INSTITUTO FEDERAL
Rio Grande do Norte
Campus Pau dos Ferros

EXPOTEC
EXPOSIÇÃO, CIENTÍFICA, TECNOLÓGICA E CULTURAL CAMPUS PAU DOS FERROS 2015/2016
GESTÃO DE RECURSOS HÍDRICOS: 100 ANOS DE ABANDONO

Paralelismo em Computadores com Tecnologia *Multicore*

Prof. Me. Demétrios Coutinho

IFRN - Pau dos Ferros

Pau dos Ferros/RN, 25 de fevereiro de 2016

Descrição: Para se utilizar os vários núcleos de processamento disponíveis nos computadores atuais de forma eficiente, faz necessário utilizar técnicas de programação paralela. Neste curso, pretende-se apresentar conceitos fundamentais e uma visão prática de programação paralela, utilizando-se OpenMP para computadores com múltiplos núcleos.
Conteúdo:

- Introdução
- OpenMP
- Análise de desempenho de sistemas paralelos

Pré-requisitos: C++ e básico de Sistema Operacionais.
Recursos: Laboratório de Informática com 30 computadores com processadores com 4 núcleos ou acima. Sistema operacional Linux. Compiladores c/c++. Projetor e quadro branco.
Carga-horária: 8 horas.

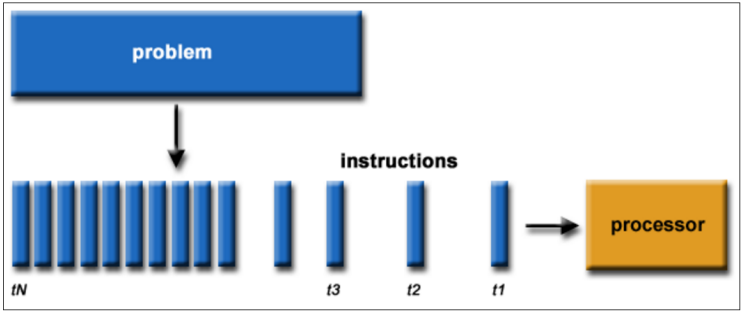
Sumário

Introdução
O que é computação paralela?

- Tradicionalmente, software são escritos para serem seriais:
  - Rodar em um único computador tendo apenas uma CPU;
  - Instruções são escritas uma depois da outra;
  - Apenas uma instrução pode ser executada em um determinado instante.

Introdução
O que é computação paralela?

Figura: Problema Serial.



Introdução
O que é computação paralela?

- Computação paralela é a forma em quem múltiplos cálculos são realizados de forma paralela;
- De forma simples, usa-se múltiplos recursos computacionais de forma simultânea para resolver um problema:
  - Rodando em múltiplas CPUs;
  - O problema é quebrado de forma que possam ser resolvidos de forma paralela;
  - Cada parte é então quebrada em séries de instruções;
  - Essas instruções de cada parte são executadas simultaneamente em diferentes CPUs.

Introdução
O que é computação paralela?

Figura: Problema Paralelo.

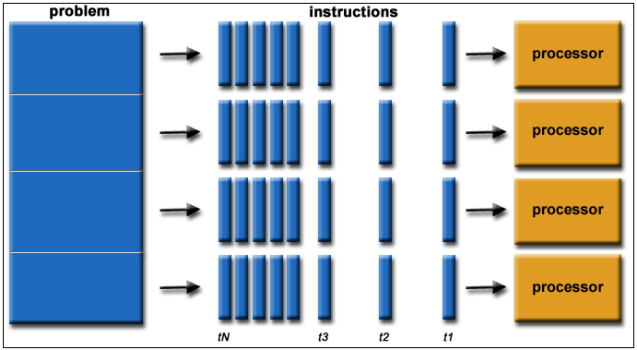
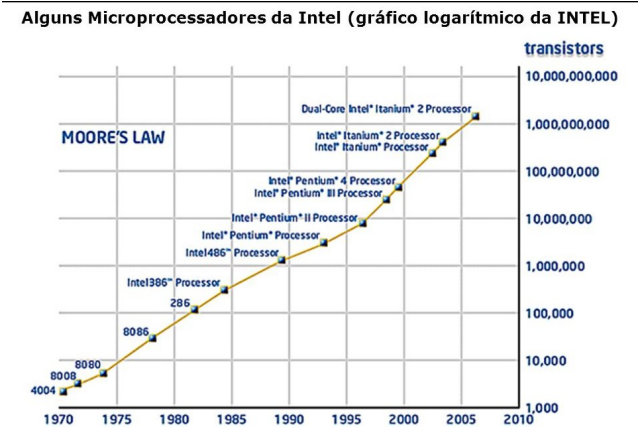
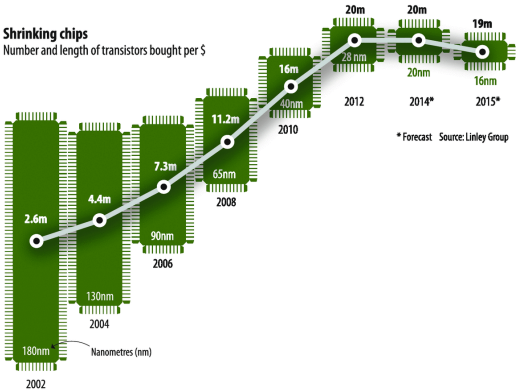


