Curso de Informática – DCC-IM / UFRJ

Programação Paralela e Distribuída

Um curso prático

Mario J. Júnior **Gabriel P. Silva**

Colaboração: Adriano O. Cruz, Julio S. Aude

Ementa

- Paradigma de Troca de Mensagens
  - PVM
  - MPI
- Paradigma de Memória Compartilhada
  - OpenMP
  - PThreads

Ementa

- Conceitos Básicos de concorrência e paralelismo. Conceitos de avaliação de desempenho. Modelos de programação paralela. Modelos de programação por troca de mensagens. Programação utilizando PVM e MPI. Exemplos.
- Modelos de programação com Memória Compartilhada. Conceitos de Thread e Processos. Primitivas de Sincronização em memória compartilhada. Algoritmos paralelos com memória compartilhada. Programação utilizando bibliotecas OpenMP e Pthreads. Exemplos. Ferramentas de avaliação e depuração de programas paralelos.

Conceitos Básicos de Paralelismo

Processo

- Um processo é criado para execução de um programa pelo sistema operacional.
- A criação de um processo envolve os seguintes passos:
  - Preparar o descritor de processo;
  - Reservar um espaço de endereçamento;
  - Carregar um programa no espaço reservado;
  - Passar o descritor de processo para o escalonador.
- Nos sistemas operacionais modernos, um processo pode gerar cópias, chamadas de processos “filhos”.

Processo

- Os processos *filhos* compartilham do mesmo código que os processos *pais*, contudo não compartilham o mesmo espaço de endereçamento.
- A troca de contexto entre processos é normalmente um processo lento, que exige uma chamada ao sistema operacional, com salvamento dos registradores e tabelas de gerência na memória, além da restauração dos mesmos para o processo que vai ser executado.

Threads

- Muitos dos sistemas operacionais modernos suportam o conceito de *threads*, ou seja, tarefas independentes dentro de um mesmo processo que podem ser executadas em paralelo ou de forma intercalada no tempo, concorrentemente.
- Nesses sistemas, uma aplicação é descrita por um processo composto de várias *threads*.
- As *threads* de um mesmo processo compartilham o espaço de endereçamento de memória, arquivos abertos e outros recursos que caracterizam o contexto global de um processo como um todo.

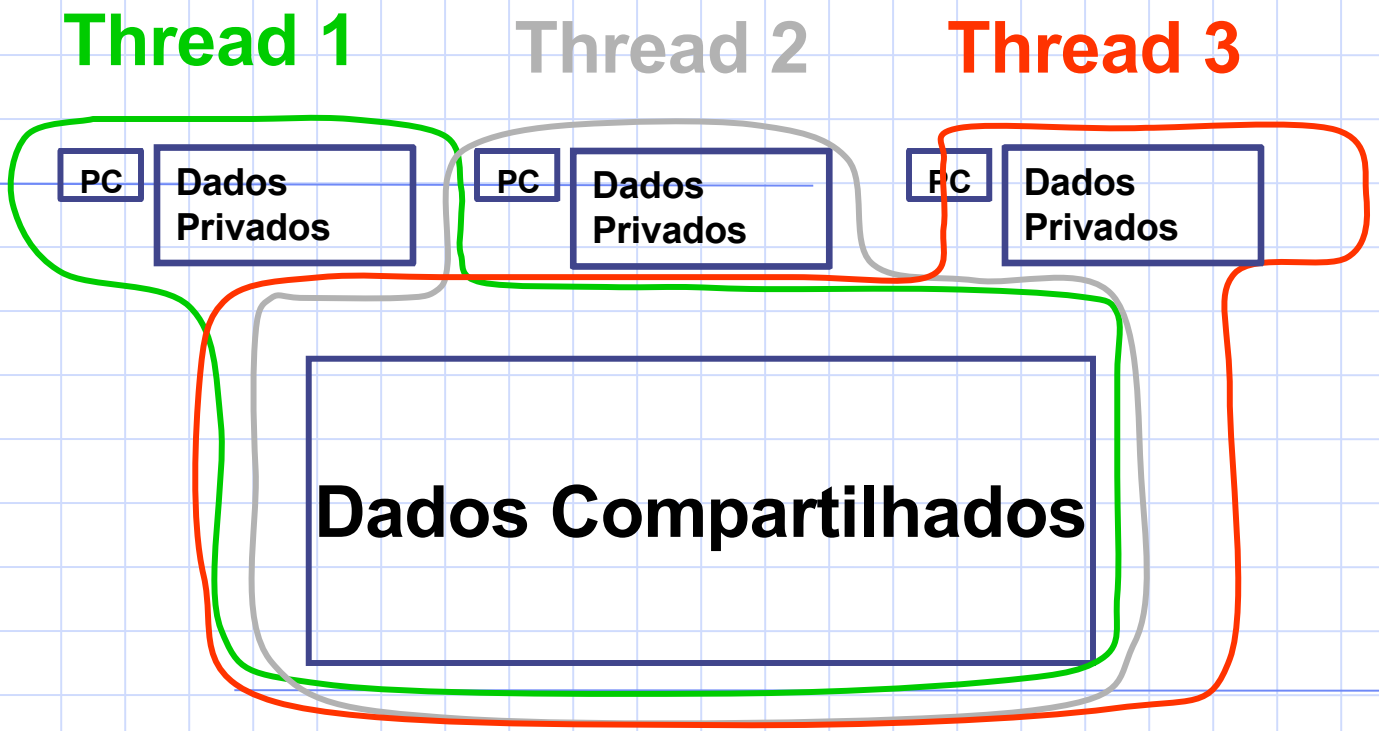
Threads

- Cada *thread*, no entanto, tem seu próprio contexto específico, normalmente caracterizado pelo conjunto de registradores em uso, contador de programas e palavra de *status* do processador.
- O contexto específico de uma *thread* é semelhante ao contexto de uma função e, conseqüentemente, a troca de contexto entre *threads* implica no salvamento e recuperação de contextos relativamente leves, a exemplo do que ocorre numa chamada de função dentro de um programa.

Threads

- Este fato é o principal atrativo em favor do uso de *threads* para se implementar um dado conjunto de tarefas em contraposição ao uso de múltiplos processos.
- A comunicação entre tarefas é grandemente simplificada numa implementação baseada em *threads*, uma vez que neste caso as tarefas compartilham o espaço de endereçamento de memória do processo como um todo que engloba as *threads*, eliminando a necessidade do uso de esquemas especiais, mais restritos e, usualmente, mais lentos de comunicação entre processos providos pelos sistemas operacionais.

Threads


Thread 1
Thread 2
Thread 3
PC
Dados Privados
PC
Dados Privados
PC
Dados Privados
Dados Compartilhados

Paralelismo x Concorrência

- Execução concorrente está associada ao modelo de um servidor atendendo a vários clientes através de uma política de escalonamento no tempo.
- Execução paralela está associada ao modelo de vários servidores atendendo a vários clientes simultaneamente no tempo.
- As linguagens de programação podem então ser classificadas como seqüenciais, concorrentes e paralelas.
- Estudaremos diversos padrões de bibliotecas de funções que permitem a codificação de programas paralelos utilizando linguagens convencionais (C, FORTRAN, JAVA, etc.).

