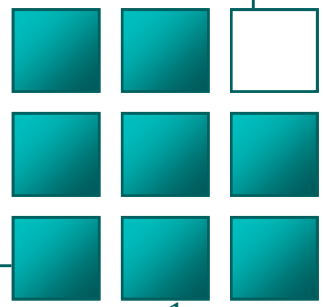
Programa de Pós-graduação em Informática

Tópicos em Sistemas de Computação – Computação em Nuvem

Aletéia Patrícia Favacho de Araújo

Aula 5 – Modelo de Programação*



*Material adaptado de Carnegie Mellon University in Qatar

Lei de Amdahl

- Nós paralelizamos programas a fim de executá-los mais rápido.
- Quanto mais rápido será um programa paralelo ?
  - Suponha que a execução seqüencial de um programa leve T1 unidades de tempo, e a execução paralela em p processadores leve Tp unidades de tempo.
  - Suponha que de todo o programa, uma fração *s* dele não é paralelizável, e que uma fração *1-s* é paralelizável.

Lei de Amdahl

- Então o Speedup (Lei de Amdahl) é:

$$\frac{T_1}{T_p} = \frac{T_1}{(T_1 \times s + T_1 \times \frac{1-s}{p})} = \frac{1}{s + \frac{1-s}{p}}$$

- Ela é usada em computação paralela para prever o ganho máximo (*speedup*) teórico usando múltiplos processadores.

Lei de Amdahl - Exemplo

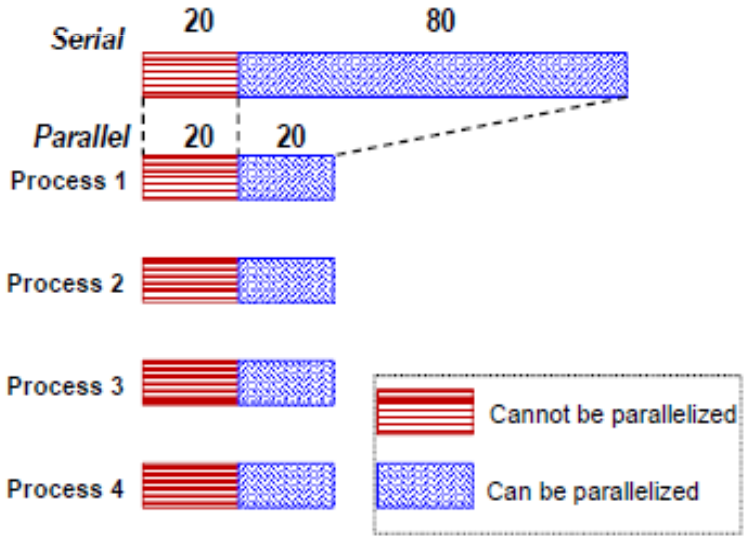
- Suponha que 80% do seu programa possa ser paralelizado e que você pretende usar 4 processadores para executar a sua versão paralela do programa.
- Assim, de acordo com a Lei de Amdahl, o aumento de velocidade possível de ser atingido é:

$$\frac{1}{s+\frac{1-s}{p}} = \frac{1}{0.2+\frac{0.8}{4}} = 2.5 \text{ times}$$

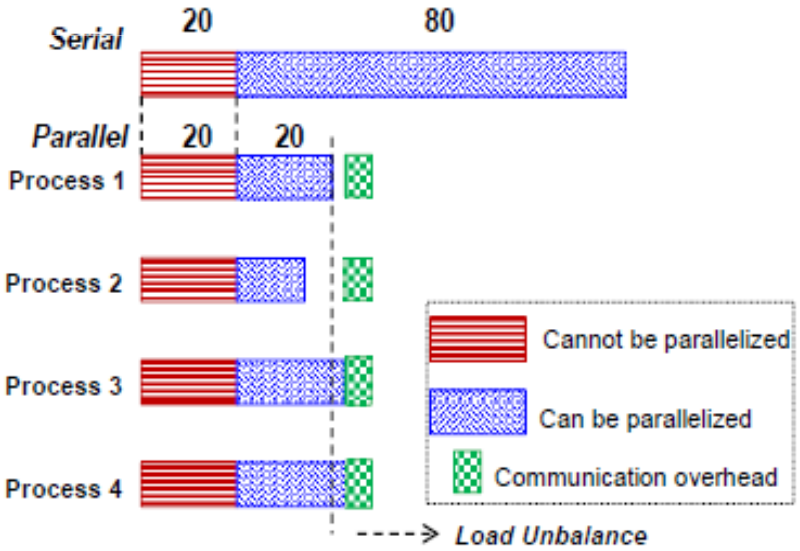
- Embora você use 4 processadores, não é possível obter um aumento de velocidade maior do que 2,5 vezes (ou 40% do tempo de execução).

Casos Reais...

- Na prática, quando executamos um programa paralelo, há uma sobrecarga com comunicação e um desbalanceamento de carga entre os processadores.



