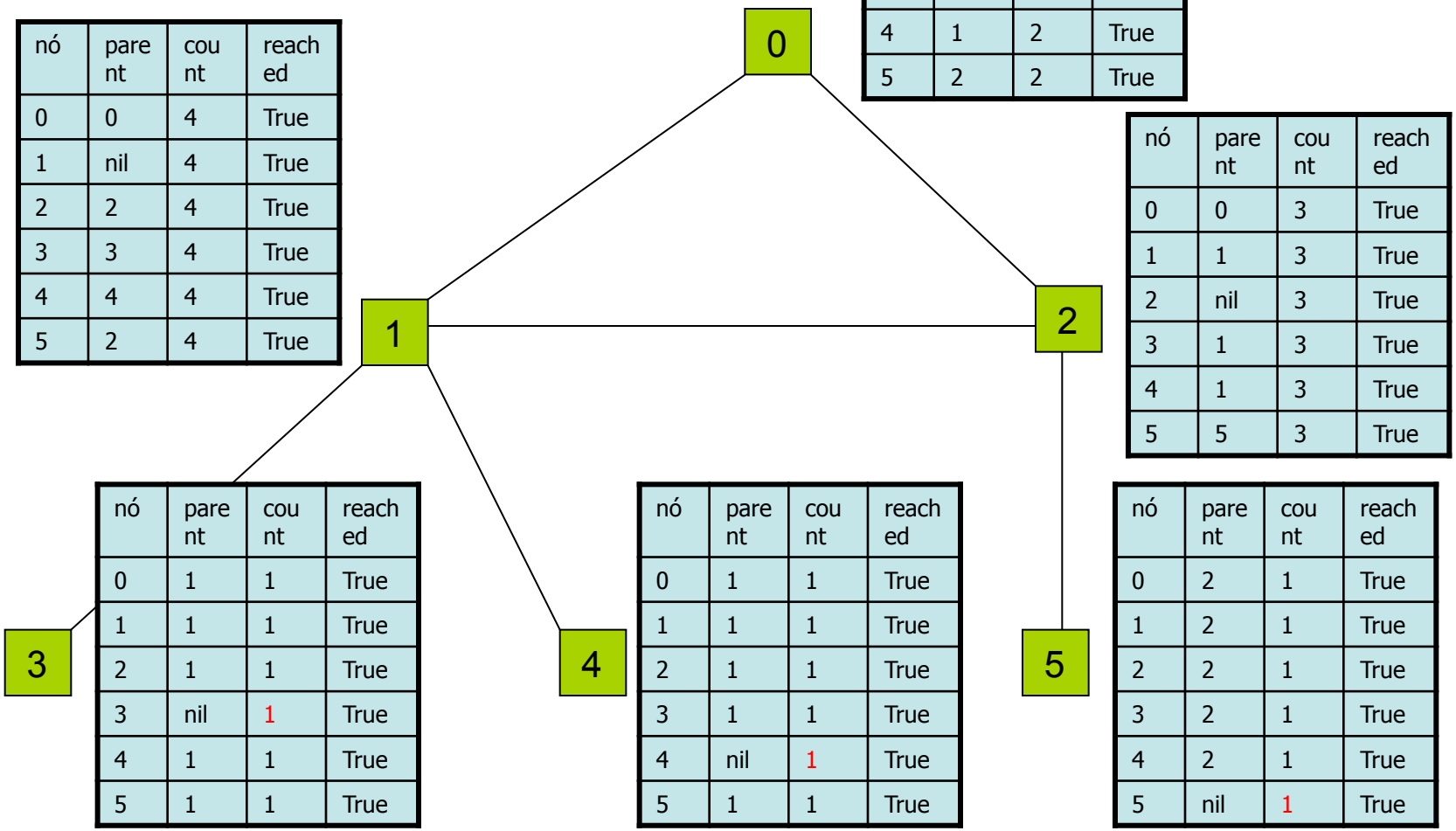
nó	parent	count	reached
0	0	3	True
1	1	3	True
2	nil	3	True
3	1	3	True
4	1	3	True
5	5	3	True

nó	parent	count	reached
0	1	1	True
1	1	1	True
2	1	1	True
3	nil	0	True
4	1	1	True
5	1	1	True

nó	parent	count	reached
0	1	1	True
1	1	1	True
2	1	1	True
3	1	1	True
4	nil	0	True
5	1	1	True

nó	parent	count	reached
0	2	1	True
1	2	1	True
2	2	1	True
3	2	1	True
4	2	1	True
5	nil	0	True

Conectividade em Grafos - Execução



nó	parent	count	reached
0	0	4	True
1	nil	4	True
2	2	4	True
3	3	4	True
4	4	4	True
5	2	4	True

nó	parent	count	reached
0	nil	2	True
1	1	2	True
2	2	2	True
3	1	2	True
4	1	2	True
5	2	2	True

nó	parent	count	reached
0	0	3	True
1	1	3	True
2	nil	3	True
3	1	3	True
4	1	3	True
5	5	3	True

nó	parent	count	reached
0	1	1	True
1	1	1	True
2	1	1	True
3	nil	1	True
4	1	1	True
5	1	1	True

nó	parent	count	reached
0	1	1	True
1	1	1	True
2	1	1	True
3	1	1	True
4	nil	1	True
5	1	1	True

nó	parent	count	reached
0	2	1	True
1	2	1	True
2	2	1	True
3	2	1	True
4	2	1	True
5	nil	1	True