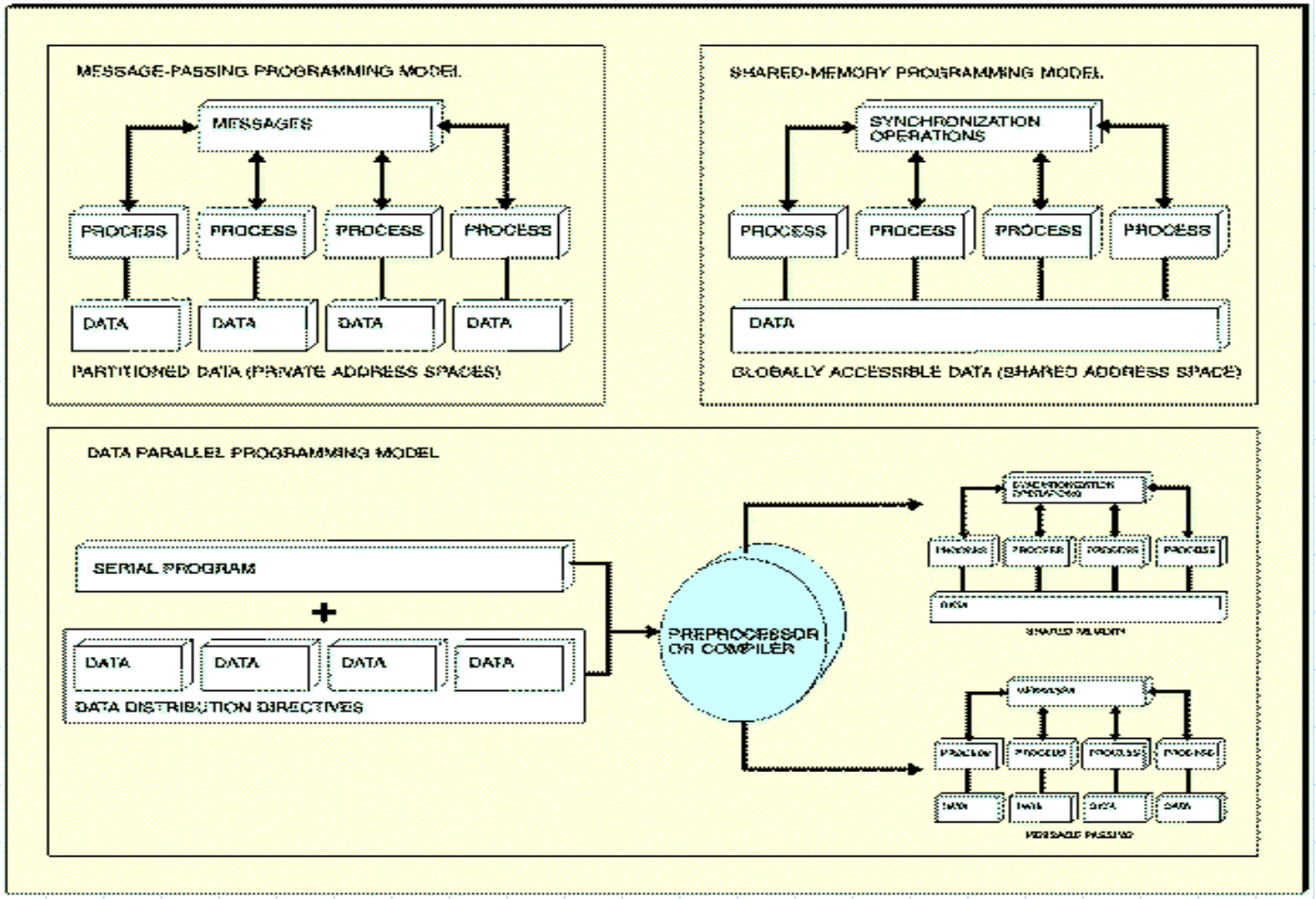
Paradigmas de Programação Paralela

Memória Compartilhada

- Comunicação através de variáveis em memória compartilhada.
- Uso de *threads* e/ou processos para mapeamento de tarefas.
- *Threads* ou processos podem ser criados de forma dinâmica ou estática.
- Sincronização
  - Exclusão Mútua
  - Barreira
- Alocação de Memória Privada e Global.

