Curso de Informática – DCC-IM / UFRJ

# Programação Paralela e Distribuída

Um curso prático

**Gabriel P. Silva**

# Ementa

- **Paradigma de Troca de Mensagens**
  - **MPI**
- **Paradigma de Memória Compartilhada**
  - **OpenMP**
- **Paradigma de SIMD**
  - **Cuda**

# Ementa

- **Conceitos de Thread e Processos. Conceitos Básicos de concorrência e paralelismo. Conceitos de avaliação de desempenho. Modelos de programação paralela. Modelos de programação por troca de mensagens. Modelos de programação com Memória Compartilhada.**
- **Programação utilizando MPI. Comunicação Síncrona e Assíncrona. Exemplos.**
- **Programação utilizando bibliotecas OpenMP. Primitivas de Sincronização em memória compartilhada. Algoritmos paralelos com memória compartilhada. Exemplos.**

# Ementa

- **CUDA Programming Model. CUDA Threads. CUDA Memories. Introduction to CUDA C. Optimizing Parallel Reduction in CUDA**

# 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.

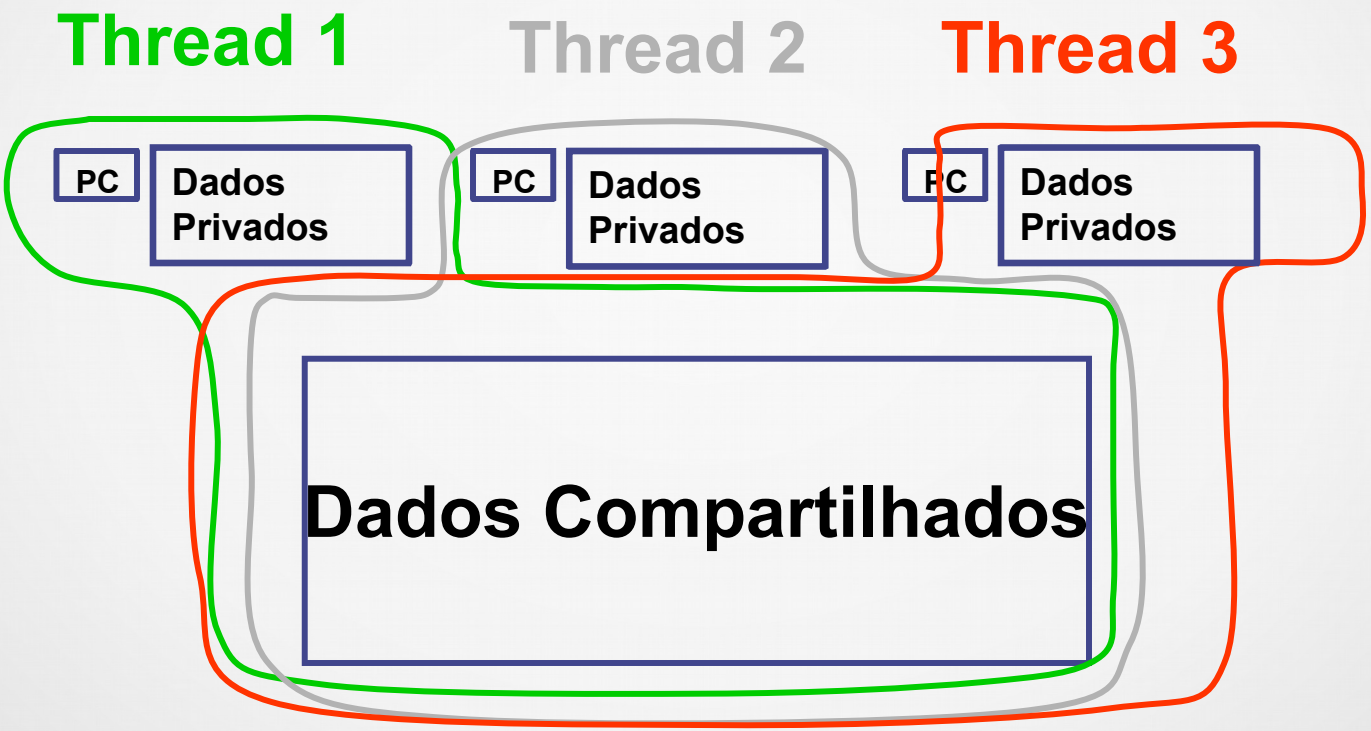
# Threads

- 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

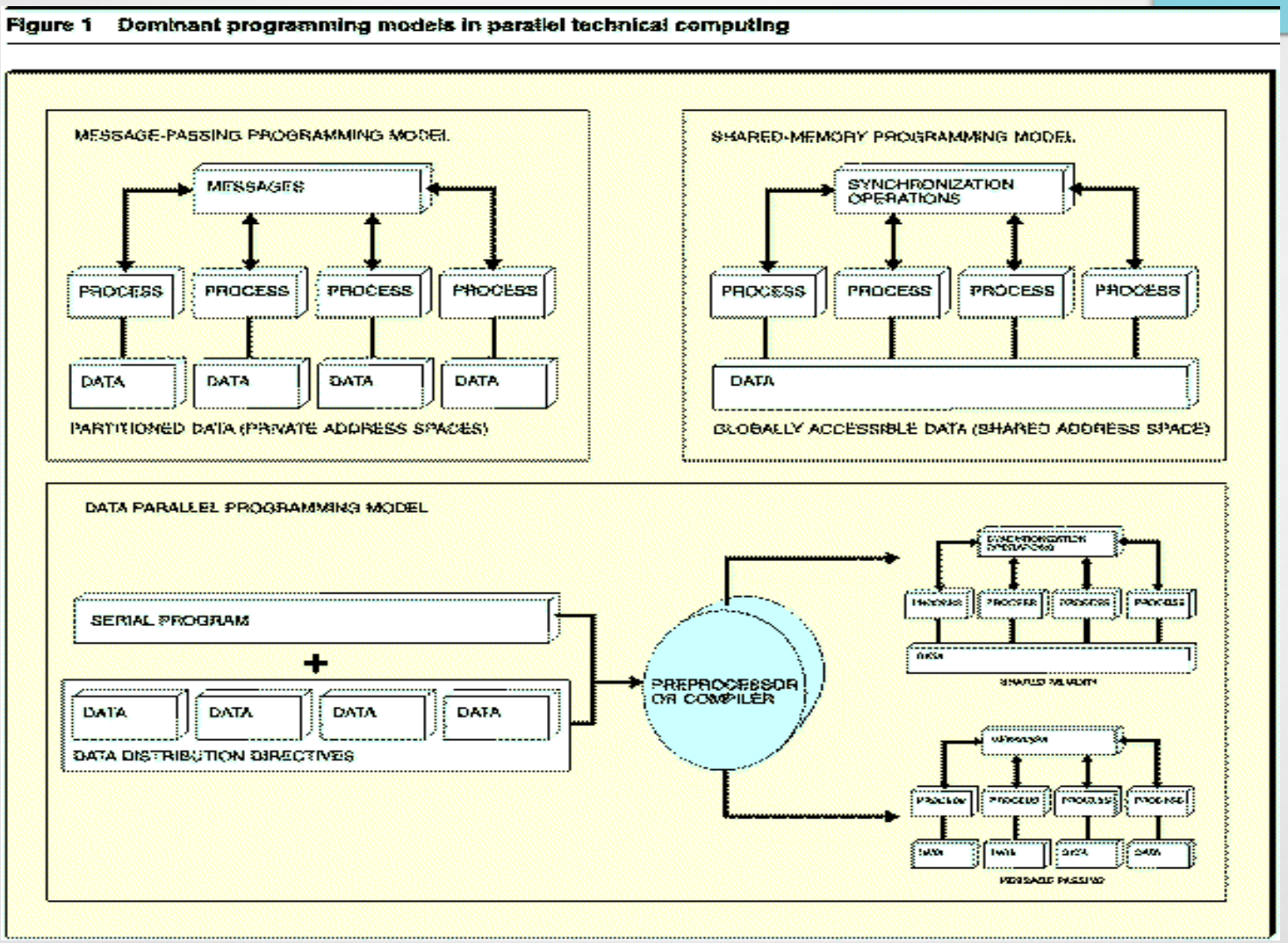
- Este fato é o principal atrativo em favor do uso de *threads* para se implementar um dado conjunto de tarefas em contraposição ao uso de múltiplos processos.
- A comunicação entre tarefas é grandemente simplificada numa implementação baseada em *threads*, uma vez que neste caso as tarefas compartilham  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