1. Parallel Speed-up: An Ideal Case



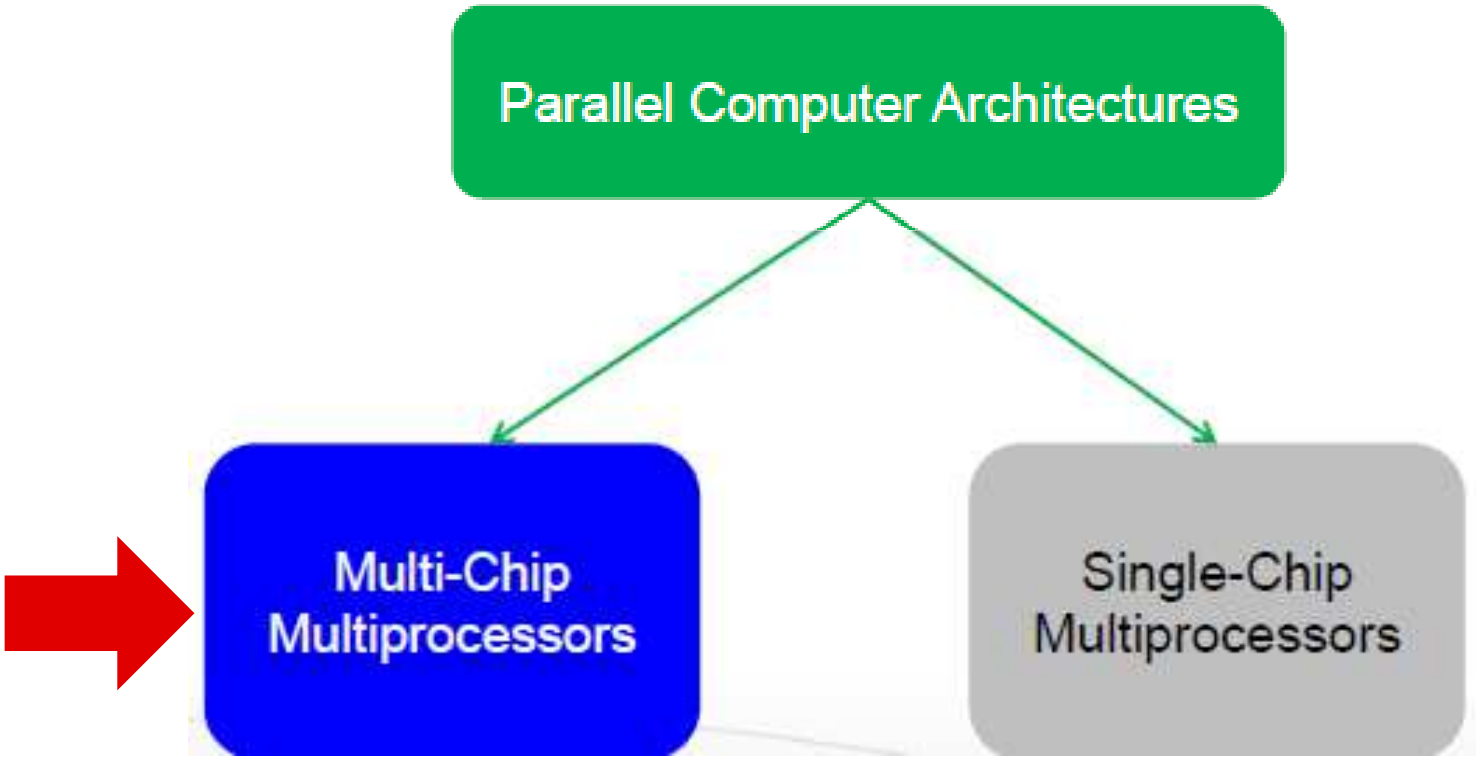
2. Parallel Speed-up: An Actual Case

Diretrizes...

- Logo, para se construir programas paralelos eficientes é importante seguir as seguintes diretrizes:

1. Maximizar a fração do programa que pode ser paralelizado;
2. Equilibrar a carga de trabalho dos processos paralelos;
3. Minimizar o tempo gasto com comunicação.

Arquitetura de Computadores Paralelos

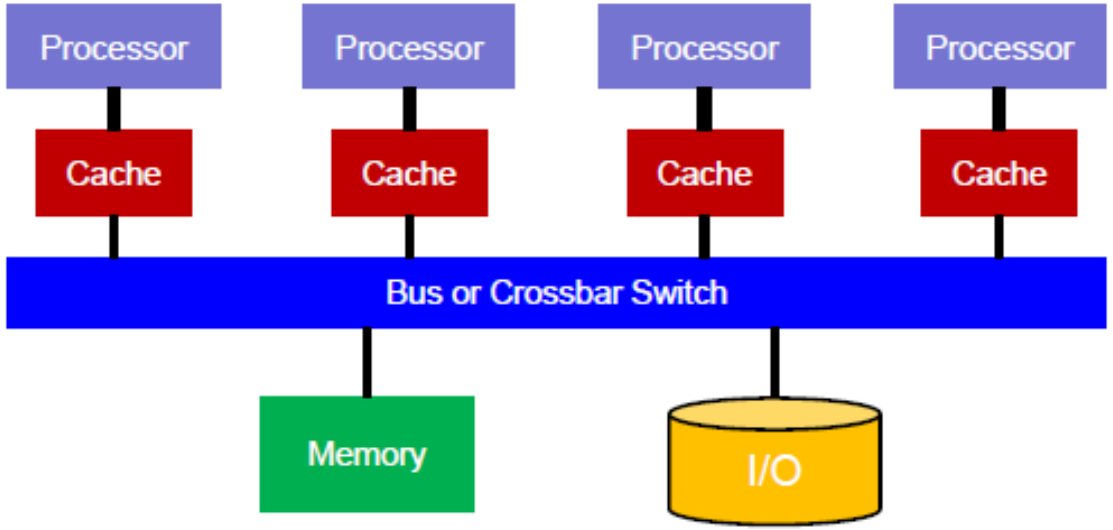


Multiprocessadores Multi-Chip

- É possível categorizar a arquitetura de multiprocessadores *multi-chip* em termos de dois aspectos:
  - Se a memória está fisicamente centralizada ou distribuída; e
  - Se há interesse ou não em compartilhar o espaço de endereçamento.

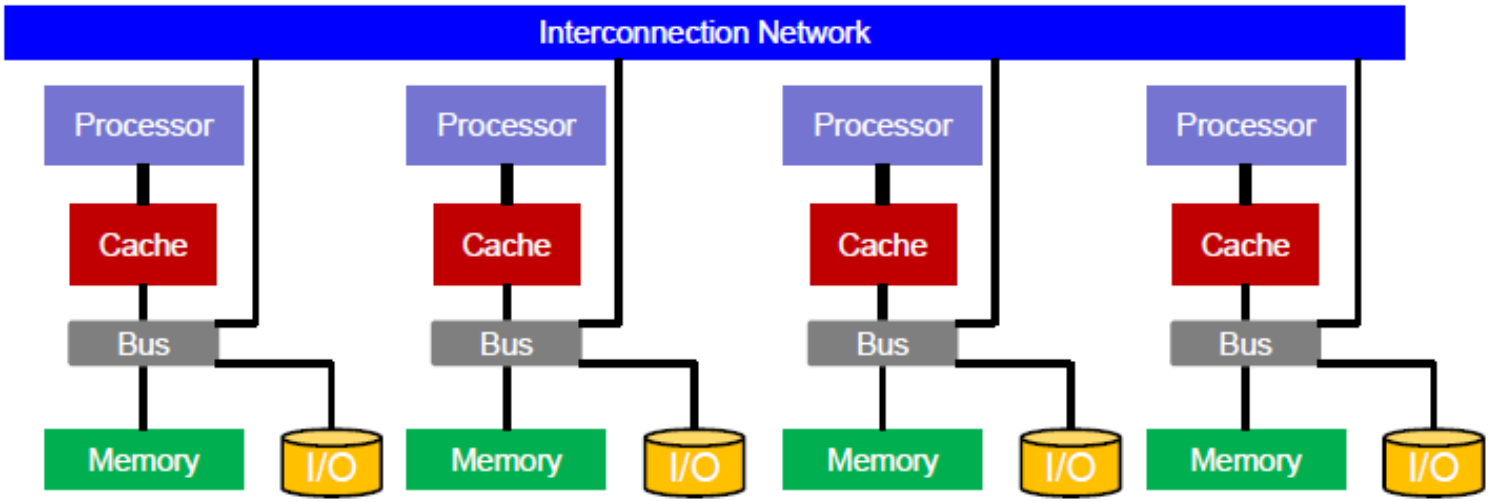
Multiprocessadores Simétricos

- Um sistema com multiprocessadores simétricos (SMP) usa uma arquitetura de memória compartilhada que pode ser acessada igualmente a partir de todos os processadores.



Processadores Maciçamente Paralelos

- Um sistema com uma arquitetura de processadores maciçamente paralelos (MPP) consiste em nós com cada um tendo seu próprio processador, memória e subsistema de E/S.