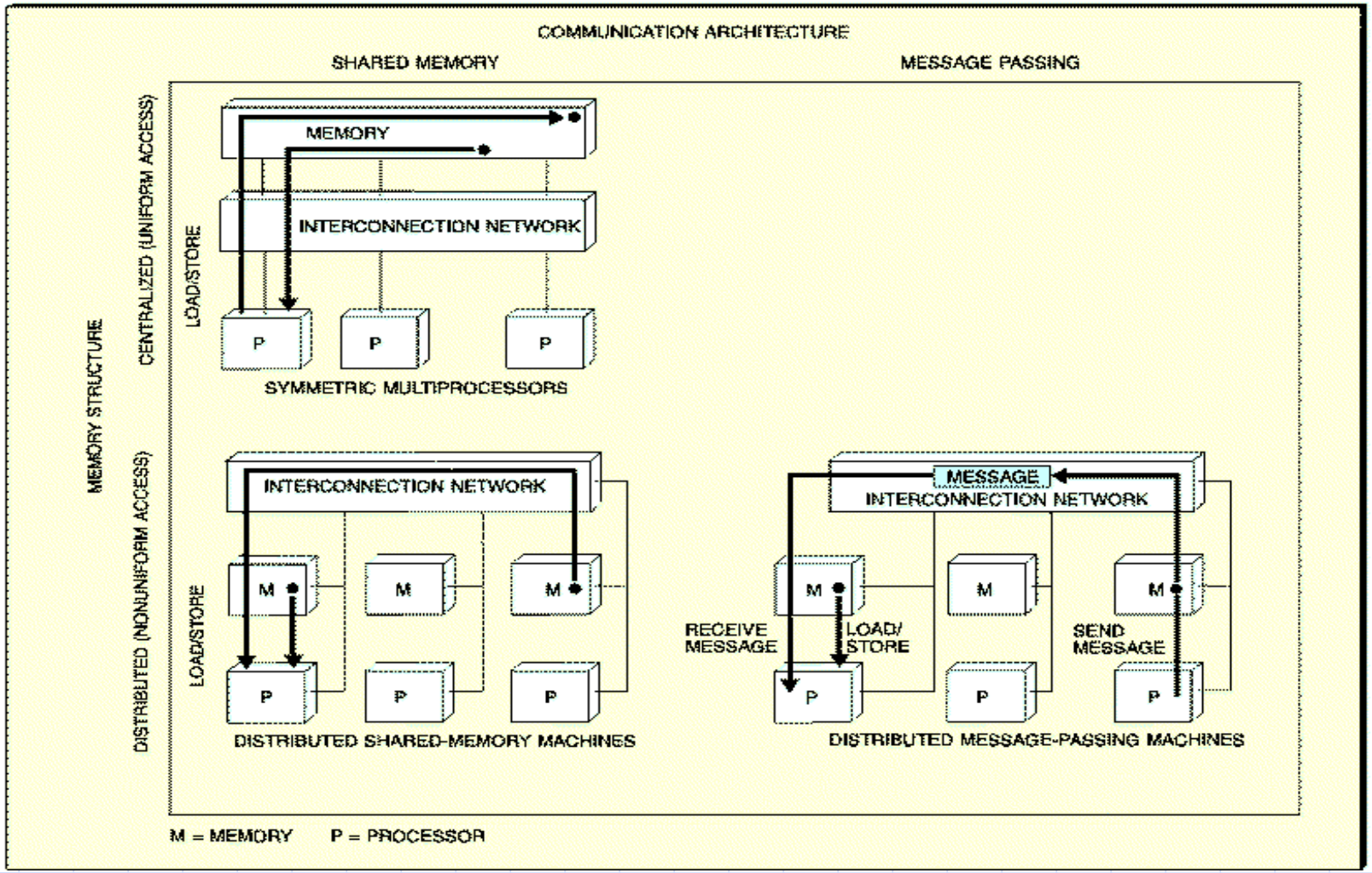
Modelos de Programação

Figure 1 Dominant programming models in parallel technical computing



Arquitetura de Comunicação

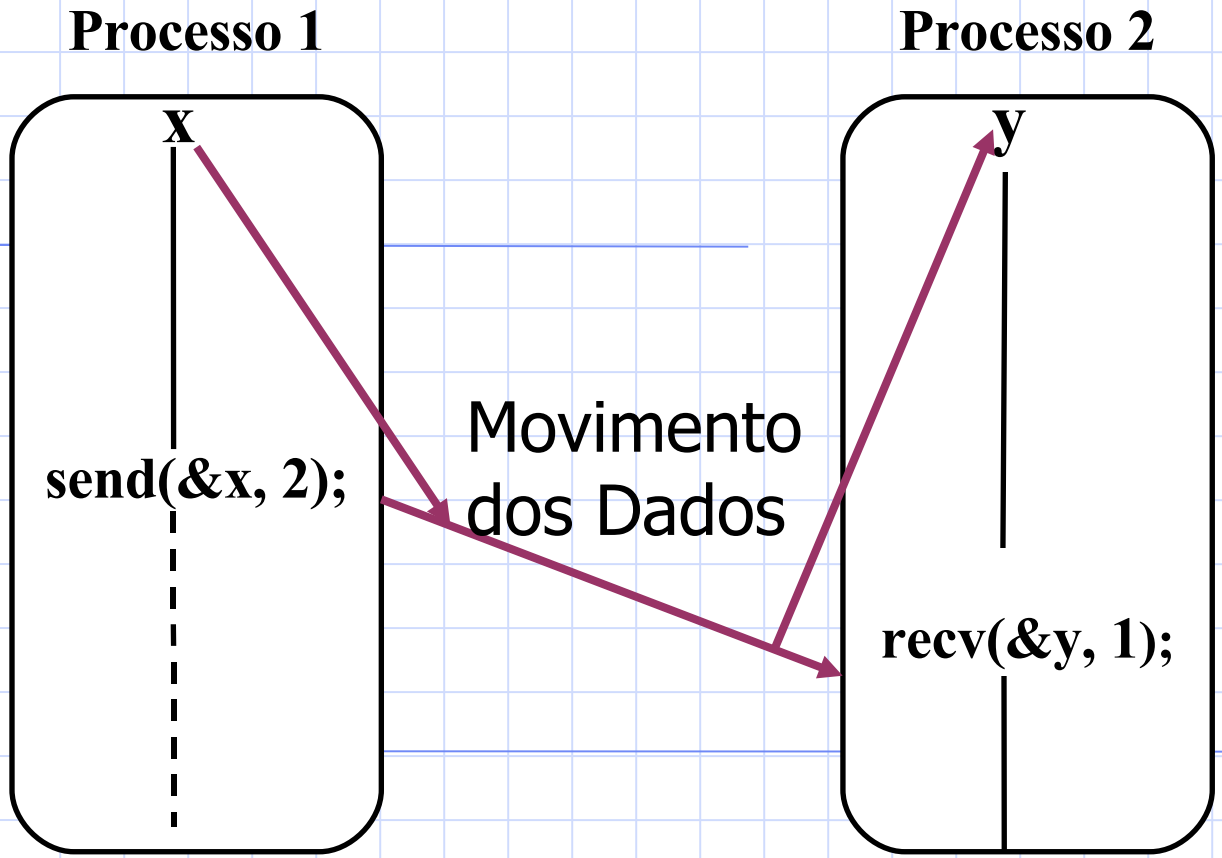
Figure 6 System architecture alternatives for scalable parallel systems



Troca de Mensagens

- Comunicação através do envio explícito de mensagens.
- Tarefas são mapeadas em processos, cada um com sua memória privada.
- Processos podem ser criados de forma estática ou dinâmica.
- Sincronização é feita de forma implícita pela troca de mensagens ou por operações coletivas de sincronização (barreiras).