Conectividade em Grafos – Implementação

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

int numeroDeTarefas = 8;
int matrizVizinhanca[8][8] = {    {0,1,1,0,0,0,0,0},
                                  {1,0,1,1,1,0,0,0},
                                  {1,1,0,0,0,0,0,0},
                                  {0,1,0,0,0,0,0,0},
                                  {0,1,0,0,0,0,0,0},
                                  {0,0,0,0,0,0,1,1},
                                  {0,0,0,0,0,1,0,1},
                                  {0,0,0,0,0,1,1,0},
                                };
```

Conectividade em Grafos – Implementação

```
/*retorna o número de vizinhos da tarefa myRank*/
int contaNumeroDeVizinhos(int myRank)
{
   int i;
   int contador = 0;

   for (i = 0; i < numeroDeTarefas; i++)
      if (matrizVizinhanca[myRank][i] == 1)
         contador++;

   return contador;
}
```

Conectividade em Grafos – Implementação

```
int finaliza(int contador[], int numeroDeVizinhos,
             int myRank) {
   int i, todosiguaisa0=1;

   if(contador[myRank]!=numeroDeVizinhos) return 0;

   elsereturn 1;
}
```

Conectividade em Grafos – Implementação

```
/*programa principal*/
int main(int argc, char** argv)
{
   int i, j;
   int numeroDeVizinhos;
   int myRank;
   int source;
   int tag = 50;
   int pai[numeroDeTarefas];
   int contador[numeroDeTarefas];
   int reached[numeroDeTarefas];
   int id;
   int origem;
   MPI_Status status;
```

Conectividade em Grafos – Implementação

```
//inicialização do MPI
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

numeroDeVizinhos = contaNumeroDeVizinhos(myRank);

//inicializando variáveis
for (i=0; i<numeroDeTarefas; i++)
{
    pai[i]=0;
    contador[i]=0;
    reached[i]=0;
}
reached[myRank]=1;
```

Conectividade em Grafos – Implementação

```
 //enviando minha id para os vizinhos
for (i=0; i<numeroDeTarefas; i++)
{
    if (matrizVizinhanca[myRank][i]==1)
       MPI_Send(&myRank, 1, MPI_INT, i, tag,
                MPI_COMM_WORLD);
}
```

Conectividade em Grafos – Implementação

```
while (!finaliza(contador, numeroDeVizinhos, myRank))
{
    MPI_Recv(&id, 1, MPI_INT, MPI_ANY_SOURCE, tag,
              MPI_COMM_WORLD, &status);
    origem=status.MPI_SOURCE;
    contador[id]++;
    if (reached[id]==0) {
       reached[id]=1;
       pai[id]=origem;
       for (j=0; j<numeroDeTarefas; j++)
          if (matrizVizinhanca[myRank][j]==1 && j!=pai[id])
             MPI_Send(&id, 1, MPI_INT, j, tag,
                      MPI_COMM_WORLD);
    }
```

Conectividade em Grafos – Implementação

```
        if (contador[id]==numeroDeVizinhos)
                if (pai[id]!=0)
                   MPI_Send(&id, 1, MPI_INT, pai[id], tag,
                            MPI_COMM_WORLD);
    }
    //imprimindo resultado
    printf("processo %d: Elementos conectados: ", myRank);
    for (i=0; i<numeroDeTarefas; i++)
        if (contador[i]!=0) printf("%d ", i);
    printf("\n");
    fflush(stdout);
     //Finalização do MPI
    MPI_Finalize();
}
```

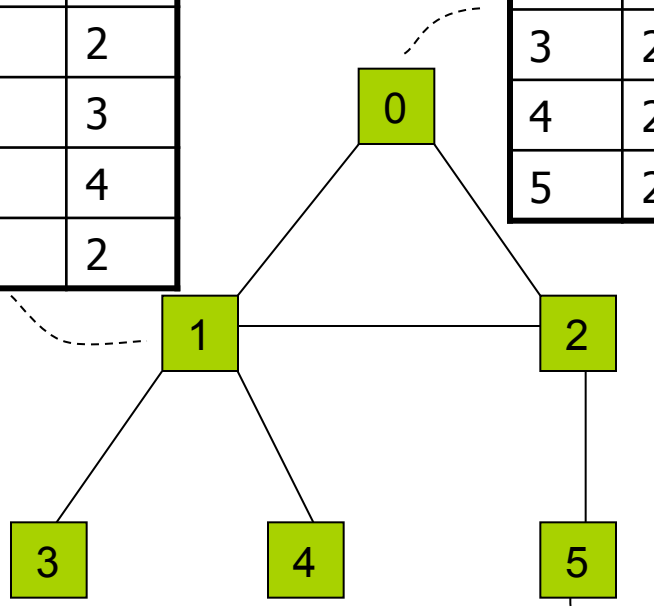
Distância Mínima

Algoritmo Síncrono
Algoritmo Assíncrono

Distância Mínima

nó	dist.	first
0	1	0
2	1	2
3	1	3
4	1	4
5	2	2

nó	dist.	first
1	1	1
2	1	2
3	2	1
4	2	1
5	2	2



nó	dist.	first
0	2	2
1	2	2
2	1	2
3	3	2
4	3	2

Ao final do algoritmo, cada nó deve conhecer a distância mais curta para cada um dos demais, e através de qual vizinho se atinge outro nó com esta distância.

Por exemplo, estas são as informações dos nós 0, 1 e 5.

Distância Mínima – Algoritmo Síncrono