Arquitetura de Computadores Paralelos

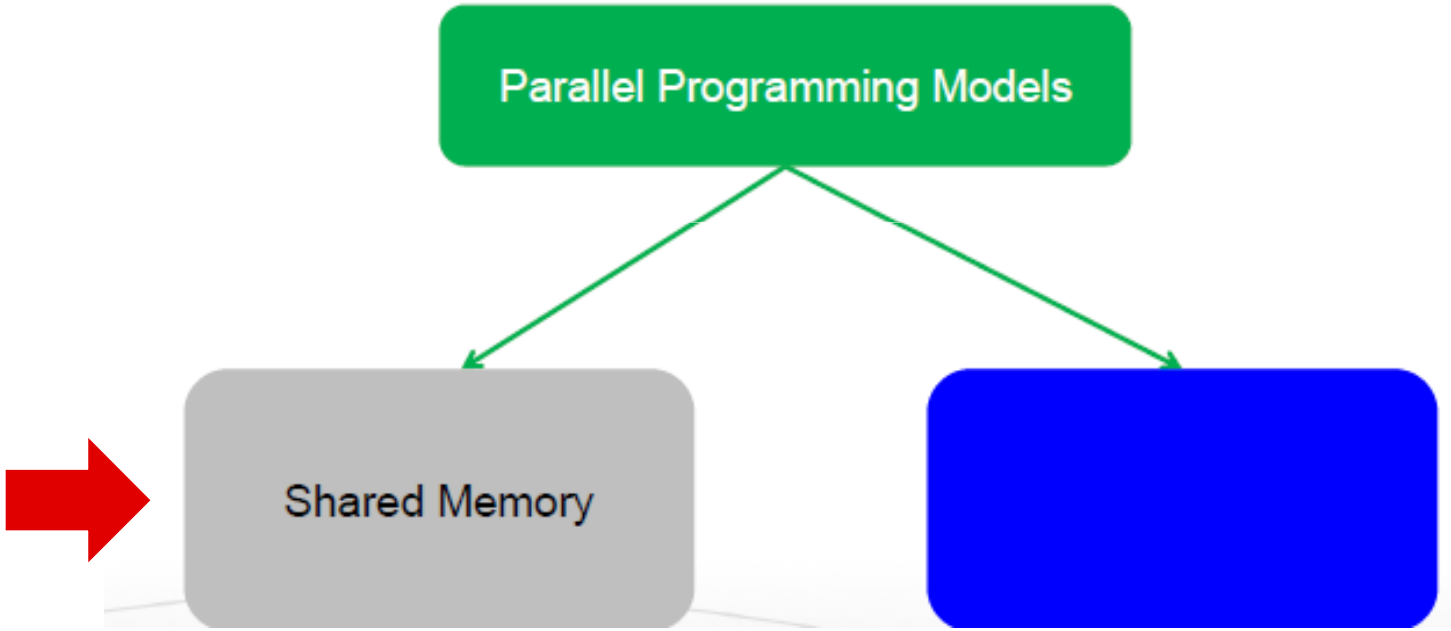
Multiprocessadores com Único-Chip (CMP)

- O CMP é atualmente considerado a arquitetura comumente escolhida;
- Núcleos em um CMP podem ser acoplado tanto de maneira fortemente quanto fracamente:
  - Os núcleos podem ou não compartilhar caches;
  - O método de comunicação pode ser implementado por passagem mensagem ou por memória compartilhada.
- Os CMPs podem ser homogênios (núcleos idênticos) ou heterogênios (núcleos não idênticos).

Modelos de Programação Paralela

- É uma abstração que determina quão facilmente os programadores podem especificar seus algoritmos em unidade paralela de cálculos (ou seja, tarefas) que o hardware entenda;
- Eles existem como uma abstração acima da arquitetura de CPU e de memória, ou seja, eles determinam quão eficientemente tarefas paralelas podem ser executadas no hardware;
- O principal objetivo é utilizar todos os processadores da arquitetura (por exemplo, SMP, MPP, CMP) e minimizar o tempo decorrido do programa.

Modelos Tradicionais de Programação Paralela



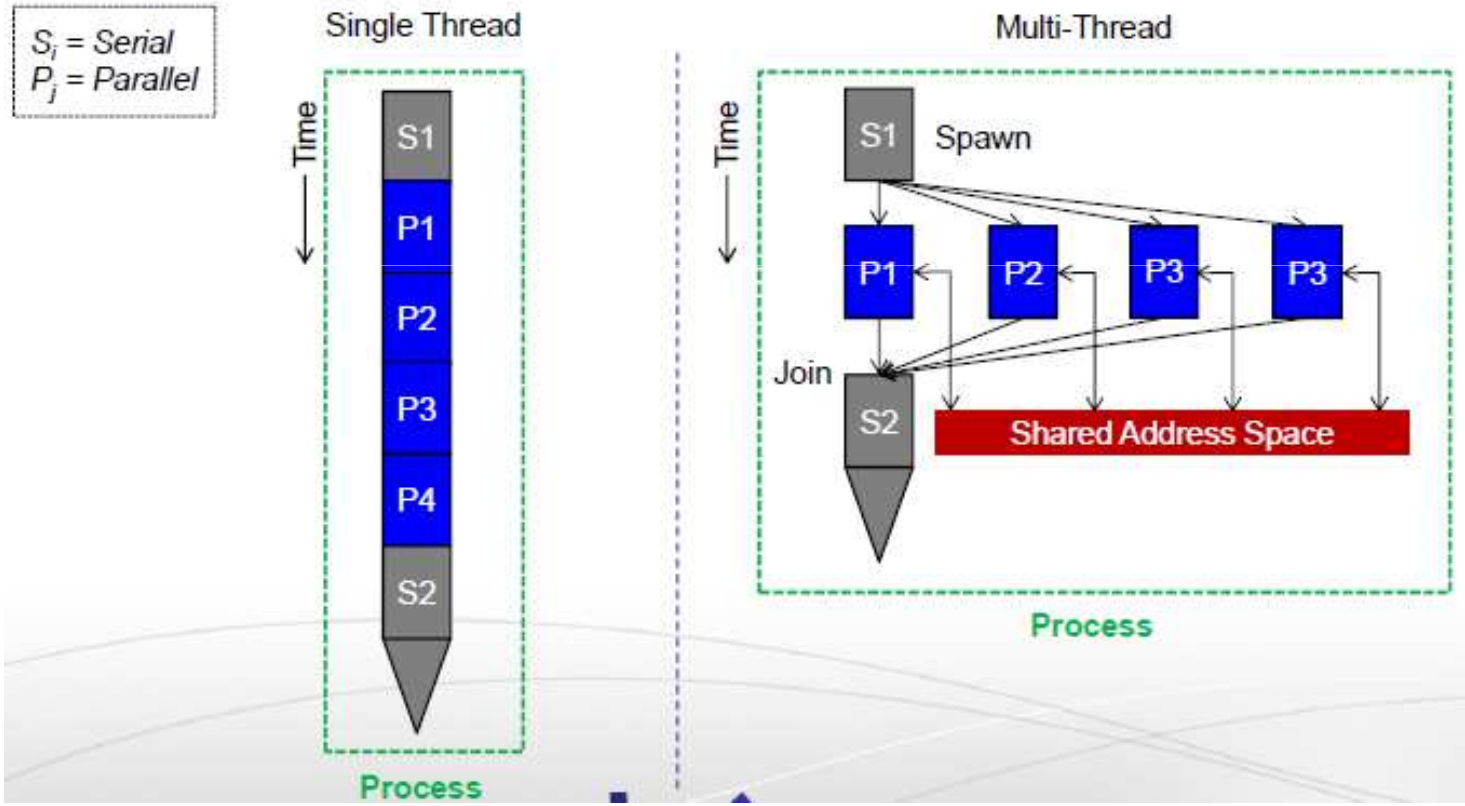
Modelo de Memória Compartilhada

- Nesse modelo, as tarefas compartilham um espaço de endereço comum, onde elas podem ler e escrever simultaneamente;
- Mecanismos como semáforos podem ser utilizados para controlar o acesso à memória compartilhada;
- Isto é semelhante às *threads* de um processo que compartilham um único espaço de endereçamento;
- Programas *multi-threaded* (por exemplo, programas OpenMP) são um bom exemplo do modelo de programação de memória compartilhada.