Troca de Mensagens - Modelo



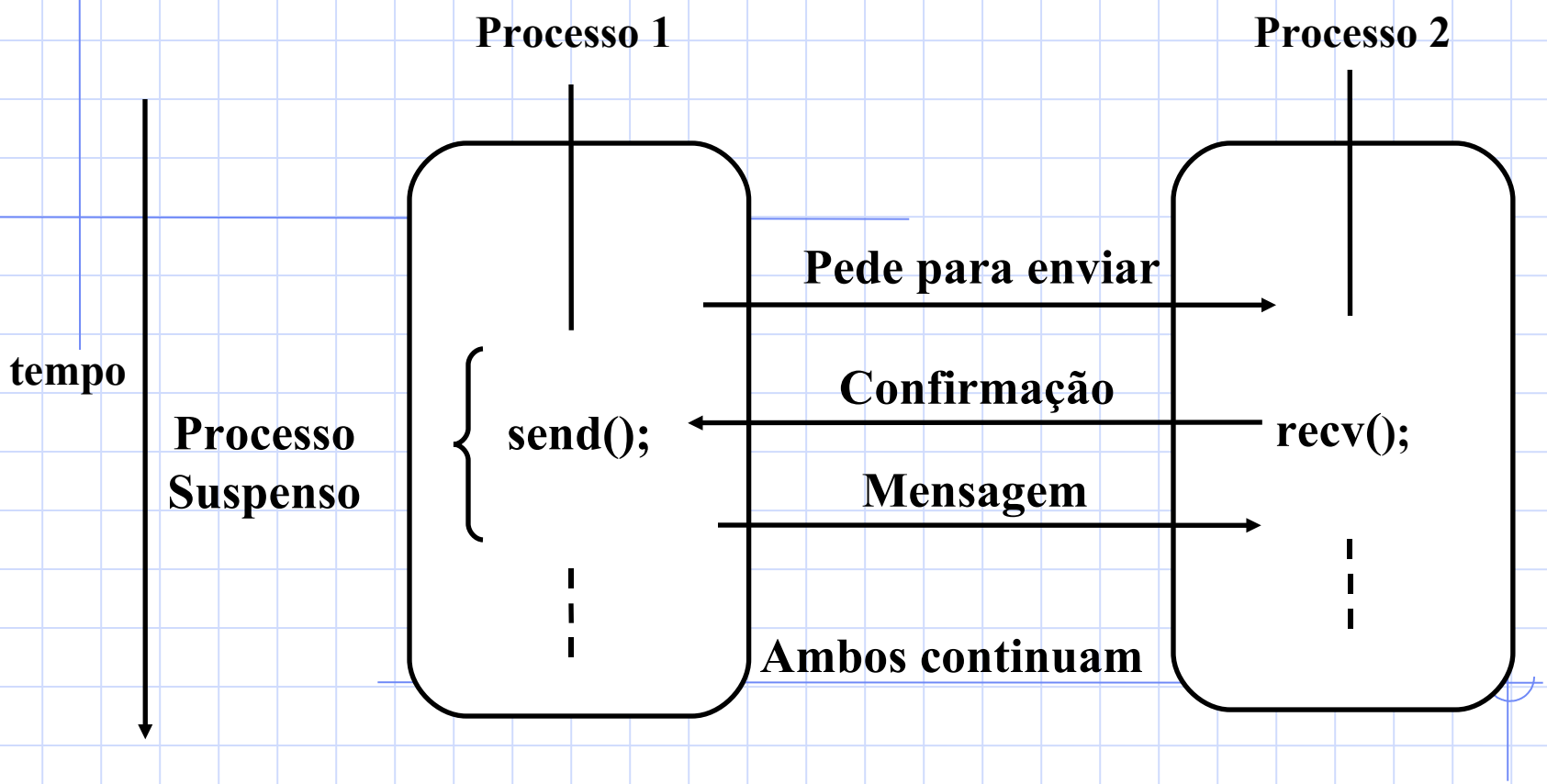
Comunicação Síncrona

- O processo que deseja enviar uma mensagem faz uma requisição inicial de autorização para enviar a mensagem ao processo destinatário.
- Depois de receber essa autorização, a operação de envio (*"send"*) é realizada.
- A operação de envio só se completa quando a operação de recepção (*"receive"*) correspondente retorna uma mensagem de *"acknowlegement"*.
- Portanto, ao se concluir a operação de envio, os *buffers* e estruturas de dados utilizados na comunicação podem ser reutilizados.

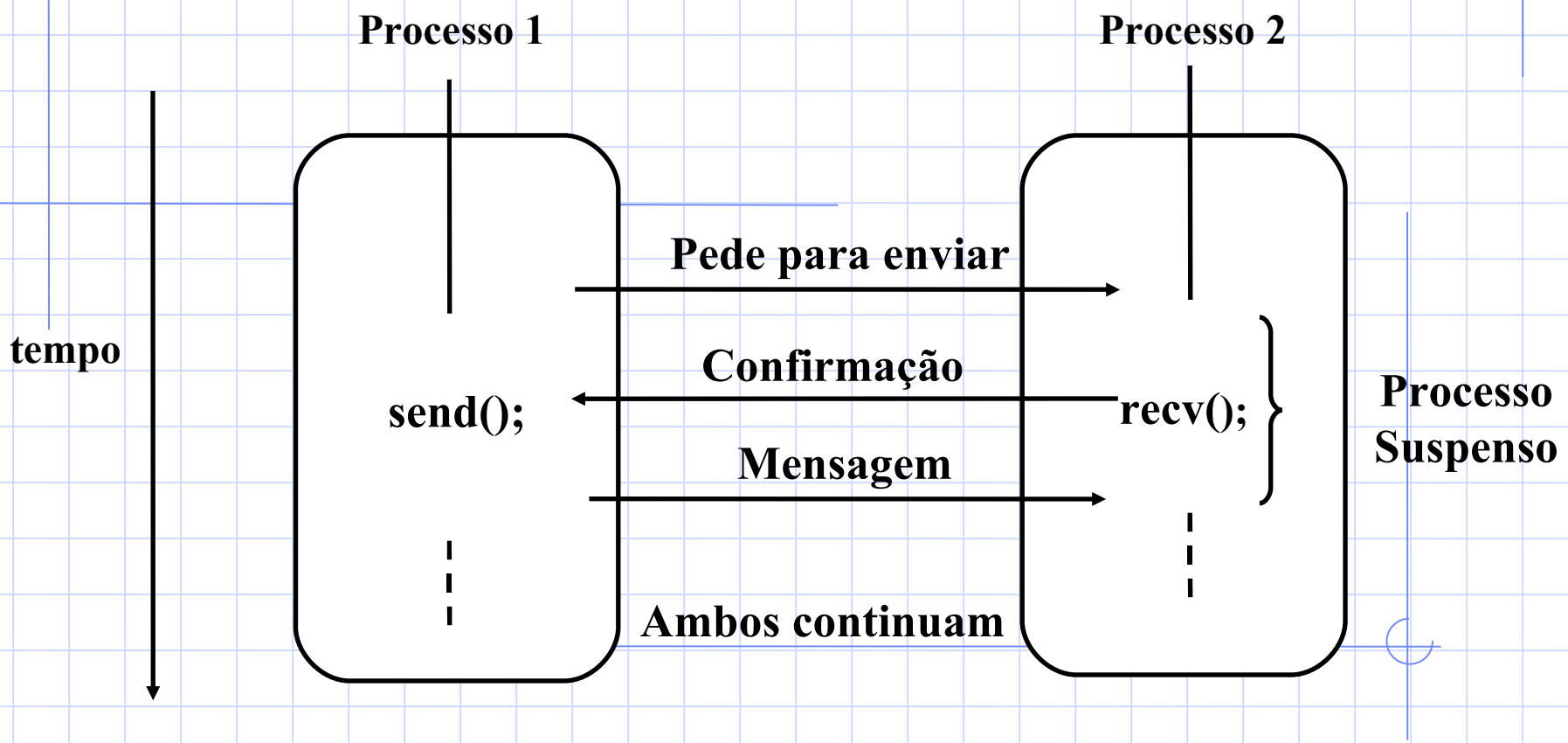
Comunicação Síncrona

- Este modo de comunicação é simples e seguro, contudo elimina a possibilidade de haver superposição entre o processamento da aplicação e o processamento da transmissão das mensagens.
- Requer um protocolo de quatro fases para a sua implementação do lado do transmissor: pedido de autorização para transmitir; recebimento da autorização; transmissão propriamente dita da mensagem; e recebimento da mensagem de *"acknowledgement"*.