FIGURA: Lei de Moore criada em 1965.



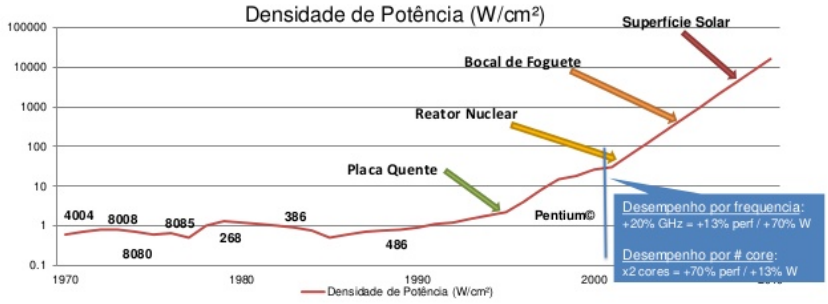
- A lei de Moore dizia que a cada 1 ano e meio a quantidade de transistores dobraria.

FIGURA: Diminuição do tamanho dos chips.

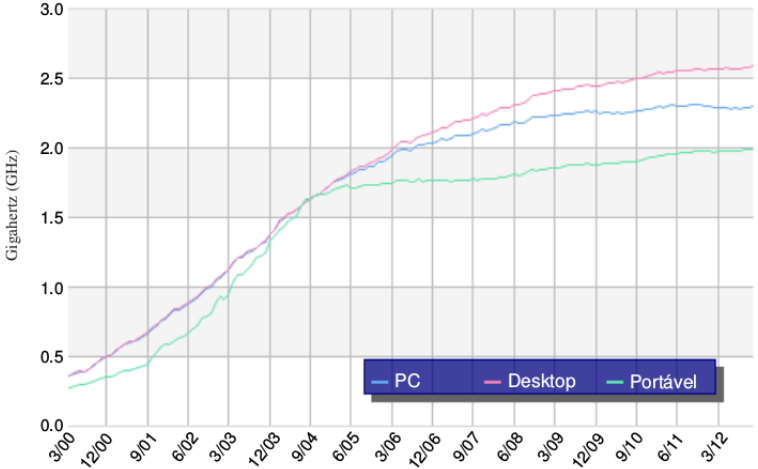


- Assumia-se que quanto mais transistores em um chip maior desempenho era adquirido.

Térmica
Perspectiva ~~Histórica~~ da Lei de Moore



Média de velocidade de operação dos processadores de 2000 até 2012.



Fonte: http://techtalk.pcpitstop.com/research-charts-cpu/

Introdução
Multiprocessamento

- Explorar diferentes tipos de paralelismo;
- Reduzir a potência.

Introdução
Multiprocessamento

- Explorar diferentes tipos de paralelismo;
- Reduzir a potência.(Consequentemente, reduzir calor);
- *Cores* mais simples são mais fáceis de projetar e menor custo.

Onde está o paralelismo?