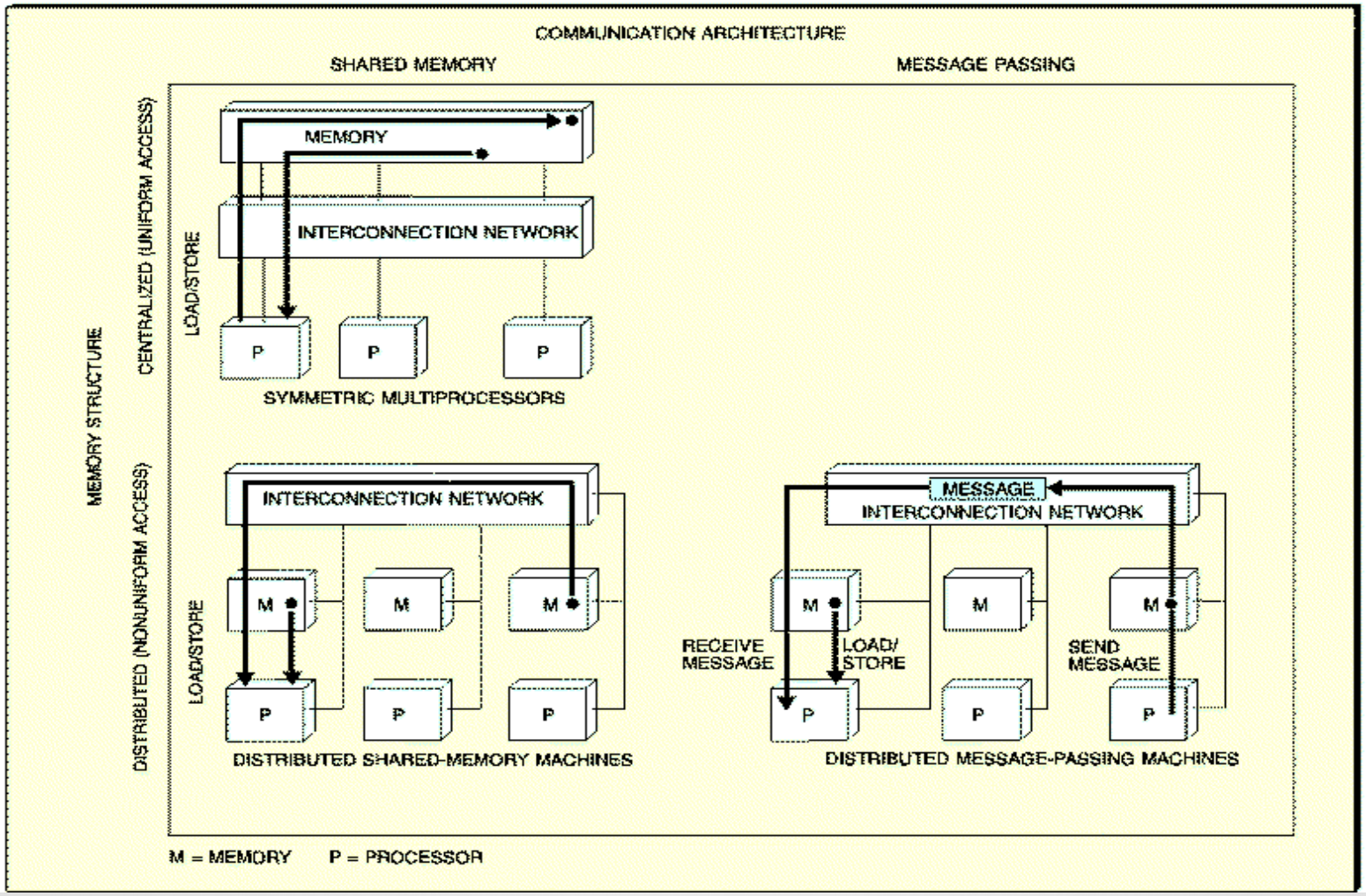
# Paradigmas de Programação Paralela

# Modelos de Programação



# Arquitetura de Computação

**Figure 6 System architecture alternatives for scalable parallel systems**



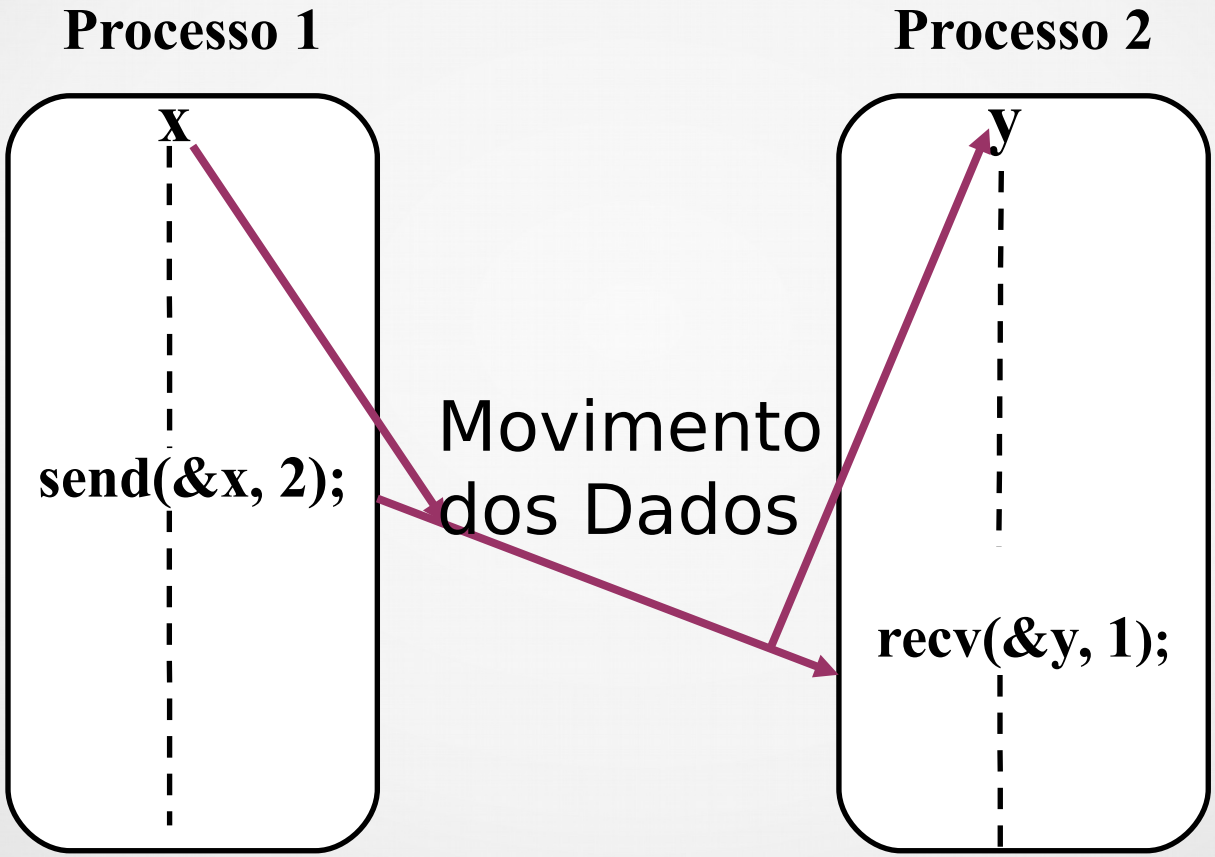
# Memória Compartilhada

- 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.