```
Variáveis:
   dist(i) := 0;
   dist(k) := n para todo nó k ≠ i;
   first(k) = nil para todo nó k ≠ i;
   set = {id};

Inicio: s=0
   envie set para todos os vizinhos
```

Distância Mínima – Algoritmo Síncrono

```
Entrada:
   0<s≤n-1,
   msg(s) com set`s recebidos de nós n(j);

   set_out := {};
   para cada set recebido em msg
        para cada id(k) em set
             se dist(k)>s
                  dist(k) := s;
                  first(k) := n(j);
                  set_out := set_out U {id(k)}
   envie set_out para todos os vizinhos
```

Distância Mínima – Algoritmo Assíncrono

```
Variáveis:
   dist(i) := 0;
   dist(k) := n para todo nó k ≠ i;
   first(k) := nil para todo nó k ≠ i;
   set := {id};
   level(j) := 0 para todos os vizinhos de i;
   state := 0;

Ação Inicial:
   envie set para todos os vizinhos;
```

Distância Mínima – Algoritmo Assíncrono

```
Input: set do nó j;
   se state < n
         level(j) := level(j) + 1;
         para cada id(k) em set
               se dist(k) > level(j)
                     dist(k) := level(j);
                     first(k) := j;
```

Distância Mínima – Algoritmo Assíncrono