Aplicações envolvendo grande capacidade de processamento

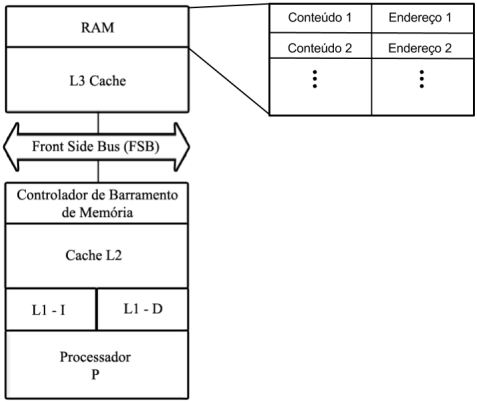
- Meteorologia.
- Simulação de fenômenos físicos.
- Modelagem de sistemas.
- Algebra Linear.
- Otimização de Sistemas.
- Processamento de Imagens.
- Computação gráfica.

It's not a piece of cake!

- Programas de computador paralelos são mais difíceis de programar que sequenciais!
- Pois, introduz diversas novas classes de defeitos potenciais que podem ocorrer.
- Sincronização x Computação

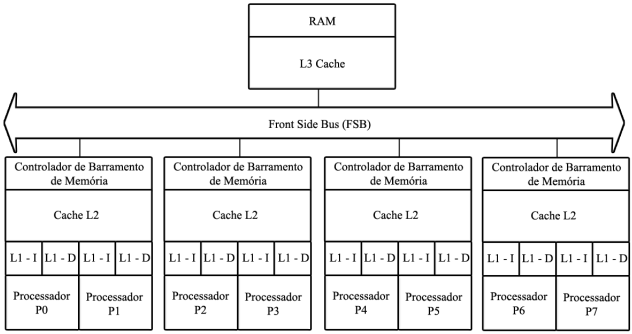
Arquitetura

Arquitetura de um processador com uma unidade de processamento



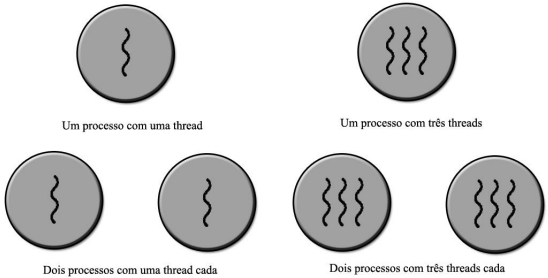
Arquitetura

Arquitetura de um processador *multicore*



Unidades de Paralelismo

- Um processo é basicamente um programa em execução.
- Threads é uma forma de um processo dividir a si mesmo em duas ou mais tarefas que podem ser executadas concorrentemente.



Dependência

- Entender a dependência de dados é fundamental na implementação de algoritmos paralelos.

```
1:Função dep( a , b )
2:c ← a · b
3:d ← 2 · c
4:FimFunção
```

Dependência

- Entender a dependência de dados é fundamental na implementação de algoritmos paralelos.

```
1:Função dep( a , b )
2:c ← a · b
3:d ← 2 · c
4:FimFunção
```

- Operação 3 depende da operação 2!

Dependência

- Entender a dependência de dados é fundamental na implementação de algoritmos paralelos.

```
1:Função dep( a , b )
2:c ← a · b
3:d ← 2 · c
4:FimFunção
```

- Operação 3 depende da operação 2!

```
1:Função dep( a , b )
2:c ← a · b
3:d ← 2 · b
4:e ← a + b
4:FimFunção
```

Dependência

- Entender a dependência de dados é fundamental na implementação de algoritmos paralelos.

```
1:Função dep( a , b )
2:c ← a · b
3:d ← 2 · c
4:FimFunção
```

- Operação 3 depende da operação 2!

```
1:Função dep( a , b )
2:c ← a · b
3:d ← 2 · b
4:e ← a + b
4:FimFunção
```

- Não existe dependência!

POSIX Threads

- **POSIX threads** é um padrão para threads, o qual define uma API para criar e manipular threads.
- As bibliotecas que implementam a POSIX threads são chamadas Pthreads, sendo muito difundidas no universo Unix, Linux.

Criando Threads

Sintaxe:

```
int   pthread_create (pthread_t thread, const pthread_attr_t
        atributo, (void *)*função(void *), void * param )
```

Parâmetros:
thread - objeto thread usado no programa
atributo - atributo da thread. Se NULL, a thread usa os valores padrão
função - ponteiro para a função que implementará as funcionalidades da thread
param - parâmetro simples da função que implementará as funcionalidades da thread
Retorno:
Sucesso se zero, caso contrário retorna código do erro.