# 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



# Avaliação de Desempenho

# Medidas de Desempenho

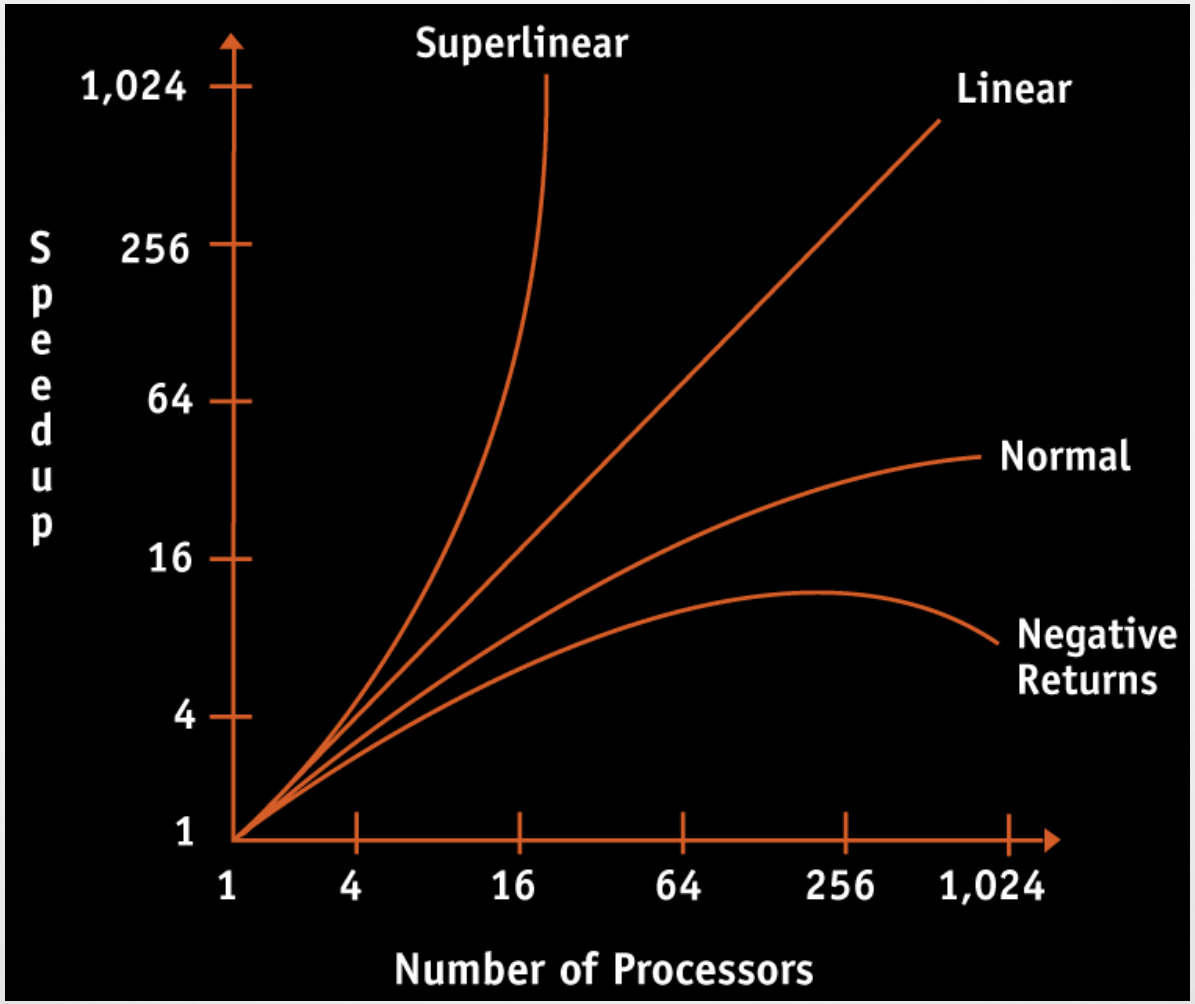
- Speed-up (Aceleração):
  - Mede a razão entre o tempo gasto para execução de um algoritmo ou aplicação em um único processador e o tempo gasto na execução com *n* processadores:

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

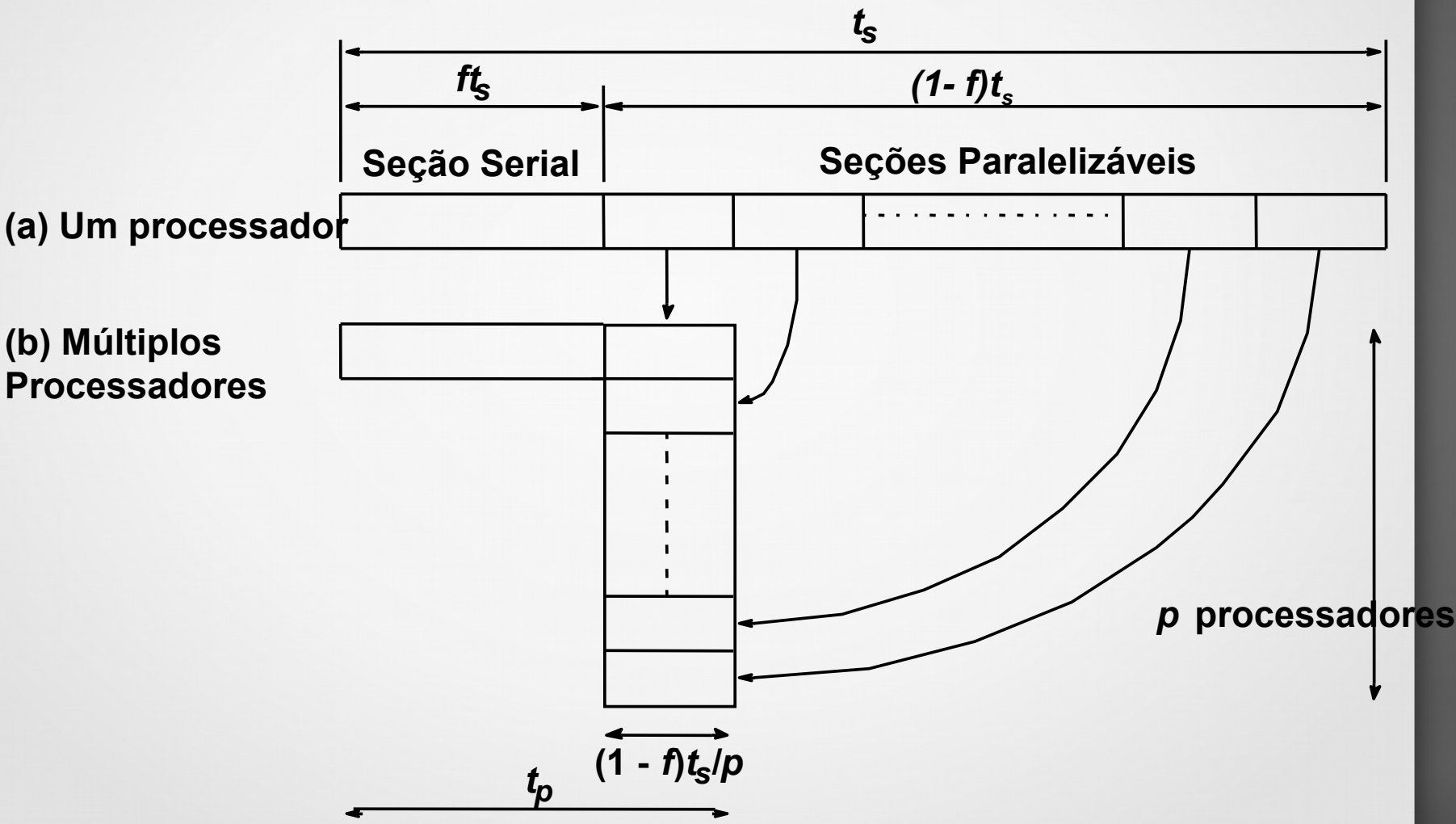
- Eficiência:

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

# Lei de Amdahl



# Lei de Amdahl



# Lei de Amdahl

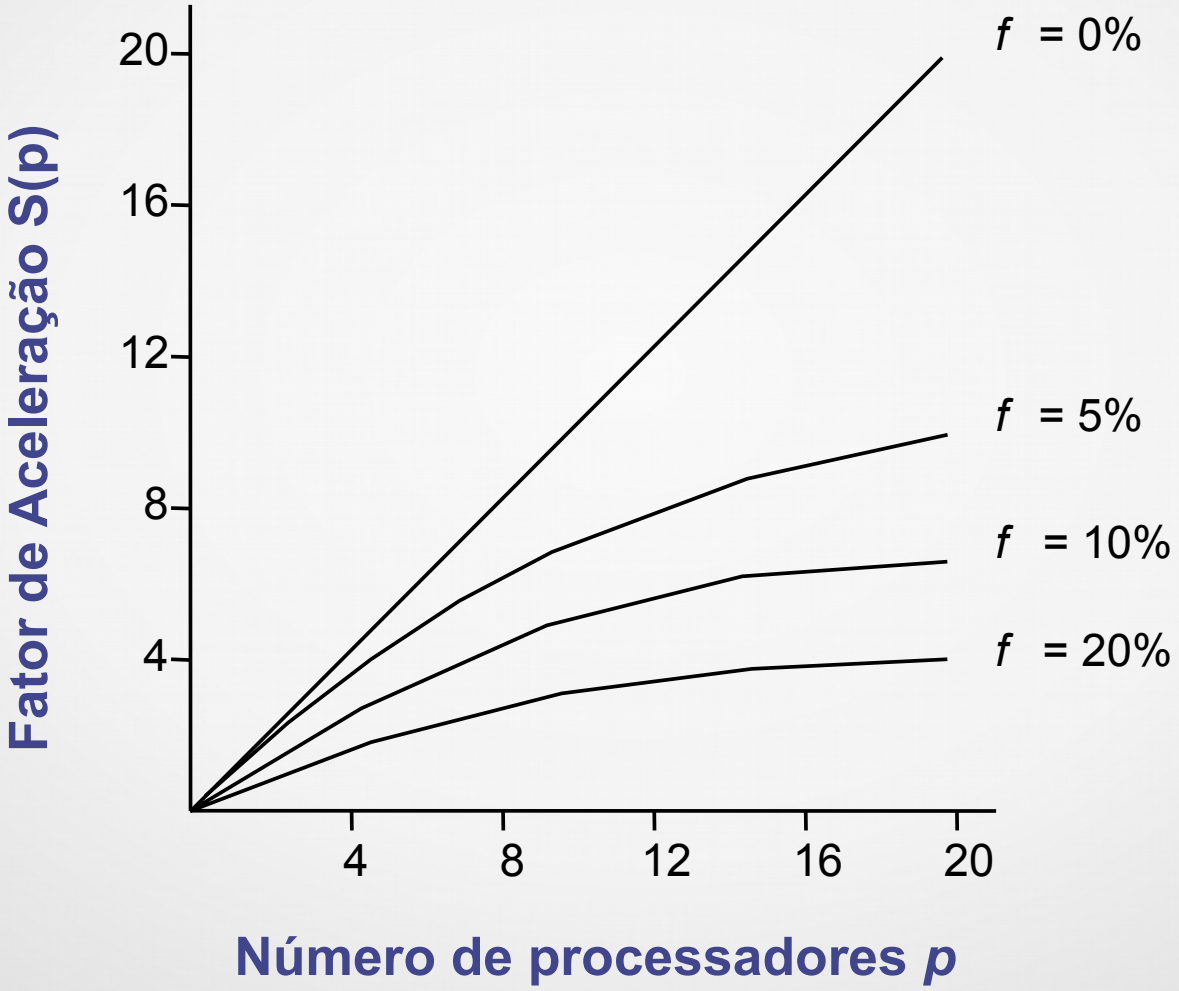
O fator de aceleração é dado por:

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

Esta equação é conhecida como Lei de Amdahl

$$S_t(P) = \frac{1}{f_s + \frac{(1-f_s)}{P}}$$

# Speed-up x Processadores



# Lei de Amdahl

- 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.

$$S_t(P) = \frac{1}{\frac{1}{20}} = \frac{1}{0{,}05} = 20$$

# Escalabilidade

- Um sistema é dito *escalável* quando sua eficiência se mantém constante à medida que o número de processadores (*P*) 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 *P* cresce, de forma a contrabalançar o natural aumento do “overhead” de comunicação quando *P* cresce.