```
se state < level(j) para todo j vizinho
     state := state + 1;
     set := {id(k) | dist(k) = state};
     envie set para todos os vizinhos;
```

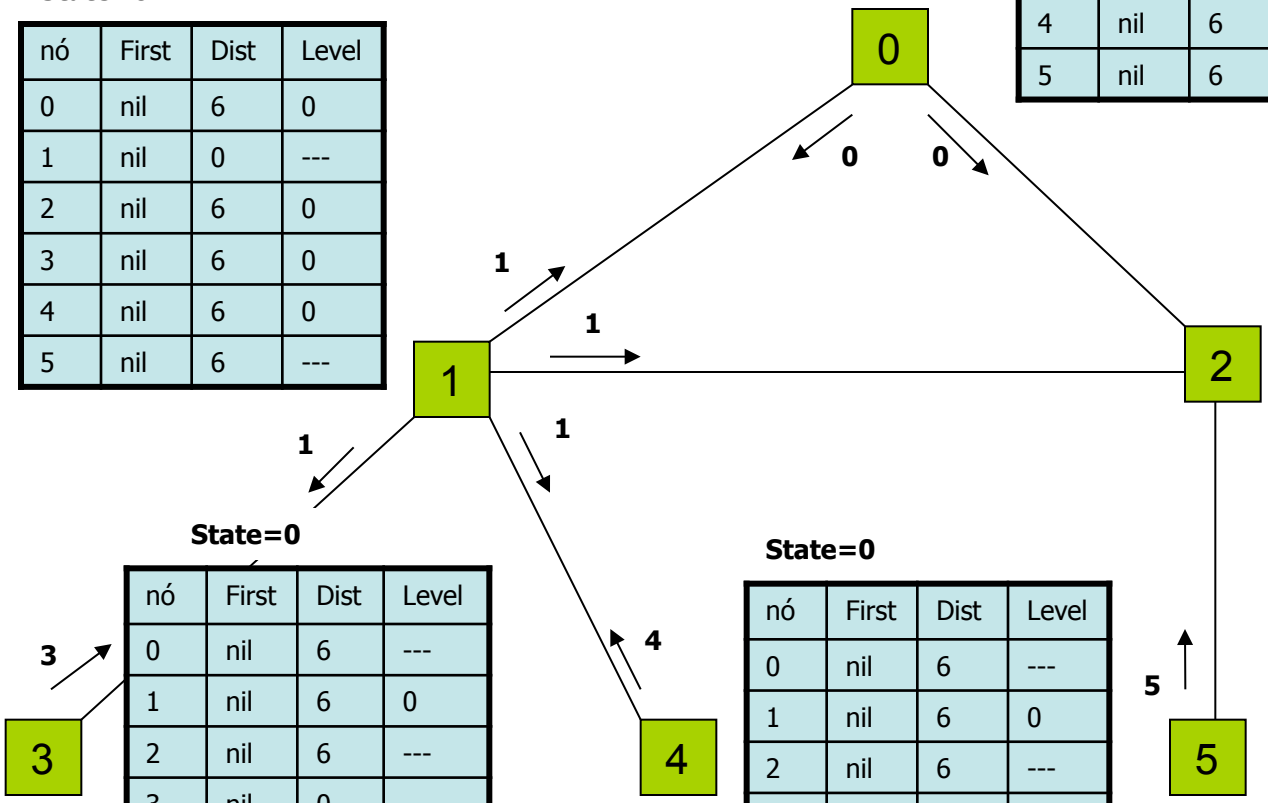
Distância Mínima – Execução

State=0

nó	First	Dist	Level
0	nil	0	---
1	nil	6	0
2	nil	6	0
3	nil	6	---
4	nil	6	---
5	nil	6	---

State=0

nó	First	Dist	Level
0	nil	6	0
1	nil	0	---
2	nil	6	0
3	nil	6	0
4	nil	6	0
5	nil	6	---



State=0

nó	First	Dist	Level
0	nil	6	0
1	nil	6	0
2	nil	0	---
3	nil	6	---
4	nil	6	---
5	nil	6	0

State=0

nó	First	Dist	Level
0	nil	6	---
1	nil	6	0
2	nil	6	---
3	nil	0	---
4	nil	6	---
5	nil	6	---

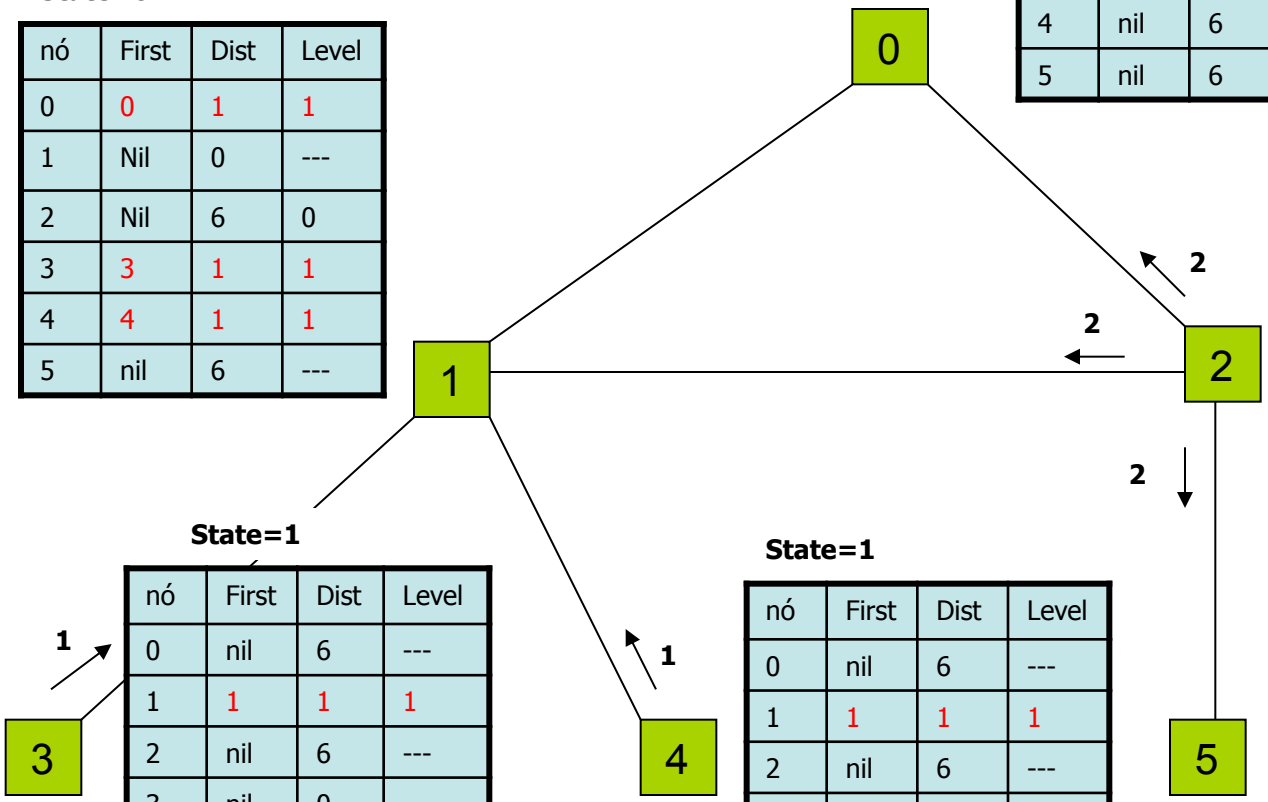
State=0

nó	First	Dist	Level
0	nil	6	---
1	nil	6	0
2	nil	6	---
3	nil	6	---
4	nil	0	---
5	nil	6	---

State=0

nó	First	Dist	Level
0	nil	6	---
1	nil	6	---
2	nil	6	0
3	nil	6	---
4	nil	6	---
5	nil	0	---

Distância Mínima – Execução



State=0

nó	First	Dist	Level
0	nil	0	---
1	1	1	1
2	Nil	6	0
3	nil	6	---
4	nil	6	---
5	nil	6	---

State=0

nó	First	Dist	Level
0	0	1	1
1	Nil	0	---
2	Nil	6	0
3	3	1	1
4	4	1	1
5	nil	6	---

State=0

nó	First	Dist	Level
0	0	1	1
1	1	1	1
2	nil	0	---
3	nil	6	---
4	nil	6	---
5	5	1	1

State=1

nó	First	Dist	Level
0	nil	6	---
1	1	1	1
2	nil	6	---
3	nil	0	---
4	nil	6	---
5	nil	6	---

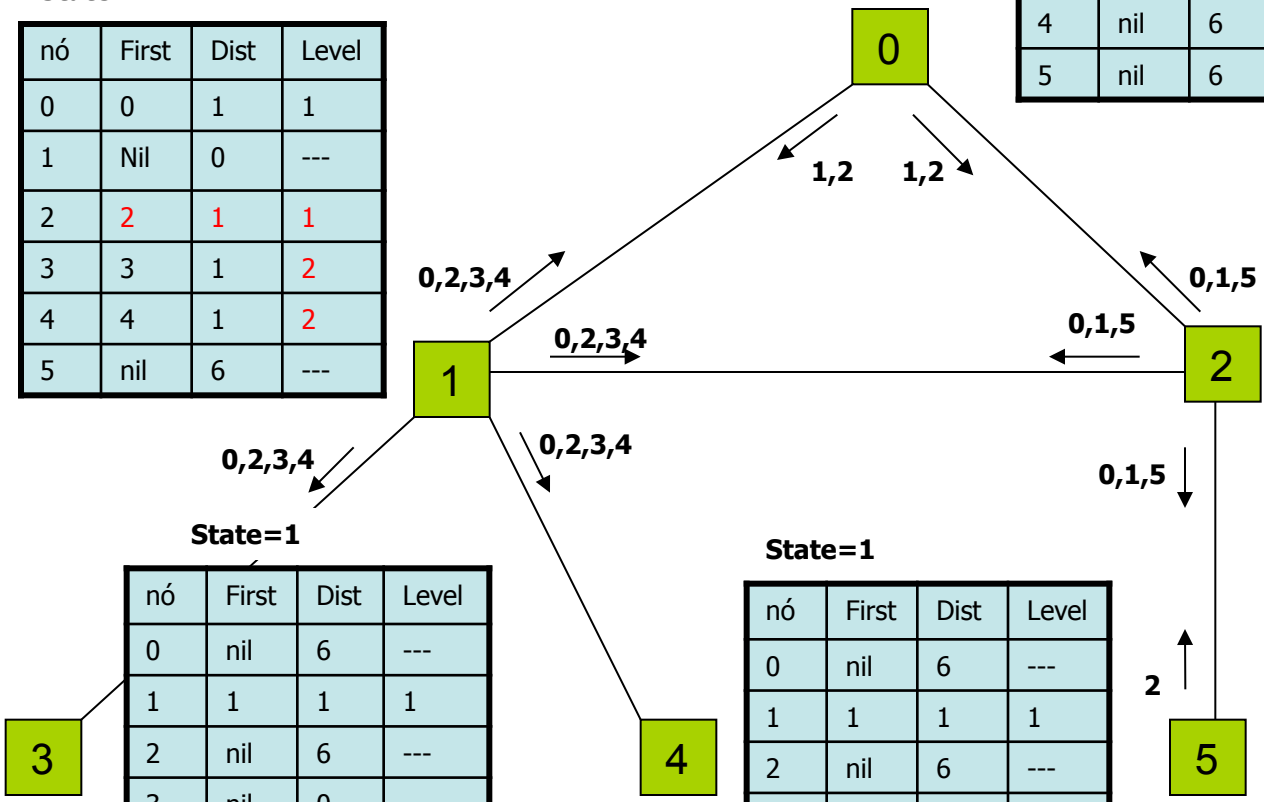