Comunicação Síncrona



Comunicação Síncrona



Comunicação Assíncrona Bloqueante

- A operação de envio (*"send"*) só retorna o controle para o processo que a chamou após ter sido feita a cópia da mensagem a ser enviada de um *buffer* da aplicação para um *buffer* do sistema operacional.
- Portanto, ao haver o retorno da operação de envio, a aplicação está livre para reutilizar o seu *buffer*, embora não haja nenhuma garantia de que a transmissão da mensagem tenha completado ou vá completar satisfatoriamente.

Comunicação Assíncrona Bloqueante

- A operação de envio bloqueante difere da operação de envio síncrona, uma vez que não é implementado o protocolo de quatro fases entre os processos origem e destino.
- A operação de recepção (*"receive"*) bloqueante é semelhante à operação de recepção síncrona, só retornando para o processo que a chamou após ter concluído a transferência da mensagem do *buffer* do sistema para o *buffer* especificado pela aplicação. A diferença em relação a operação de recepção síncrona é que a mensagem de *"acknowledgement"* não é enviada.

Comunicação Assíncrona Não Bloqueante

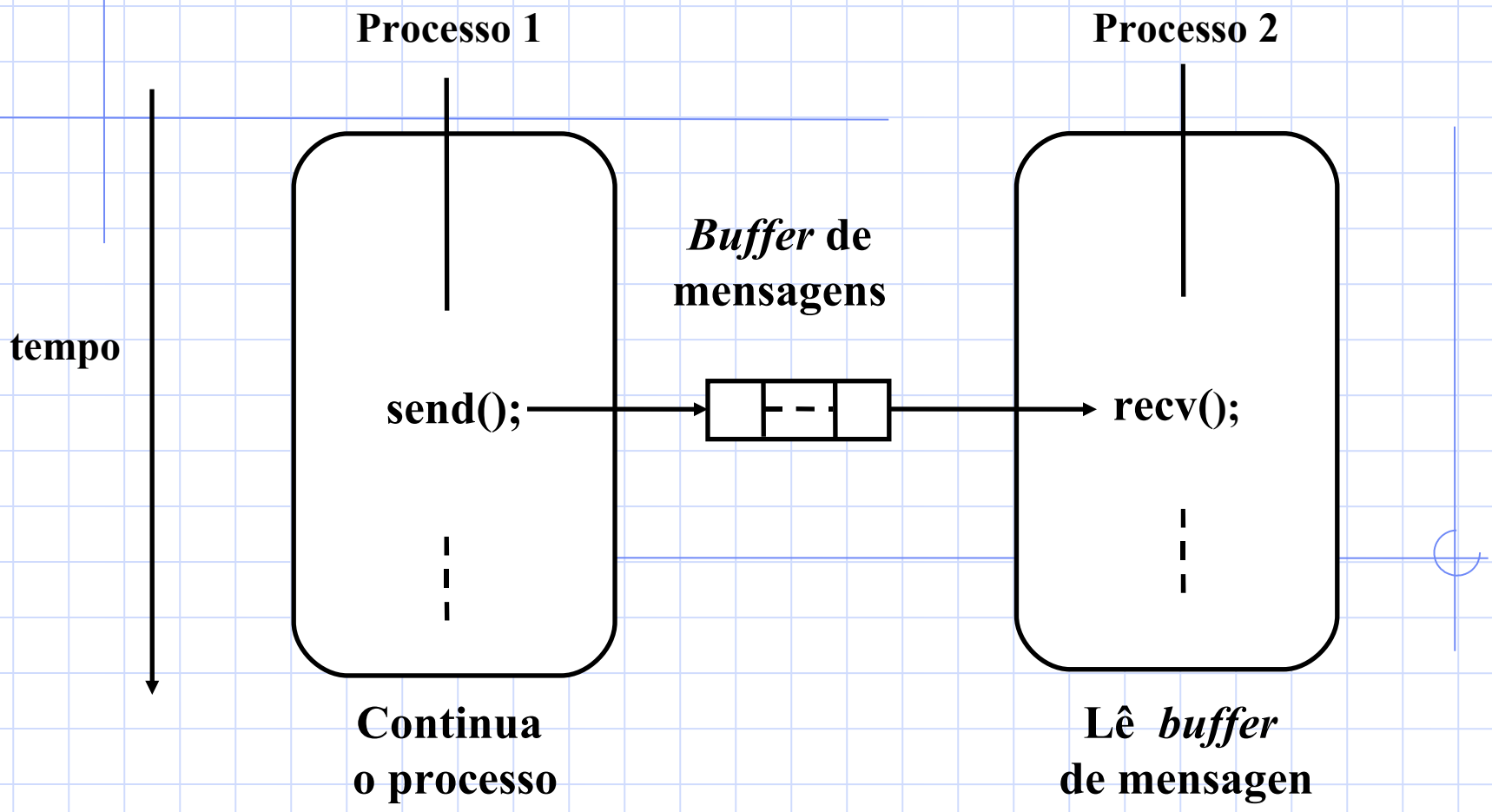
- Na primeira fase, a operação de *"send"* retorna imediatamente, tendo apenas dado início ao processo de transmissão da mensagem e a operação de *"receive"* retorna após notificar a intenção do processo de receber uma mensagem.
- As operações de envio e recepção propriamente ditas são realizadas de forma assíncrona pelo sistema.
- O retorno das operações de *"send"* e de *"receive"* nada garantem e não autoriza a reutilização das estruturas de dados para nova mensagem.