# Lei de Gustafson

$$S_t(P) = P - f_s \cdot (P - 1)$$

$$S_t(P) = f_s + P \cdot (1 - f_s)$$

# Escalabilidade

- 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 S usando P processadores leva um tempo T para ser executado.
- O sistema é dito escalável se um problema de tamanho 2S em 2P processadores leva o mesmo tempo T.
- Escalabilidade é uma propriedade mais desejável que o speed-up.

# Etapas da Criação de um Programa Paralelo

# Decomposição de 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 de Tarefas

- 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.

# Balanceamento de Carga

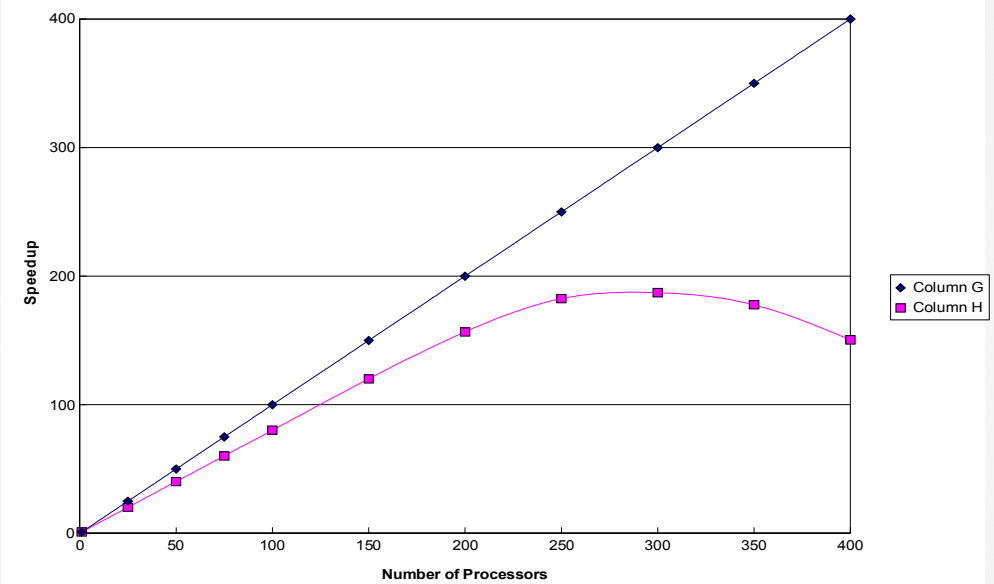
# Balanceamento de Carga

- O Balanceamento de Carga é muito importante para o desempenho das aplicações.
- Assegurar que a parcela de atividade de cada máquina  seja justa pode representar um grande incremento no desempenho.
- O método mais simples de balanceamento de carga é o estático.
- Neste método,  o problema é dividido e tarefas são atribuídas a cada máquina apenas uma vez.

# Balanceamento de Carga

- O objetivo é manter todos os processadores ocupados todo o tempo:
  - Mesmo que seja por motivos econômicos, você vai gostar de manter o uso de cada processador o mais alto possível.
  - A qualquer tempo que você tenha um processador ocioso, você terá uma evidência que sua carga de trabalho não está igualmente distribuída por todas as máquinas.
  - Existe a possibilidade que sua aplicação não esteja rodando tão rápido ou tão eficientemente como poderia.
  - Você poderá então descobrir uma distribuição mais balanceada da carga de trabalho.

# Balanceamento de Carga




- Speed-up sublinear devido a:
  - Gargalos de comunicação
  - Sobrecarga de comunicação
  - Limitações de memória
  - Decomposição de tarefas mal feita.

# Balanceamento Estático

- O particionamento pode acontecer antes do trabalho iniciar, ou como o primeiro passo da aplicação.
- O tamanho e o número de tarefas pode ser ajustado dependendo da capacidade computacional de cada máquina.
- Se a única informação utilizada para alocar o trabalho aos processadores for unicamente a sua dimensão, então o método é dito estático  por natureza, já que ele não muda independente do conteúdo dos dados, mas apenas se sua dimensão muda.
- Em uma rede com baixa carga, este esquema pode ser bastante eficiente.