State=1

nó	First	Dist	Level
0	nil	6	---
1	1	1	1
2	nil	6	---
3	nil	6	---
4	nil	0	---
5	nil	6	---

State=0

nó	First	Dist	Level
0	nil	6	---
1	nil	6	---
2	Nil	6	0
3	nil	6	---
4	nil	6	---
5	nil	0	---

Distância Mínima – Execução



State=1

nó	First	Dist	Level
0	nil	0	---
1	1	1	1
2	2	1	1
3	nil	6	---
4	nil	6	---
5	nil	6	---

State=1

nó	First	Dist	Level
0	0	1	1
1	Nil	0	---
2	2	1	1
3	3	1	2
4	4	1	2
5	nil	6	---

State=1

nó	First	Dist	Level
0	0	1	1
1	1	1	1
2	nil	0	---
3	nil	6	---
4	nil	6	---
5	5	1	1

State=1

nó	First	Dist	Level
0	nil	6	---
1	1	1	1
2	nil	6	---
3	nil	0	---
4	nil	6	---
5	nil	6	---

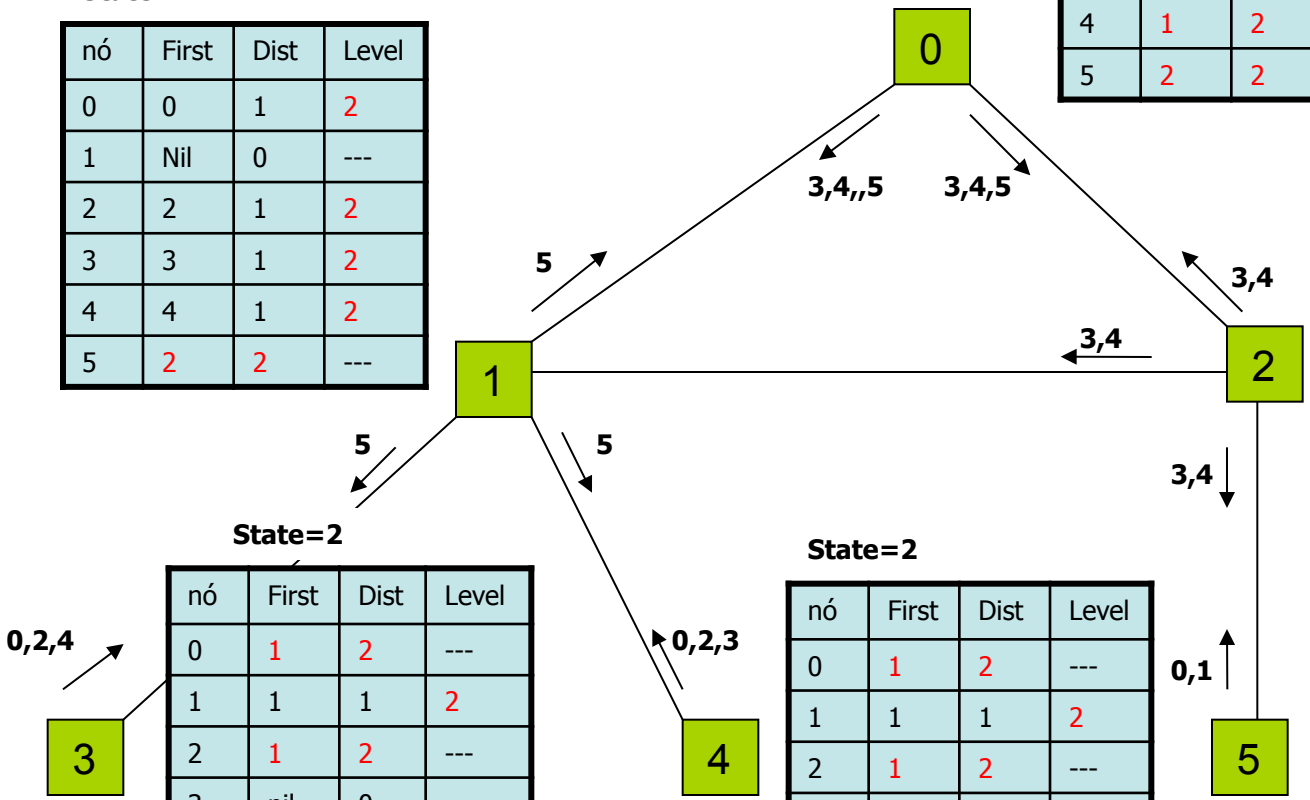
State=1

nó	First	Dist	Level
0	nil	6	---
1	1	1	1
2	nil	6	---
3	nil	6	---
4	nil	0	---
5	nil	6	---

State=1

nó	First	Dist	Level
0	nil	6	---
1	nil	6	---
2	2	1	1
3	nil	6	---
4	nil	6	---
5	nil	0	---

Distância Mínima - Execução



State=2

nó	First	Dist	Level
0	nil	0	---
1	1	1	2
2	2	1	2
3	1	2	---
4	1	2	---
5	2	2	---

State=2

nó	First	Dist	Level
0	0	1	2
1	Nil	0	---
2	2	1	2
3	3	1	2
4	4	1	2
5	2	2	---

State=2

nó	First	Dist	Level
0	0	6	2
1	1	6	2
2	nil	0	---
3	1	2	---
4	1	2	---
5	5	6	2

State=2

nó	First	Dist	Level
0	1	2	---
1	1	1	2
2	1	2	---
3	nil	0	---
4	1	2	---
5	nil	6	---

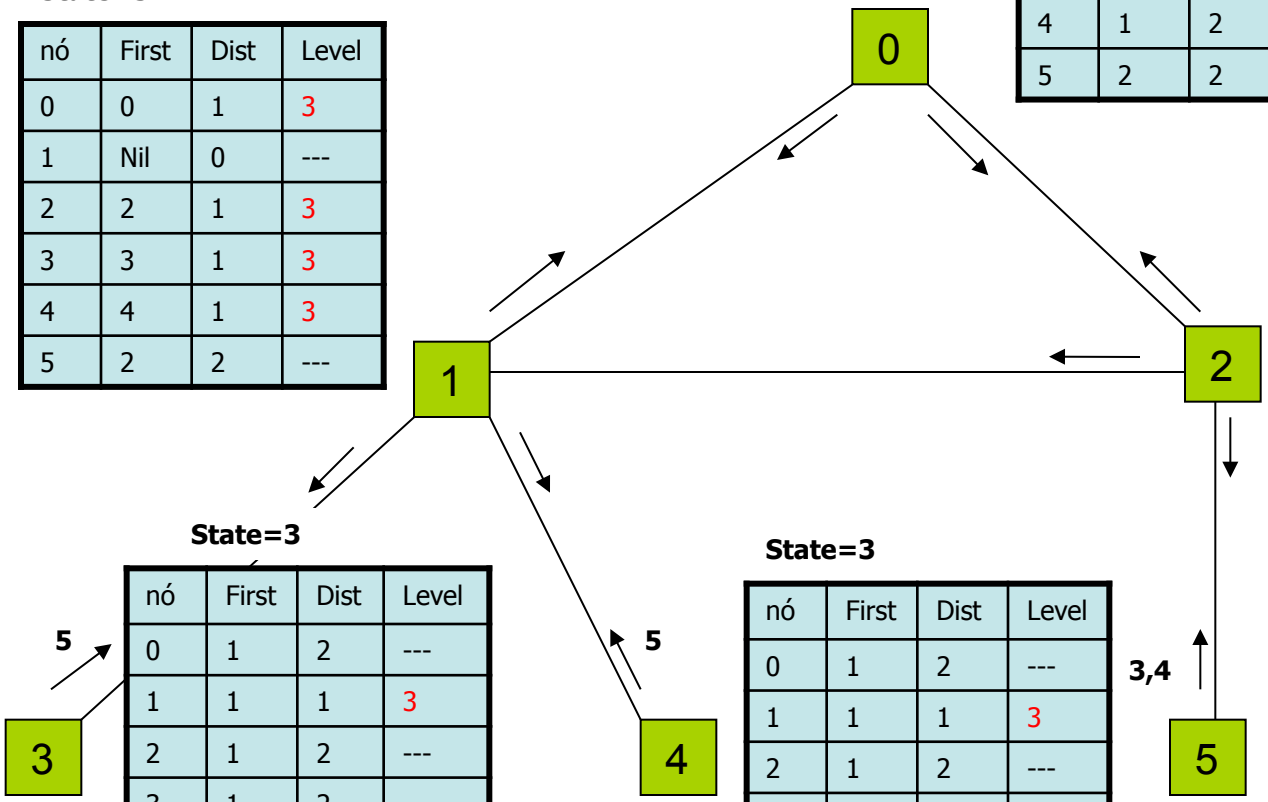
State=2

nó	First	Dist	Level