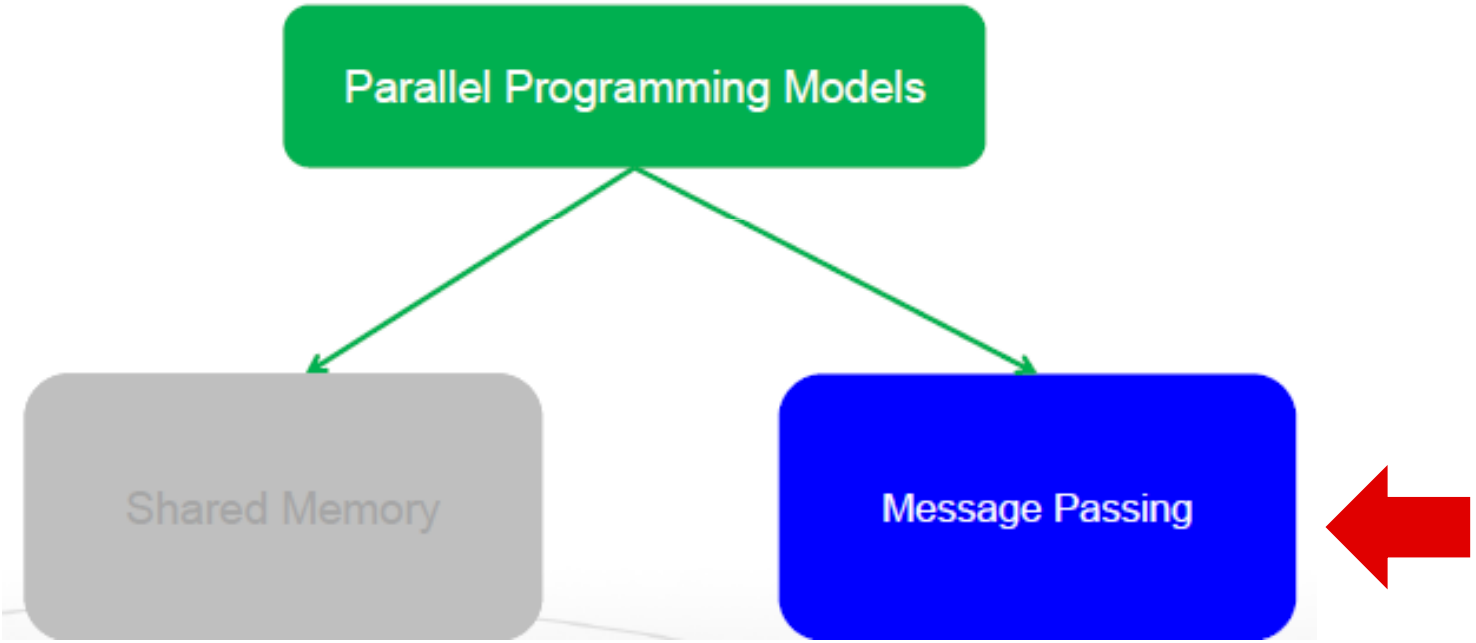
Modelo de Memória Compartilhada

- OpenMP (Open Multi-Processing):
  - É uma interface de programação de aplicativo (API) para a programação multi-processo de memória compartilhada em múltiplas plataformas;
  - Permite acrescentar simultaneidade aos programas escritos em C, C++ e Fortran;
  - Padrão definido e apoiado pelos principais fabricantes de hardware e software do Mundo;
  - Portável e Multiplataforma (Unix e Windows).

Modelo de Memória Compartilhada



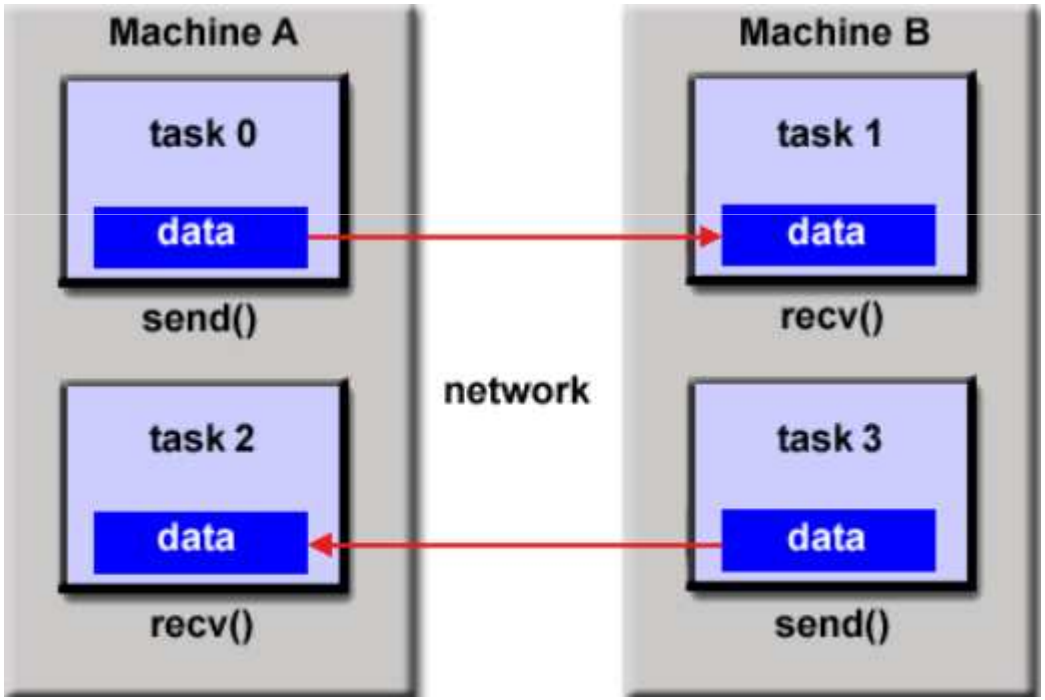
Modelos Tradicionais de Programação Paralela



Modelo de Passagem de Mensagem

- Nesse modelo as tarefas paralelas têm as suas próprias memórias locais, ou seja, uma tarefa não pode acessar a memória de outra tarefa;
- Assim, a comunicação tem que ser feita por meio de mensagens explícitas enviada para o outro processo;
- A transferência de dados requer operações de cooperação a serem executadas por cada processo. Logo, operação de envio deve ter uma operação de recepção correspondente;
- Por exemplo, o modelo MPI (*Message Passing Interface*).