# Balanceamento Estático

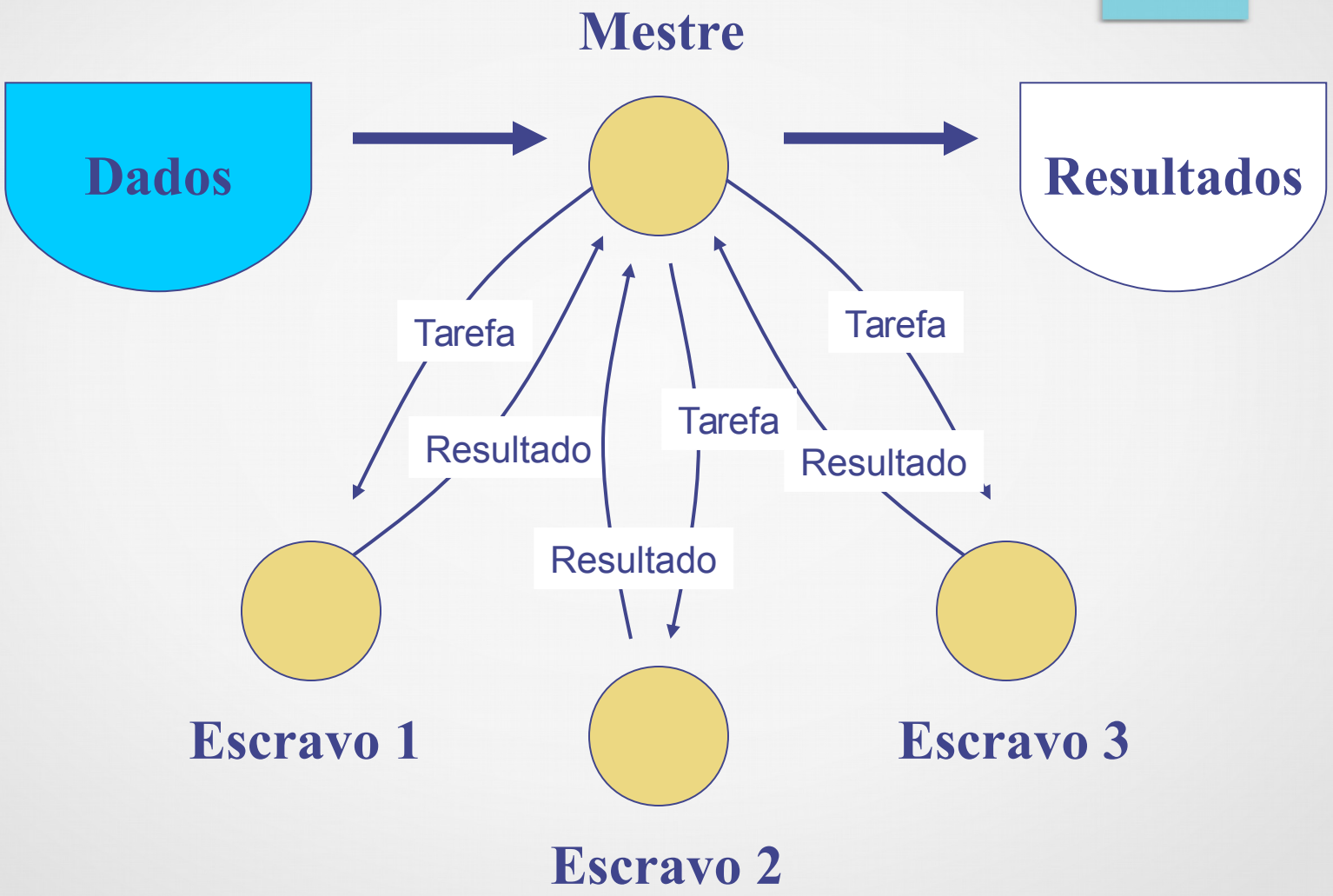
- Decomposição de Matriz:
  - Muitas operações de matrizes são muito facilmente decompostas algoritmicamente, em termos de subseções regulares da matriz original.
  - Por exemplo, o cálculo do determinante usa aspectos da estrutura da matriz bastante longe do conteúdo dos dados que possam ocupar um elemento em particular.
  - A computação pode ser facilmente dividida entre os processadores sem levar em conta os dados propriamente ditos, usando apenas conhecimentos a cerca da dimensão da matriz.
- Particionamento do Conjunto do Índice:
  - De uma forma mais geral, se você tem “n” coisas para computar  e “m” processadores disponíveis para o trabalho, um processo muito simples é dividir “n/m” para cada processador.

# Balanceamento Dinâmico

- Quando a carga computacional varia, um método dinâmico mais sofisticado de balanceamento de carga é necessário.
- O método mais popular é chamado do paradigma do “saco de tarefas”.
- Ele é tipicamente implementado como um programa mestre/escravo, onde o mestre gerencia um conjunto de tarefas.
- Ele envia trabalhos para os escravos assim que eles ficam ociosos.

# Decomposição Mestre-Escravo

# Decomposição Mestre-Escravo



# Balanceamento Dinâmico

- O método mestre-escravo não é adequado para aplicações que requerem  comunicação de tarefa para tarefa, já que as tarefas iniciarão e terminarão em tempos arbitrários.
- Neste caso, um terceiro método pode ser utilizado. Em algum tempo pré-determinado todos os processos param; as cargas de trabalho são re-examinadas  e re-distribuídas se necessário.
- Variações desses métodos são possíveis para aplicações específicas.

# Balanceamento Dinâmico

- Por mais fáceis e intuitivos que os métodos estáticos possam ser, eles apenas focam na estrutura dos dados e não nas características dos dados em si mesmos.
- Por exemplo: se a matriz cujo determinante está sendo calculado for esparsa (ou seja, com maioria dos elementos iguais a 0), um método estático não levará isto em consideração, enquanto que um projeto mais inteligente deve estar apto a reconhecer esta informação como significativa e processar a matriz mais eficientemente.

# Métodos de Grade Adaptativos

- Há uma classe inteira de problemas que podem ser caracterizados pelo uso de “grades” nos dados. Quanto mais fina a grade, mais acurado o resultado, contudo, maior será o tempo de computação necessário.
- Com freqüência você não sabe quão fina deve ser a grade que você necessita até que você inicie os cálculos e descubra, por exemplo, que o erro está muito grande, requerendo uma grade mais fina e cálculos mais precisos.
- Um algoritmo seqüencial apenas recalcularia os pontos da grade e forçaria no sentido de uma grade mais fina.
- Uma aproximação distribuída permitiria o uso de métodos de grade adaptativos, que poderia tratar a grade mais fina com um elemento de dados global, dividido entre os processadores disponíveis, e portanto alcançando um balanceamento de carga muito melhor do aquele que haveria com o processador trabalhando na sua partição inicial de dados e que a seção de refinamento por si mesmo.