0	1	2	---
1	1	1	2
2	1	2	---
3	1	2	---
4	nil	0	---
5	nil	6	---

State=2

nó	First	Dist	Level
0	2	2	---
1	2	2	---
2	2	1	2
3	nil	6	---
4	nil	6	---
5	nil	0	---

Distância Mínima - Execução



State=3

nó	First	Dist	Level
0	nil	0	---
1	1	1	3
2	2	1	3
3	1	2	---
4	1	2	---
5	2	2	---

State=3

nó	First	Dist	Level
0	0	1	3
1	Nil	0	---
2	2	1	3
3	3	1	3
4	4	1	3
5	2	2	---

State=3

nó	First	Dist	Level
0	0	6	3
1	1	6	3
2	nil	0	---
3	1	2	---
4	1	2	---
5	5	6	3

State=3

nó	First	Dist	Level
0	1	2	---
1	1	1	3
2	1	2	---
3	1	2	---
4	1	2	---
5	1	3	---

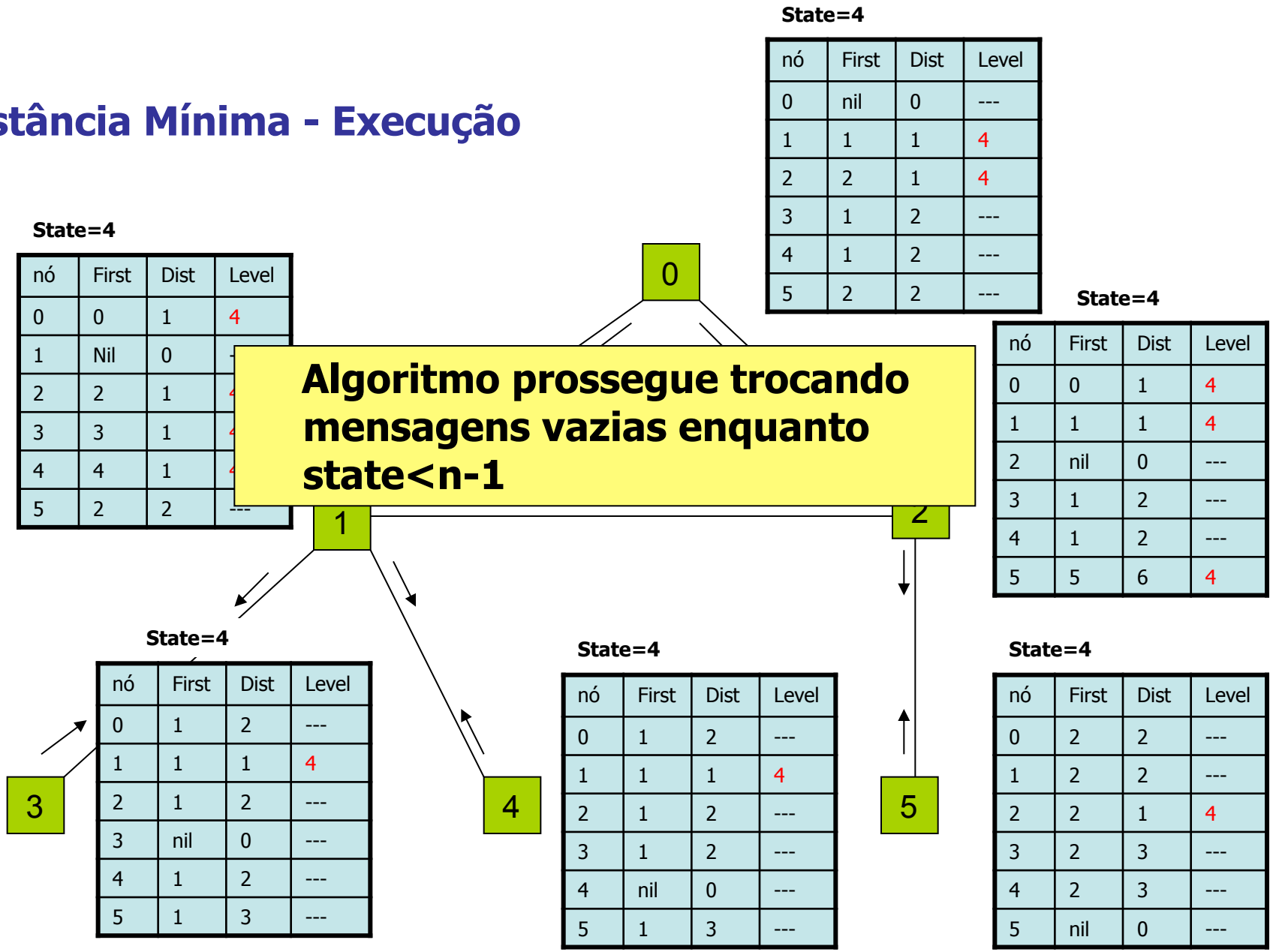
State=3

nó	First	Dist	Level
0	1	2	---
1	1	1	3
2	1	2	---
3	1	2	---
4	nil	0	---
5	1	3	---

State=3

nó	First	Dist	Level
0	2	2	---
1	2	2	---
2	2	1	3
3	2	3	---
4	2	3	---
5	nil	6	---

Distância Mínima - Execução



Distância Mínima – Implementação

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

/*Definir o grafo da aplicação antes de executar*/
int numeroDeTarefas = 6;
int matrizVizinhanca[6][6] = {    {0,1,1,0,0,0},
                                  {1,0,1,1,1,0},
                                  {1,1,0,0,0,1},
                                  {0,1,0,0,0,0},
                                  {0,1,0,0,0,0},
                                  {0,0,1,0,0,0}
                                };
```