Comunicação Assíncrona Não Bloqueante

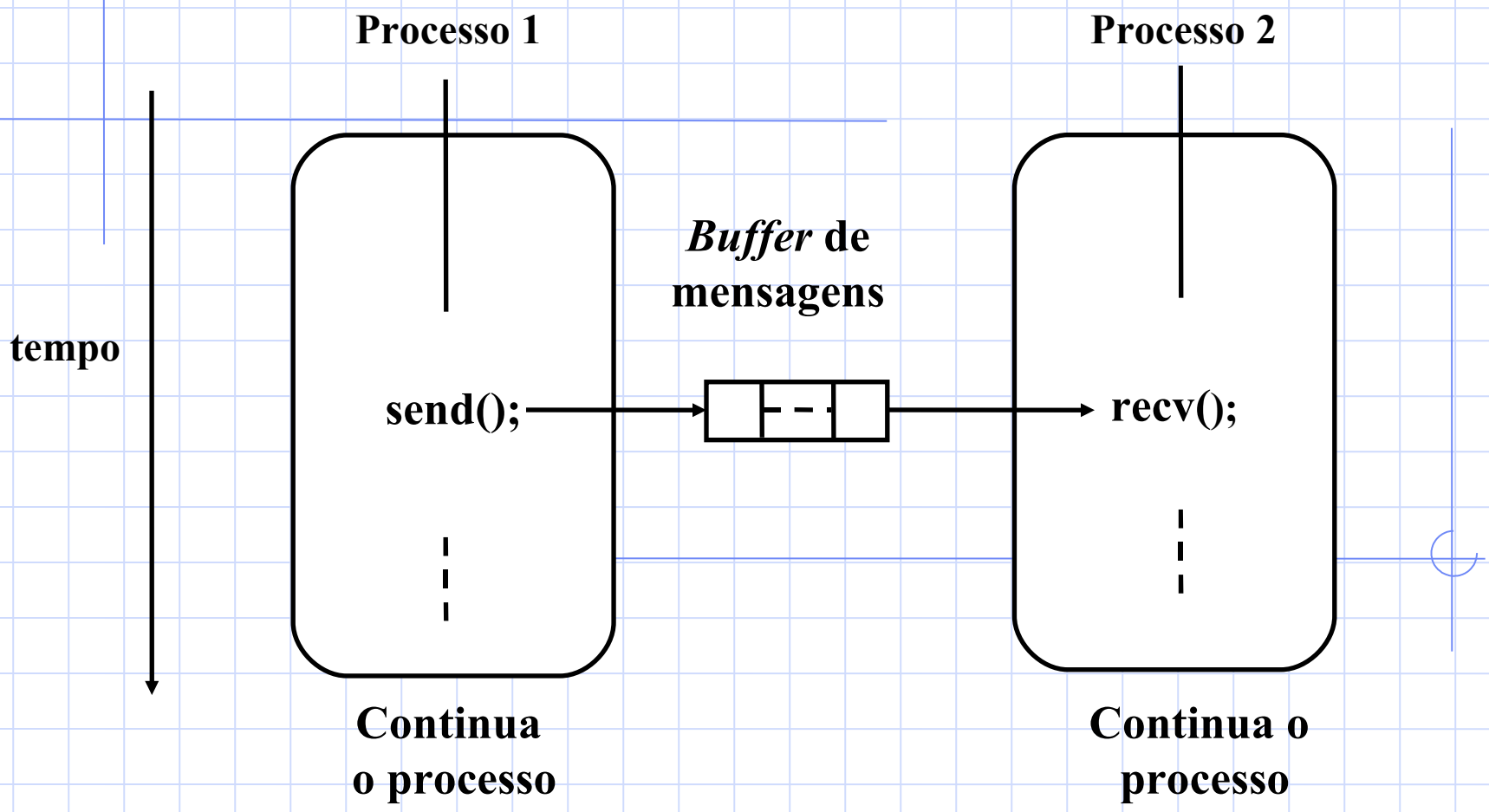
- Na segunda fase, é verificado se a operação de troca de mensagens iniciada anteriormente pelas primitivas *"send"* e *"receive"* já foi concluída.
- Somente após esta verificação é que o processo que realizou o envio de dados pode alterar com segurança sua área de dados original.
- Este modo de comunicação é o que permite maior superposição no tempo entre computações da aplicação e o processamento da transmissão das mensagens.

Comunicação Assíncrona

- Necessita de um *buffer* para guardar a mensagem



Comunicação Assíncrona



Etapas da Criação de um Programa Paralelo

Decomposição da Computação em Tarefas

- Busca expor suficiente paralelismo sem gerar uma sobrecarga de gerenciamento das tarefas que se torne significativo perante o trabalho útil a ser feito.
- Ou seja, as tarefas de comunicação e sincronização não podem ser mais complexas que as tarefas de computação. Se isto ocorrer, a execução em paralelo certamente será mais ineficiente do que a seqüencial.

Mapeamento nos Processadores

- Os processos e *threads* devem ser mapeadas para os processadores reais que existam no sistema.
- Para que isto seja feito de uma maneira eficiente, deve-se explorar a localidade da rede de interconexão, mantendo os processos relacionados alocados ao mesmo processador.
- Se a rede apresentar custos de comunicação variáveis, alocar os processos com maior carga de comunicação aos nós com menor custo de comunicação entre si.

Alocação de Tarefas

- As tarefas devem ser alocadas a processos ou threads de modo que:
  - Haja um balanceamento de carga adequado e a redução do tempo de espera por conta da sincronização;
  - Possa haver uma redução da comunicação entre processos ou *threads*;
  - A sobrecarga do gerenciamento dessa alocação, em tempo de execução, seja a mais reduzida possível (estática x dinâmica).

Implementação da Funcionalidade

- A implementação da funcionalidade dos processos ou *threads*:
  - Depende do modelo de programação adotado e da eficiência com que as primitivas deste modelo são suportadas pela plataforma;
  - Explora a localidade dos dados;
  - Redução da serialização no acesso a recursos compartilhados;
  - Redução do custo de comunicação e da sincronização visto pelos processadores;
  - Escalonar as tarefas de modo que aquelas que tenham uma maior cadeia de dependência sejam executadas mais cedo.

Avaliação de Desempenho

Medidas de Desempenho

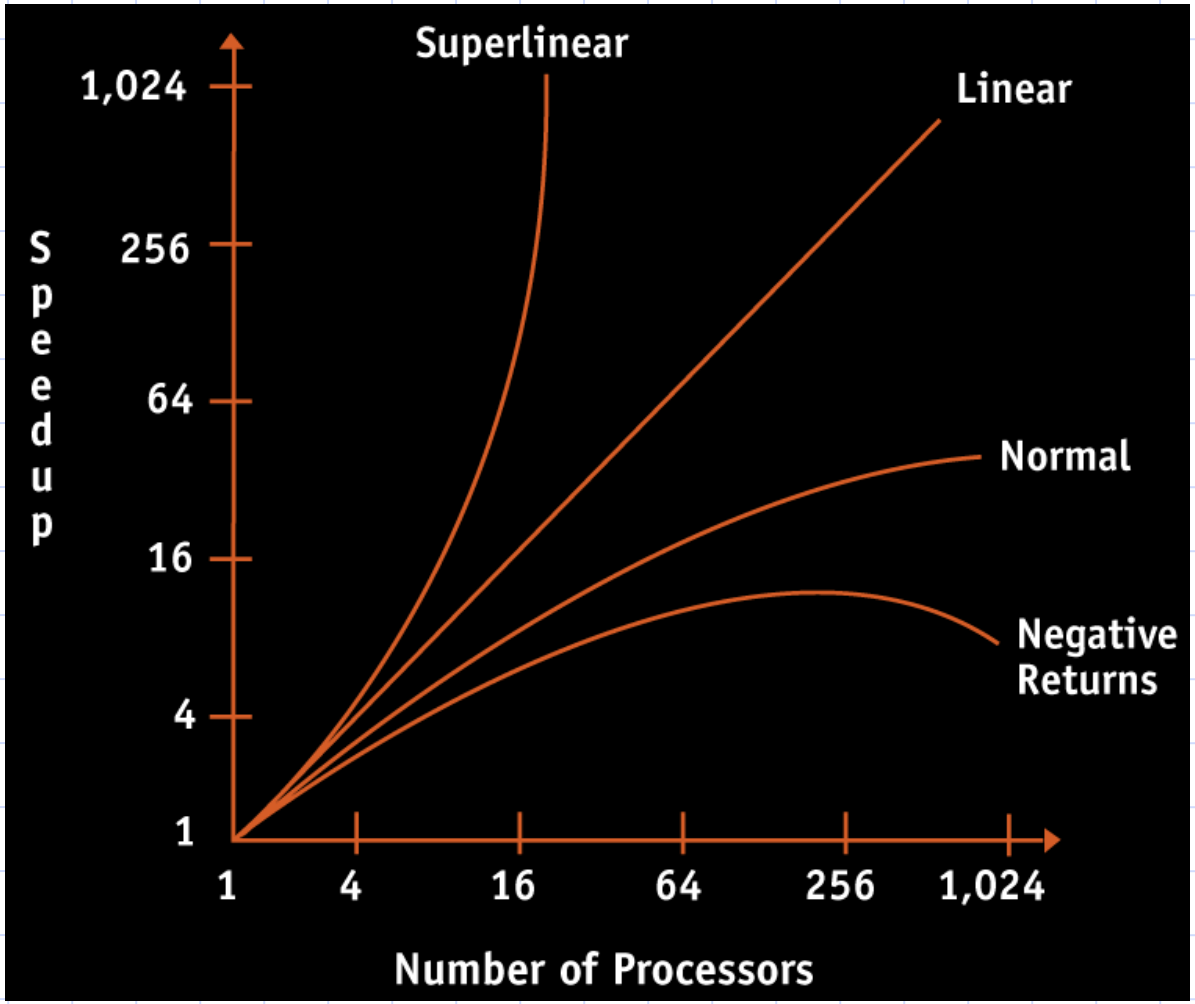
- **Speed-up (Aceleração):**
  - **Mede a razão entre o tempo gasto para execução de um algoritmo ou aplicação em um único processador e o tempo gasto na execução com *n* processadores:**