# Métodos de Grade Adaptativos

- Há uma classe inteira de problemas que podem ser caracterizados pelo uso de “grades” nos dados. Quanto mais fina a grade, mais acurado o resultado, contudo, maior será o tempo de computação necessário.
- Com freqüência você não sabe quão fina deve ser a grade que você necessita até que você inicie os cálculos e descubra, por exemplo, que o erro está muito grande, requerendo uma grade mais fina e cálculos mais precisos.
- Um algoritmo seqüencial apenas recalcularia os pontos da grade e forçaria no sentido de uma grade mais fina.
- Uma aproximação distribuída permitiria o uso de métodos de grade adaptativos, que poderia tratar a grade mais fina com um elemento de dados global, dividido entre os processadores disponíveis, e portanto alcançando um balanceamento de carga muito melhor do aquele que haveria com o processador trabalhando na sua partição inicial de dados e que a seção de refinamento por si mesmo.

# Simulações N-body

- Uma outra classe de problemas (em realidade há alguma sobreposição entre as duas) lida com o caso onde N partículas unicamente identificadas (mas geralmente chamadas de “bodies”), as quais pode ou não ter interação uma sobre as outras, deve ter alguns cálculos efetuados em cada uma delas.
- Em alguns casos, especialmente naqueles onde não há interação intra-partículas, isto evolui para uma situação de particionamento estático de conjunto de índice.

# Simulações N-body

- Mais freqüentemente, contudo, cada partícula exerce alguma forma de influência sobre seus pares, algumas vezes muito localizadamente (i.e., os efeitos tem curto alcance, como a Força Atômica Forte), algumas vezes não (como a gravidade).
- Nos casos onde há áreas bem definidas de interação, algumas vezes é melhor atribuir todas as partículas que interagem a um mesmo processador, e lidar com todas as iterações restantes (i.e., interações envolvendo partículas fora da área local) como se todas as partículas fossem apenas uma única grande partícula.

# Balanceamento Dinâmico

- Mestre/Trabalhadores
  1. Algumas vezes é possível projetar sua aplicação de modo que o trabalho seja criado em unidades que possam ser colocadas em uma fila e distribuídas para os processadores que estão em outra fila, onde vão sendo adicionados assim que terminem as tarefas em que estejam trabalhando.
  2. Em realidade esta é uma estratégia geral que pode ser aplicada para a situação acima ou sempre que não houver dependências de dados entre as  partes em que você dividir os dados.
  3. Lembre-se , contudo, que este esquema não costuma ser bem escalável para um grande número de processadores.

# Considerações de Desempenho

- O MPI não coloca limites no paradigma de programação que um usuário pode escolher.
- Contudo, há algumas considerações de desempenho que devem ser levadas em conta. A primeira é a granulosidade da tarefa.
- Isto tipicamente se refere à taxa entre os números de bytes recebidos por um processo e o número de operações de ponto flutuante que ele realiza.
- Aumentando a granulosidade irá aumentar a velocidade de execução da aplicação, mas o compromisso é uma redução no paralelismo disponível.

# Considerações de Desempenho

- **O número total de mensagens enviadas também é um fator a ser considerado.**
- **Como uma regra geral, enviando um pequeno número de mensagens grandes leva menos tempo do que enviar um grande número de pequenas mensagens.**
- **Isto nem sempre se aplica, contudo. Algumas aplicações podem sobrepor computação com o envio de pequenas mensagens. O número ideal de mensagens é específico para cada aplicação.**

# Considerações de Desempenho

- Algumas aplicações são bem adequadas ao paralelismo funcional, enquanto que outras se aplicam bem ao paralelismo de dados.
- No paralelismo funcional diferentes máquinas fazem diferentes tarefas baseadas nas suas capacidades específicas.
- Por exemplo, um supercomputador pode resolver parte de um problema adequado ao uso de vetorização, um multiprocessador pode resolver outra parte com uso de paralelização e estações gráficas podem ser utilizadas para visualizar os dados em tempo real.

# Considerações de Desempenho

- No paralelismo de dados, os dados são distribuídos para todas a tarefas na máquina virtual. Operações (freqüentemente bastante similares) são então realizadas em cada conjunto de dados e a informação é transmitida entre os processadores até que o problema seja resolvido.
- Programas em MPI podem também usar um mistura de ambos os modelos para explorar totalmente as potencialidades de cada máquina.
- Computadores diferentes terão capacidades de processamento diferentes.

# Considerações de Desempenho

- Mesmo que todas máquinas sejam do mesmo tipo e modelo, eles podem apresentar diferenças de desempenho devido à carga dos outros usuários.
- Considerações a respeito da rede também são importantes se você estiver utilizando um conjunto de máquinas.
- As latências de rede podem causar problemas e também a capacidade de processamento disponível pode variar dinamicamente dependendo da carga de cada máquina.
- Para combater esses problemas, alguma forma de balanceamento de carga deve ser utilizada.

# Bibliografia


- 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.
- Parallel Programming in OpenMP
  - Chandra, R. et alli; Morgan Kaufman Publishers, 2001
- Parallel Computer Architecture: A Hardware/Software Approach
  - David E. Culler and J. P. Singh – Morgana Kaufman 1999