Distância Mínima – Implementação

```
/*retorna o número de vizinhos da tarefa myRank*/
int contaNumeroDeVizinhos(int myRank)
{
   int i;
   int contador = 0;

   for (i = 0; i < numeroDeTarefas; i++)
       if (matrizVizinhanca[myRank][i] == 1)
              contador++;

   return contador;
}
```

Distância Mínima – Implementação

```
/*programa principal*/
int main(int argc, char* argv[])
{
   int i, j, contador;
   int numeroDeVizinhos;
   int myRank;
   int source;
   int tag=50;
   int pai;
   MPI_Status status;
   int origem;
   int state;//marca o pulso atual deste processo
   int dist[numeroDeTarefas];//distância
   int first[numeroDeTarefas];//primeiro nó no caminho
   int set[numeroDeTarefas];
   int level[numeroDeTarefas];//pulso dos meus vizinhos
```

Distância Mínima – Implementação

```
//inicialização do MPI
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

numeroDeVizinhos = contaNumeroDeVizinhos(myRank);
printf("iniciando...\n");
fflush(stdout);
//inicializando vetor dist e first
for (i=0; i<numeroDeTarefas; i++)
{   dist[i]=numeroDeTarefas;//considerado como infinito
    first[i]=0;
    set[i]=0;
    level[i]=0;
}
dist[myRank]=0;
set[myRank]=1;
state=0;
```