$$S(n) = T(1)/T(n)$$

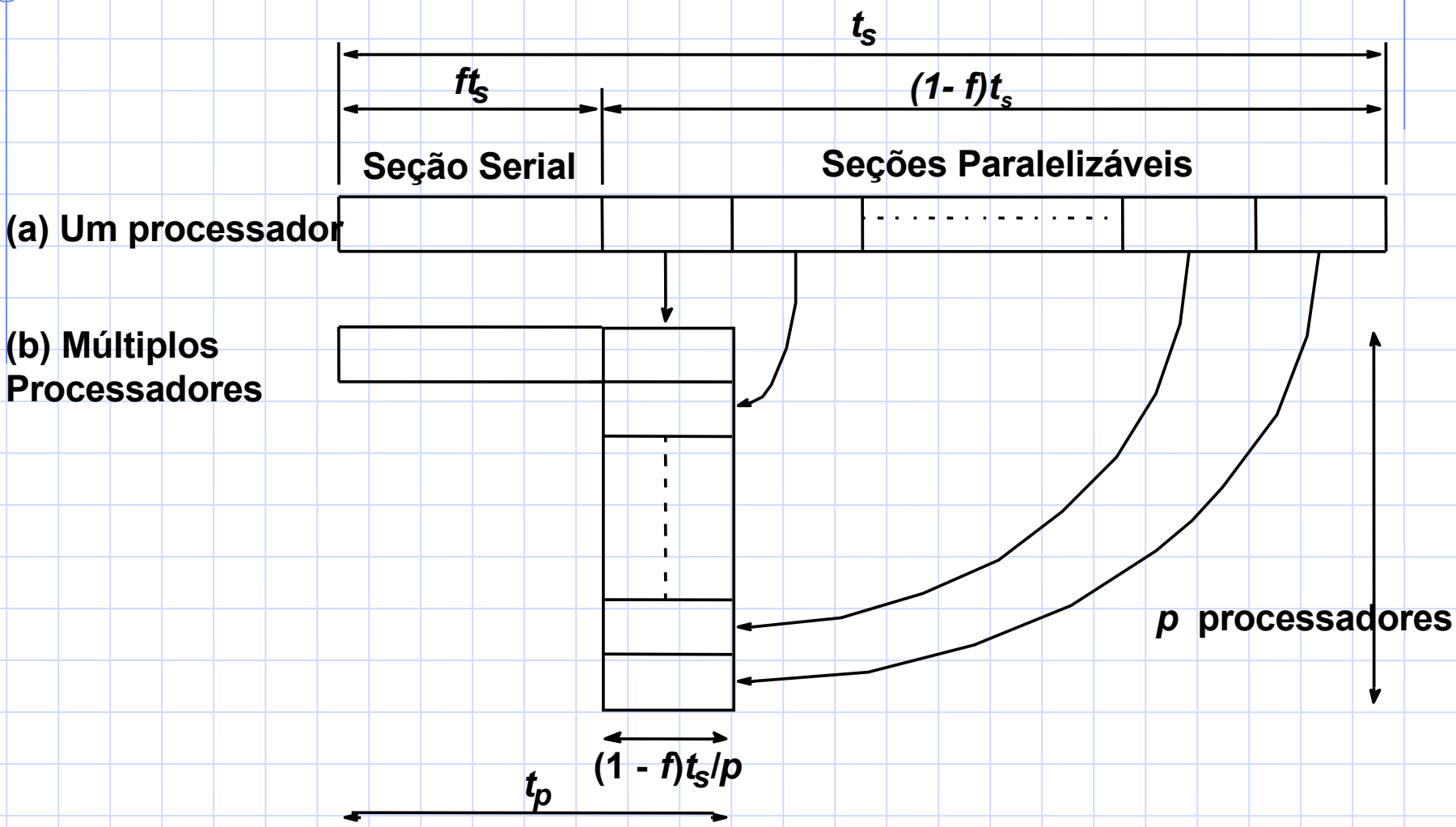
- **Eficiência:**

$$E(n) = S(n)/n$$

Speedup


Superlinear
Linear
Normal
Negative Returns
1,024
256
64
16
4
1
Speedup
1
4
16
64
256
1,024
Number of Processors