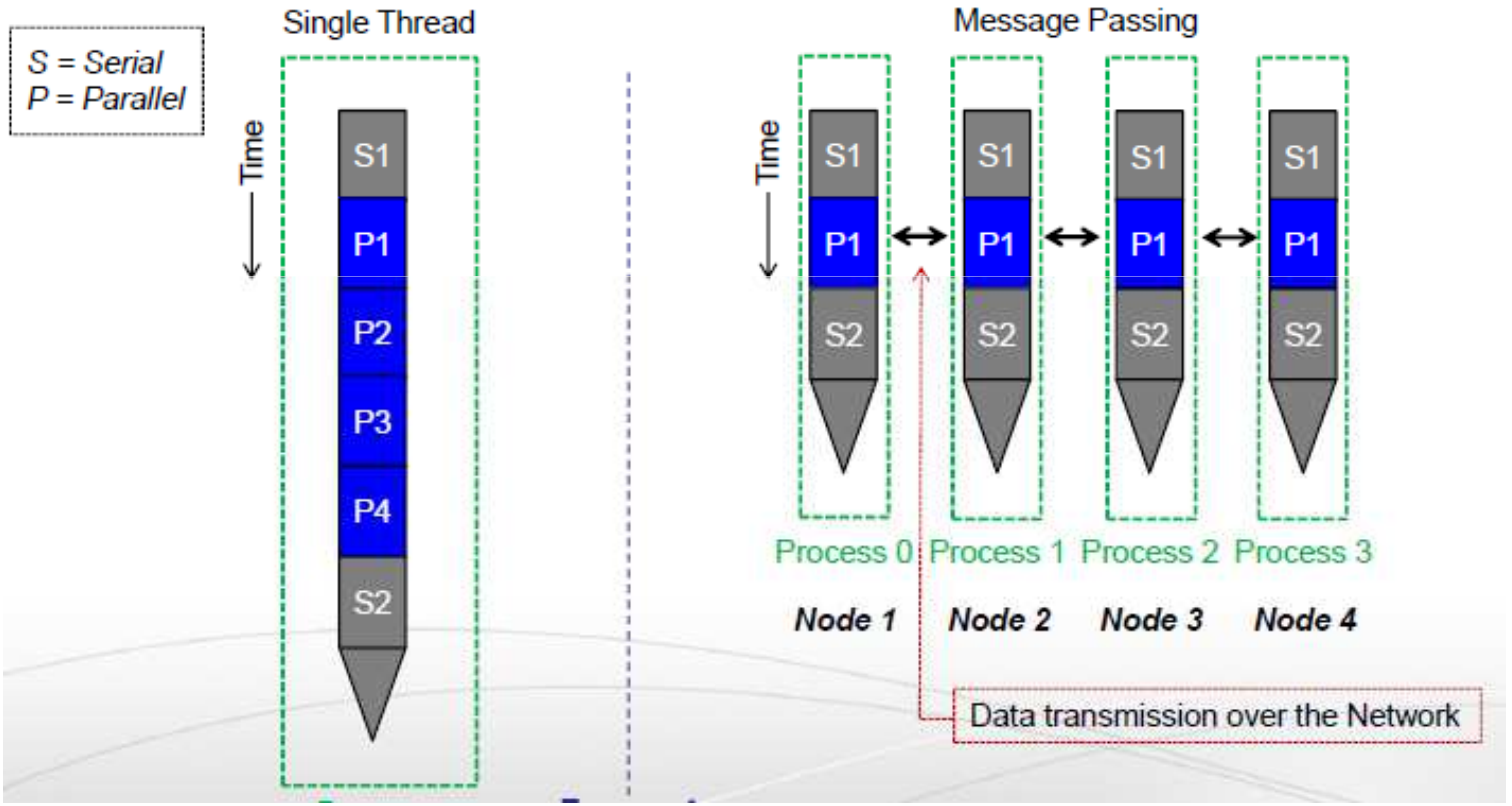
Modelo de Passagem de Mensagem



Modelo de Passagem de Mensagem

- O programador é responsável por determinar todo o paralelismo;
- Várias bibliotecas disponíveis desde a década de 1980
  - Muitas diferenças ➔ Dificuldade de desenvolver aplicações portáteis.
- 1992 ➔ Fórum MPI estabelece um padrão de interface para implementação de passagem de mensagens;
- 1994 ➔ Parte 1
- 1996 ➔ Parte 2 (MPI-2), o MPI é o padrão da indústria para ambientes de passagem de mensagem.

Modelo de Passagem de Mensagem



Comparação entre os Modelos de Programação

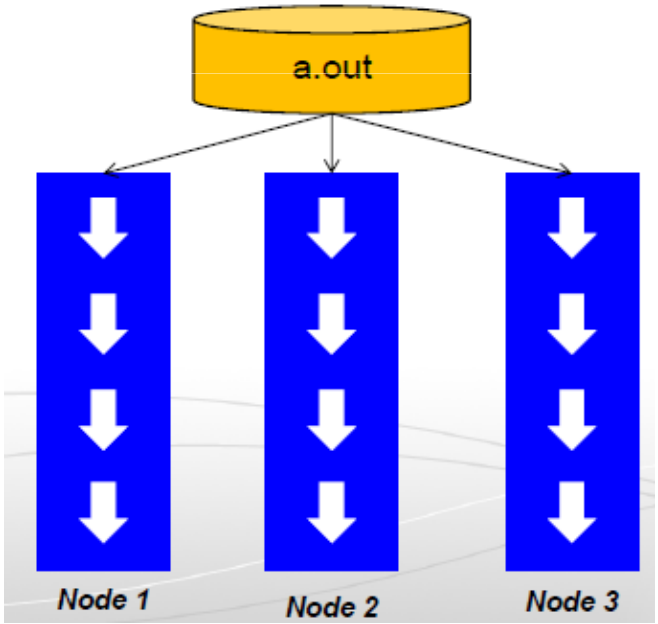
Aspect	Shared Memory	Message Passing
Communication	Implicit (via loads/stores)	Explicit Messages
Synchronization	Explicit	Implicit (Via Messages)
Hardware Support	Typically Required	None
Development Effort	Lower	Higher

Modelo de Passagem de Mensagem

- Quando implementamos por meio de passagem de mensagens, existem maneiras diferentes de como os programas vão cooperar na execução paralela;
- É possível distinguir entre dois modelos:
  - Modelo *Single Program Multiple Data* (**SPMD);**
  - Modelo *Multiple Programs Multiple Data* (**MPMP).**

Single Program Multiple Data – SPMD

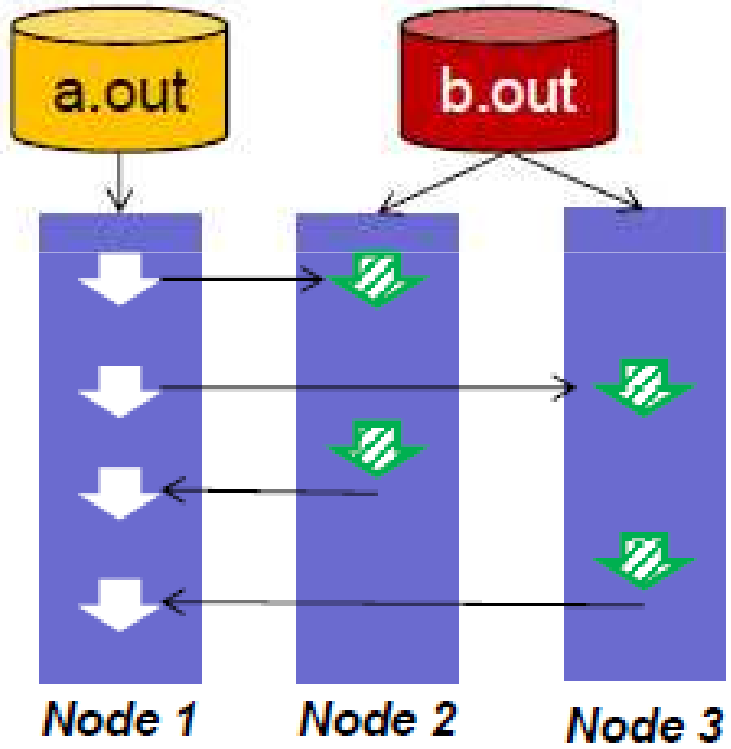
- No modelo SPMD, existe apenas um programa e cada processo executa o mesmo código com diferentes conjuntos de dados.



Multiple Program Multiple Data – MPMD

- O modelo MPMD usa programas diferentes para diferentes processos, mas os processos colaboram para resolver o mesmo problema.
- O estilo MPMD mais comum é o mestre/escravo:
  - O processo mestre executa a coordenação e pode ou não contribuir para a computação;
  - Os escravos executam as tarefas independentemente. E só se comunicam com o mestre para passar o resultado da tarefa por ele executada, e obter a sua próxima tarefa.