Distância Mínima – Implementação

```
//enviando meu conjunto para os meus vizinhos
for (i = 0; i < numeroDeTarefas; i++)
    if (matrizVizinhanca[myRank][i] == 1)
    {
          //pede aos vizinhos para enviar
          MPI_Send(set, numeroDeTarefas, MPI_INT,
                i, tag, MPI_COMM_WORLD);
    }
```

Distância Mínima – Implementação

```
while (state < numeroDeTarefas)
{
    MPI_Recv(set, numeroDeTarefas, MPI_INT,
             MPI_ANY_SOURCE, tag, MPI_COMM_WORLD,
             &status);
    origem=status.MPI_SOURCE;
    level[origem]++;
    for (i=0; i<numeroDeTarefas; i++)
    {
        if (set[i]==1)
        {
            if (dist[i]>level[origem])
            {
                dist[i]=level[origem];
                first[i]=origem;
            }
```

Distância Mínima – Implementação

```
//testando se meu state é menor ou igual ao
//level de todos os meus vizinhos
int continua=1;
for (j=0; j<numeroDeTarefas; j++)
if (matrizVizinhanca[myRank][j]==1 &&
    state>=level[j])
      {
            continua=0;
            break;
      }
```

Distância Mínima – Implementação

```
   if (continua){
        state++;
        //se minha distancia até j for igual a state,
        //então o adiciono a set
        for (j=0; j<numeroDeTarefas; j++){
                if (dist[j]==state)
                        set[j]=1;
        }
        for (j=0; j<numeroDeTarefas; j++)
           if (matrizVizinhanca[myRank][j]==1)
              MPI_Send(set, numeroDeTarefas, MPI_INT, j,
                       tag, MPI_COMM_WORLD);
    }
   }
  }
 printf("processo %d: state=%d\n", myRank, state);
 fflush(stdout);
}
```

Distância Mínima – Implementação

```
    //imprimindo as distâncias
   printf("processo %d: ", myRank);
   for (i=0; i<numeroDeTarefas; i++)
       printf("dist(%d)=%d, ", i, dist[i]);
   printf("\n");
   printf("processo %d: ", myRank);
   for (i=0; i<numeroDeTarefas; i++)
       printf("first(%d)=%d, ", i, first[i]);
   printf("\n");
   fflush(stdout);
   //Finalização do MPI
   MPI_Finalize();
}
```

Exercício

- Desenvolva uma aplicação paralela em MPI para multiplicar duas matrizes.

Referências

- [1] N. MacDonald et alli, Writing Message Passing Programs with MPI, Edinburgh Parallel Computer Centre Englewood Cliffs, NJ, Prentice--Hall, 1988
- [2] P. S. Pacheco, Parallel Programming with MPI, Morgan Kaufman Pub, 1997
- [3] Message Passing Interface Forum, MPI: A Message Passing Interface Standard, International Journal of Supercomputer Applications, vol. 8, n. 3/4, 1994
- [4] Valmir C. Barbosa, An Introduction to Distributed Algorithms, MIT Press, 1996
- [5] Nancy A. Lynch, Distributed Algorithms, Morgan Kaufman Pub, 1996
- [7] M. J. Quinn, Parallel Computing Theory and Practice, 2nd edition, McGraw-Hill, 1994.
- [8] Ian Foster, Designing and Building Parallel Programs, Addison-Wesley, 1995.
- [9] S. Akl, Parallel Computation – Models and Methods, Prentice-Hall, 1997.