Lei de Amdahl



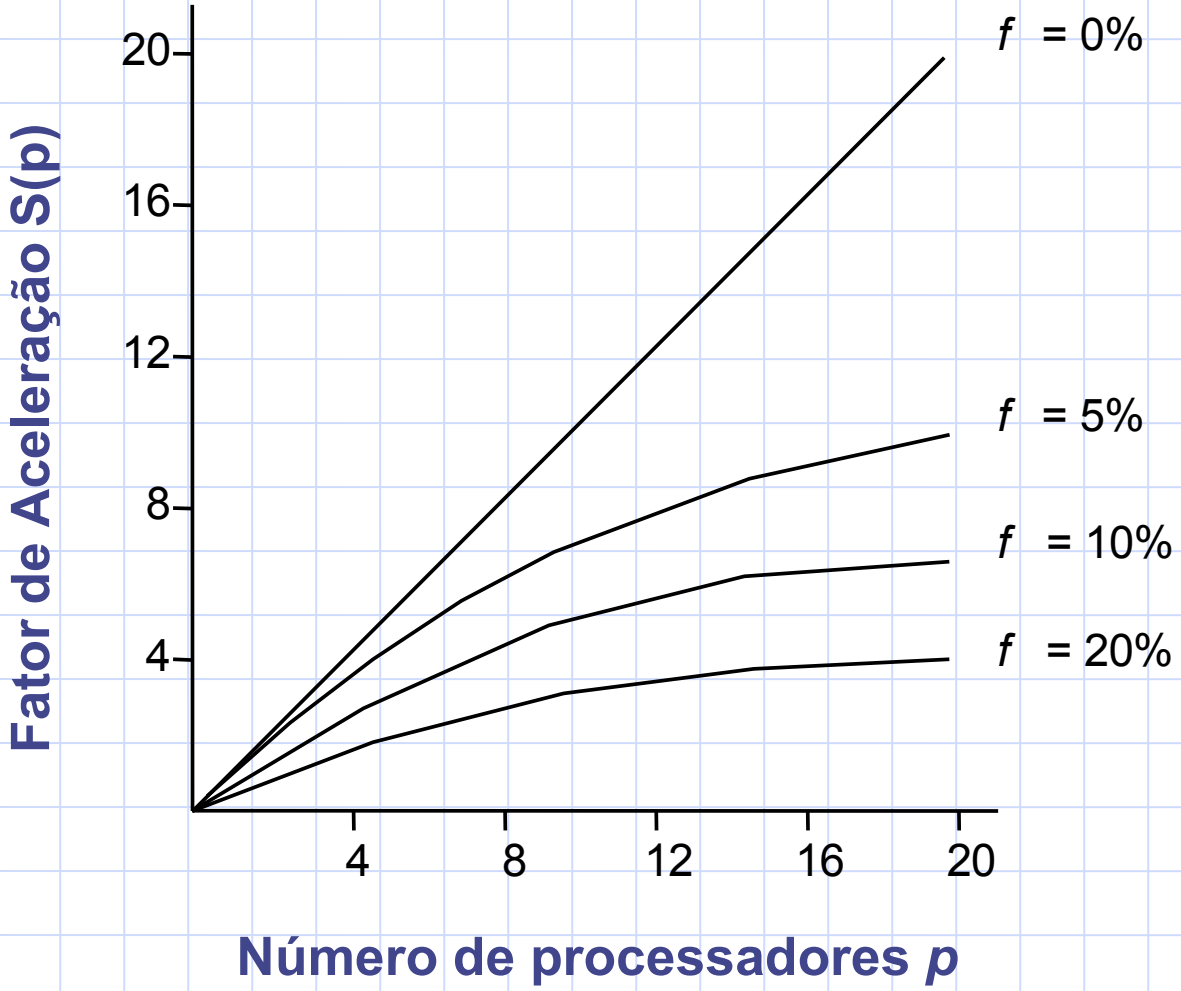
Lei de Amdahl

O fator de aceleração é dado por:

$$S(p) = \frac{t_s}{f t_s + (1-f) t_s / p} = \frac{p}{1 + (p-1) f}$$

Esta equação é conhecida como Lei de Amdahl

Speedup x Processadores



Speedup x Processadores

- Mesmo com um número infinito de processadores, o speedup máximo está limitado a 1/f , onde f é a fração serial do programa.
- Exemplo:
  - Com apenas 5% da computação sendo serial, o speedup máximo é 20, não interessando o número de processadores em uso.

Escalibidade

- Um sistema é dito *escalável* quando sua eficiência se mantém constante à medida que o número de processadores (*n*) aplicado à solução do problema cresce.
- Se o tamanho do problema é mantido constante e o número de processadores aumenta, o "overhead" de comunicação tende a crescer e a eficiência a diminuir.
- A análise da escalabilidade considera a possibilidade de se aumentar proporcionalmente o tamanho do problema a ser resolvido à medida que *n* cresce, de forma a contrabalançar o natural aumento do "overhead" de comunicação quando *n* cresce.

Escalibidade

- A análise da escalabilidade considera a possibilidade de se aumentar proporcionalmente o tamanho do problema a ser resolvido à medida que *n* cresce de forma a contrabalançar o natural aumento do "overhead" de comunicação quando *n* cresce.
- Um problema de tamanho P usando N processadores leva um tempo T para ser executado.
- O sistema é dito escalável se um problema de tamanho 2P em 2N processadores leva o mesmo tempo T.
- Escalabilidade é uma propriedade mais desejável que o speed-up.

Granulosidade

- ***Granulosidade*** **é a relação entre o quantidade de computação e a quantidade de comunicação.**
- **É uma medida de quanto trabalho é realizado antes dos processos se comunicarem.**
- **A granulosidade deve estar de acordo com o tipo de arquitetura:**

fine-grained vector/array processors

medium-grained shared memory multiprocessor

coarse/large-grained network of workstations

Bibliografia

- **PVM: Parallel Virtual Machine – A User´s Guide and Tutorial for Network Parallel Computing**
  - **Geist, A.; Beguelin, A.;Dongarra, J.; Jiang, W.; Mancheck, B.; Sunderam, V. The MIT Press, 1994**
- **Parallel Programming with MPI**
  - **Pacheco, P.S., Morgan Kaufmann Publishers, 1997.**
- **Using MPI-2: Advanced Features of the Message-Passing Interface**
  - **Gropp, W.; Lusk, E.; Thakur, R. The MIT Press, 1999.**
- **Parallel Computing: theory and pratice**
  - **Quinn, Michael J. – McGraw-Hill, 1994.**

Bibliografia

- **Parallel Programming in OpenMP**
  - Chandra, R. et alli; Morgan Kaufman Publishers, 2001
- **Pthreads Programming**
  - Nicholas, D.; Butlar, J.; Farell, P. – O`Reilly and Associates Inc., 1999
- **Programming with Threads**
  - Kleiman, S.; Shah, D.; Smaalders, B. Sun Soft Press, Prentice-Hall, 1996
- **Foundations of Multithreaded, Parallel and Distributed Programming**
  - Andrews, Gregory R. – Addison-Wesley, 2000
- **Designing and Building Parallel Programs**
  - Foster, Ian – Addison-Wesley Pub. Co, 1994

Homepage, Listas, E-mail

http://www.dcc.ufrj.br/~gabriel/progpar

e-mail: gabriel dot silva at ufrj dot br

grupo: progpar@listas.nce.ufrj.br

http://www.ufrj.br/mailman/listinfo/progpar

Avaliação

- **Quatro trabalhos (Individual, prazo de 15 dias)**
  - **22/08**
  - **19/09**
  - **17/10**
  - **21/11**
- **Quatro Provas**
  - **05/09**
  - **10/10**
  - **14/11**
  - **14/12**