Estilo de Programação Mestre-Escravo



Modelo *MapReduce*

- MapReduce é um modelo de programação para processamento massivo de dados;
- O poder do MapReduce reside na sua capacidade de se adaptar facilmente ao aumento de escala, e sua habilidade de trabalhar com milhares de computadores, cada um com vários núcleos de processamento;
- Dados Web escalam da ordem de GBs para TBs ou PBS. Assim, é provável que o conjunto de dados de entrada não caiba em um único disco rígido de um só computador. Logo, um sistema de arquivos distribuídos (por exemplo, *Google arquivo System* -GFS) é necessário.

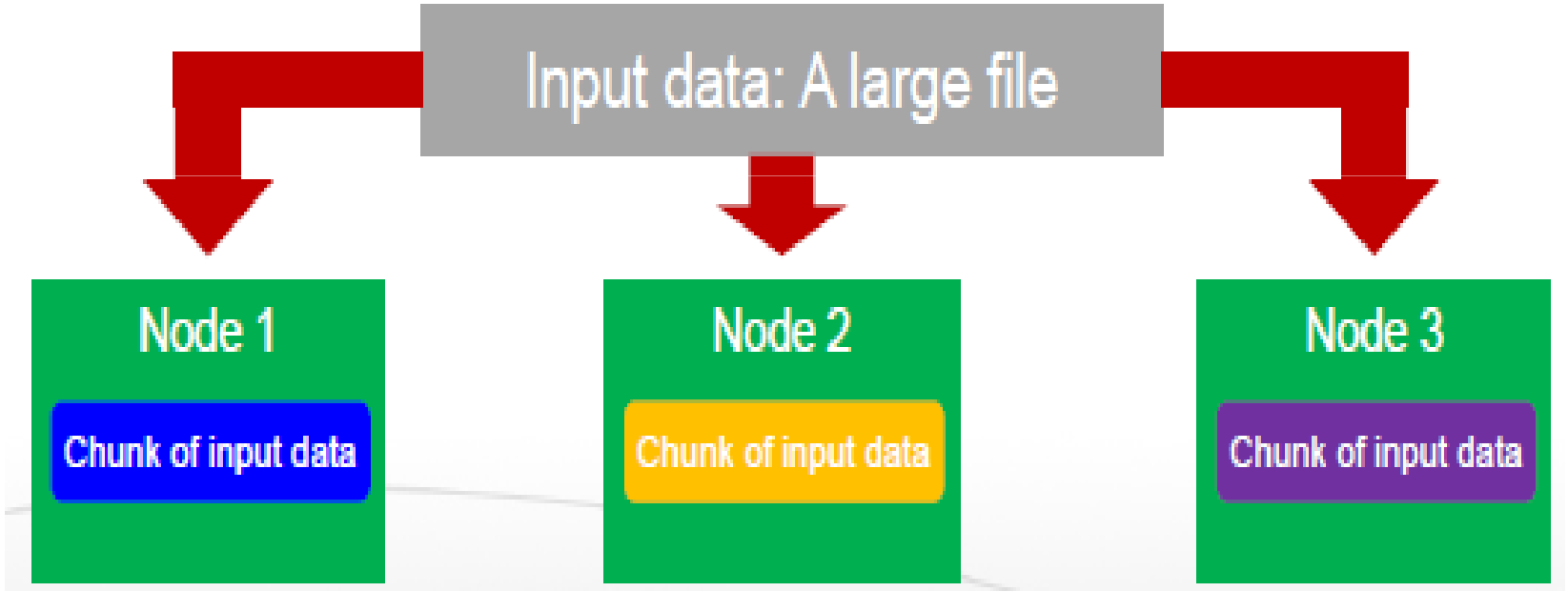
Modelo *MapReduce*

- MapReduce divide a carga de trabalho em várias tarefas independentes e as distribui nas máquinas do *cluster*;
- O trabalho realizado por cada tarefa é feita de forma isolada um do outro;
- A quantidade de comunicação que pode ser realizada por tarefas é limitada, principalmente, por razões de tolerância a falhas e de escalabilidade.
- Ele trata de tarefas como o particionamento dos dados, o agendamento de tarefas e a comunicação entre os nós do *cluster*.

Modelo *MapReduce*

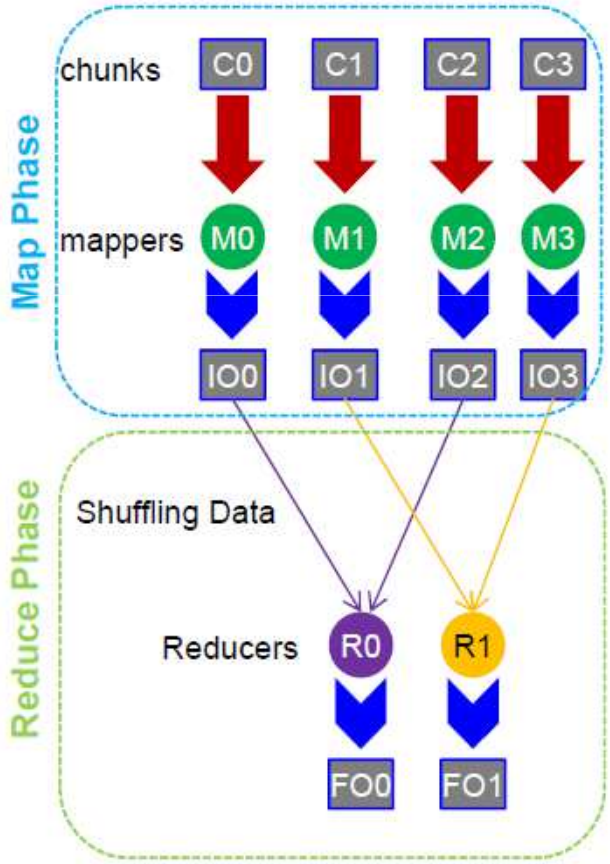
- Em um *cluster* MapReduce, os dados são distribuídos para todos os nós do *cluster* automaticamente;
- Um sistema de arquivos distribuído (por exemplo, GFS ou HDFS) divide grandes arquivos de dados em pedaços (*chunks*) que são processados por diferentes nós do *cluster*;
- Assim, mesmo que os pedaços de arquivos sejam distribuídos em várias máquinas, eles formam um único espaço de nomes (*namesapce*).

Modelo *MapReduce*



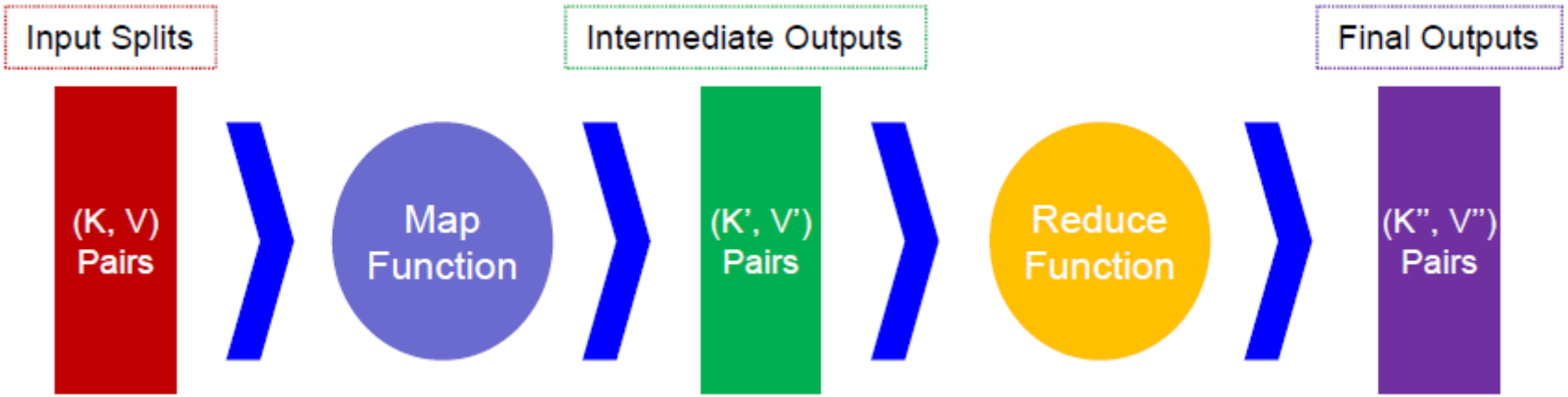
Modelo *MapReduce*

- O MapReduce quebra o fluxo de dados em duas fases:
  - Fase Map
  - Fase Reduce
- O programador tem que especificar duas funções:
  - *map(chave, valor) -> (chaveInt, valorInt)*
  - *reduce(chaveInt, valoresInt) -> (saídas)*



Modelo *MapReduce*

- No modelo Map Reduce os elementos de dados são sempre estruturados como o par chave-valor (K, V);
- As funções *map* e *reduce* sempre recebem e emitem um par *(K, V).*



Função
Map(chave, valor)

- A função *Map(chave, valor)* recebe como parâmetro de entrada um par *(chave, valor)* e emite na saída um par *(chave intermediária, valor intermediário);*
- A função *Map* recebe uma lista como entrada, e aplicando uma função dada, gera uma nova lista como saída;
- Um exemplo simples é aplicar um fator multiplicador a uma lista, por exemplo, dobrando o valor de cada elemento:

```
map({1,2,3,4}, (x2)) > {2,4,6,8}
```

Função
Reduce(chaveInt, valoresint)

- A função *Reduce* recebe o conjunto de valores intermediários *valoresInt* (saídas da função *map*) que estão associados à mesma chave intermediária *chaveInt* e emite como saída os valores finais na saída para a mesma chave;
- A função *Reduce* recebe como entrada uma lista e, em geral, aplica uma função para que a entrada seja reduzida a um único valor na saída;
- Algumas funções do tipo Reduce mais comuns seriam “mínimo”, “máximo” e “média”, por exemplo.

Função
Reduce(chaveInt, valoresint)

- Aplicando essas funções ao exemplo anterior tem-se as seguintes saídas:

```
reduce({2,4,6,8}, mínimo) > 2
reduce({2,4,6,8}, máximo) > 8
reduce({2,4,6,8}, média) > 5
```

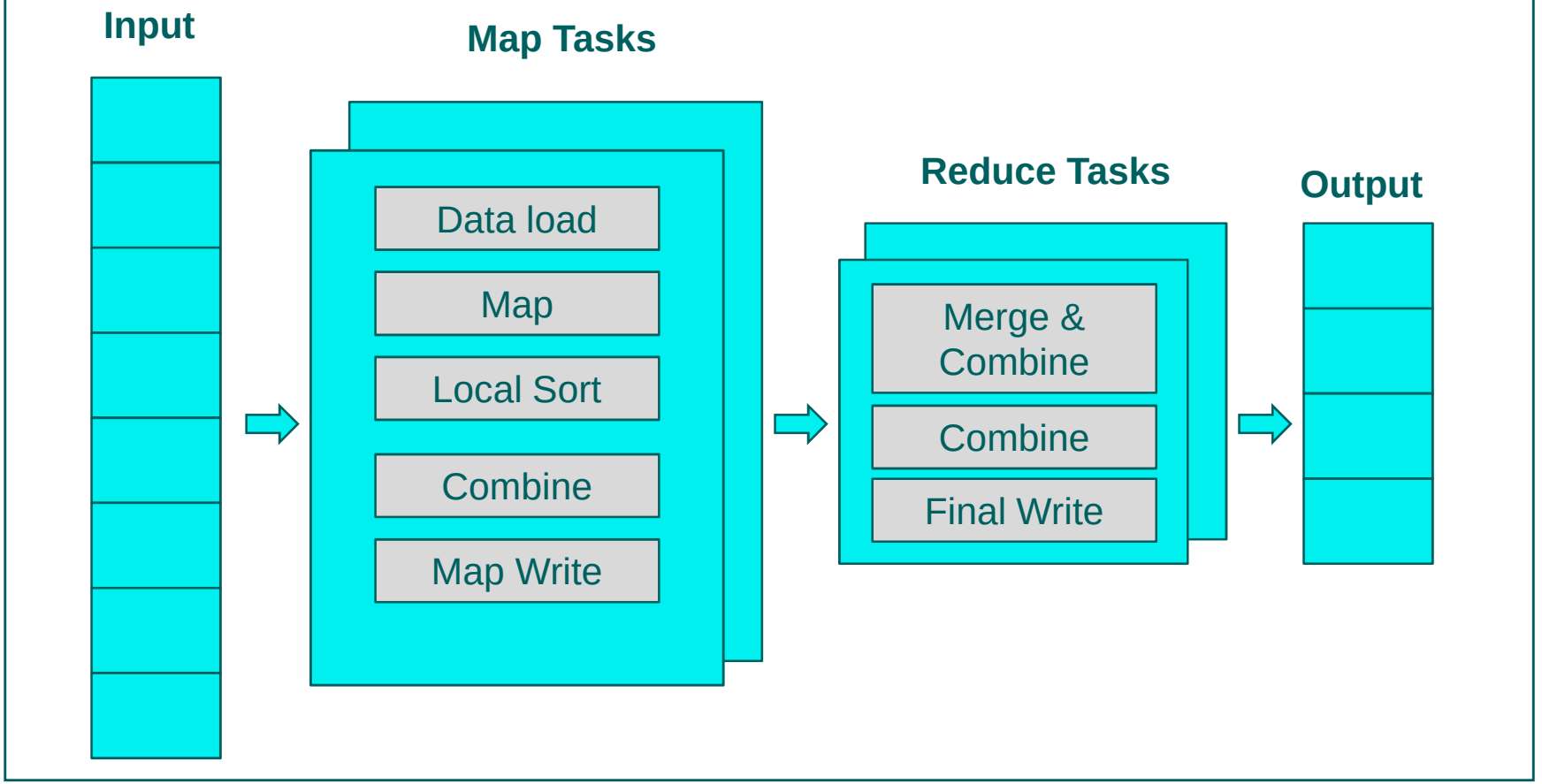
Passo a Passo

- Na fase Map, os dados são lidos a partir de um sistema de arquivos distribuídos, particionado e enviados para um conjunto de máquinas do *cluster*;
- A tarefa Map processa as entradas independentemente uma da outra, e produz resultados intermediários;
- Os resultados intermediários são armazenados no disco local de cada nó.

Passo a Passo

- Quando todas as tarefas Map forem completadas, a fase Reduce começa;
- O modelo de programação MapReduce move o processamento para onde os dados residem, diminuindo a transmissão e melhorando a eficiência;
- Tanto o *input* quanto o *output* são armazenados em um sistema de arquivo distribuídos.

Passo a Passo



Exemplo Didático Clássico (Contar Palavras – WordCount)

- Tem como entrada um conjunto de arquivos texto a partir dos quais a frequência das palavras será contada.
- Como saída, será gerado um arquivo texto contendo cada palavra e a quantidade de vezes que foi encontrada nos arquivos de entrada.
- Vamos usar dois arquivos texto distintos:
  - O arquivo "**entrada1.txt"** contendo as frases: "CSBC JAI 2012" e "CSBC 2012 em Curitiba".
  - O arquivo **"entrada2.txt"** contendo as frases: "Minicurso Hadoop JAI 2012", "CSBC 2012 Curitiba Paraná".

Exemplo Didático Clássico (Contar Palavras – WordCount)

- Na fase de mapeamento, cada arquivos texto será analisado individualmente em uma função *Map* e para cada palavra encontrada será gerada uma tupla contendo o par - palavra encontrada, valor 1.
- Ao final da fase de mapeamento, a quantidade de pares chave/valor gerados será igual ao número de palavras existentes nos arquivos de entrada.

Arquivo entrada1.txt:	Arquivo entrada2.txt:
(CSBC, 1)	(Minicurso, 1)
(JAI, 1)	(Hadoop, 1)
(2012, 1)	(JAI, 1)
(CSBC, 1)	(2012, 1)
(2012, 1)	(CSBC, 1)
(em, 1)	(2012, 1)
(Curitiba, 1)	(Curitiba, 1)
	(Paraná, 1)

Exemplo Didático Clássico (Contar Palavras – WordCount)

- Na implementação do Hadoop MapReduce, há uma fase intermediária – Fase de Shuffle.
- Assim, ao final da execução da fase map, a fase intermediária é executada com a função de agrupar os valores de chaves iguais em uma lista, produzindo a saída:

Arquivo entrada1.txt:	Arquivo entrada2.txt:
(CSBC, 1)	(Minicurso, 1)
(JAI, 1)	(Hadoop, 1)
(2012, 1)	(JAI, 1)
(CSBC, 1)	(2012, 1)
(2012, 1)	(CSBC, 1)
(em, 1)	(2012, 1)
(Curitiba, 1)	(Curitiba, 1)
	(Paraná, 1)

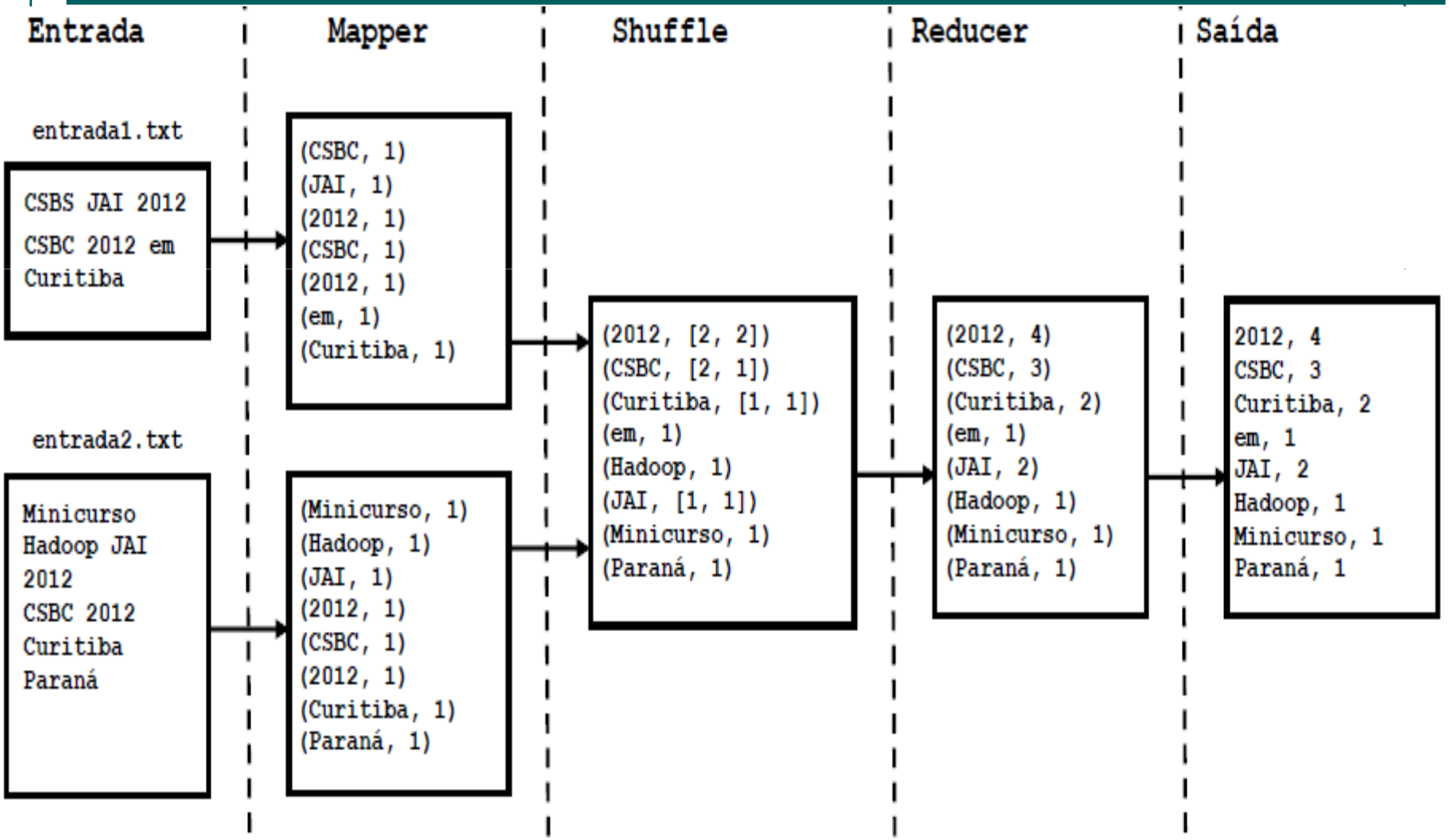
```
(2012, [2, 2])
(CSBC, [2, 1])
(Curitiba, [1, 1])
(em, 1)
(Hadoop, 1)
(JAI, [1, 1])
(Minicurso, 1)
(Paraná, 1)
```

Exemplo Didático Clássico (Contar Palavras – WordCount)

- Em seguida, os dados serão disponibilizados para a fase reduce.
- Serão executadas tarefas *Reduce* com o intuito de reduzir valores de chaves iguais provenientes de diferentes execuções de tarefas *Map.*
- Após a execução das tarefas *Reduce, a* saída gerada é:

```
(2012, 4)
(CSBC, 3)
(Curitiba, 2)
(em, 1)
(JAI, 2)
(Hadoop, 1)
(Minicurso, 1)
(Paraná, 1)
```

Fluxo Lógico de Execução da Aplicação



Fontes Bibliográficas

[1] DEAN, Jeffrey; GHEMAWAT, Sanjay. MapReduce: Simplied Data Processing on Large Clusters. *Communications of the ACM*, v. 51, n. 1, p. 107-113, jan. 2008.

[2] LÄMMEL, Ralf. Google's MapReduce Programming Model - Revisited. *Science of Computer Programming*, v.70, n. 1, p. 1-30, jan. 2008.

[3] Heshan Lin; ARCHULETA, Jeremy; Xiaosong Ma; Wu-chun Feng; Zhe Zhang; GARDNER, Mark. MOON: MapReduce On Opportunistic eNvironments. *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing - ACM HPDC '10*, p. 95-106, 2010.