Finalizando Threads

Sintaxe:

```
void   pthread_exit (void * status)
```

status - status de finalização. Geralmente NULL
A função pthread_exit() é usada para finalizar a thread de forma explícita.
Se o método main finaliza com a chamada de pthread_exit(), todas as outras threads continuam seu processamento. Caso contrário, as outras threads são encerradas.

Exemplo 1

- Exemplo 1 Baixar Código
- Para compilar use o comando `g++ nome.c -pthread -o executavel`

Exemplo 2

- Exemplo 2 ▸ Baixar Código

Sincronizando Threads

- *Join* é um mecanismo para sincronizar as threads;
- Bloqueia quem chamou o join até que a thread alvo seja finalizada;
- O programador pode obter o estado de finalização thread através do atributo *status* na chamada da função `phtread_exit(status)`.

Sincronizando Threads

Sintaxe:

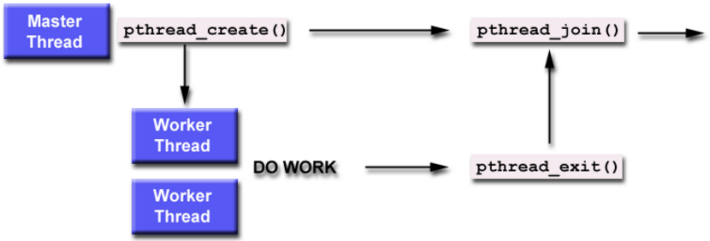
```
void   pthread_join (pthread_t thread, void * *status)
```

thread - objeto thread a qual deseja esperar finalização
status - estado de retorno da thread especificada no primeiro parâmetro

Retorno:

Zero se sucesso ou código do erro.

Sincronizando Threads



Exemplo 3

- Exemplo 3 ▸ Baixar Código

Região Crítica

- Um conjunto de instruções que pode ter apenas uma thread a executar em um momento específico.

Região Crítica

- Um conjunto de instruções que pode ter apenas uma thread a executar em um momento específico.

Condição de Corrida

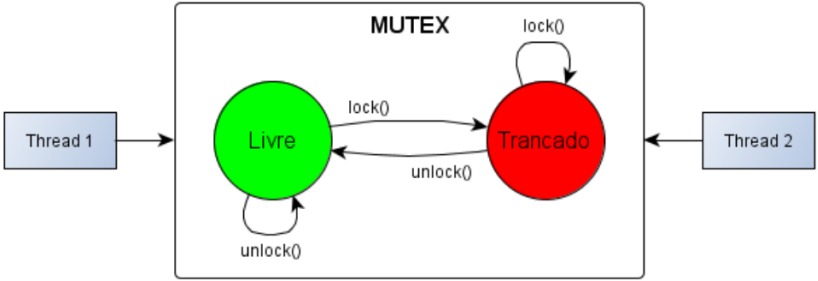
Ocorre quando mais de uma thread tenta acessar o mesmo espaço de memória simultaneamente.

Exemplo 2

- Adapte a função **thread_function** do Exemplo 1 para que faça o incremento de uma variável compartilhada (count). Cada thread deve incrementar 10000 vezes.

Exclusão mútua

- Exclusão mútua (**Mutex**) é uma técnica usada em programação concorrente para evitar que duas threads tenham acesso simultaneamente a uma região crítica.



Pthread Mutex

```
 int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t
*mutexattr);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Exemplo - Mutex

- Exemplo Mutex ▸ Baixar Código

Count 3

- Como seria o algoritmo para contar quantos números 3 existem em uma sequência numérica?

Count 3

- Como seria o algoritmo para contar quantos números 3 existem em uma sequência numérica?

```
1       for (i = 0; i < size; i++)
2           if (array[i] == 3)count++;
```

Count 3

- Como seria o algoritmo para contar quantos números 3 existem em uma sequência numérica?

```
1       for (i = 0; i < size; i++)
2           if (array[i] == 3)count++;
```

- Agora faça a versão paralela!

Count 3

- Como seria o algoritmo para contar quantos números 3 existem em uma sequência numérica?

```
1       for (i = 0; i < size; i++)
2           if (array[i] == 3)count++;
```

- Agora faça a versão paralela!
- Ajuda dos universitários! ▸ Baixar Código

Count 3

- Como seria o algoritmo para contar quantos números 3 existem em uma sequência numérica?

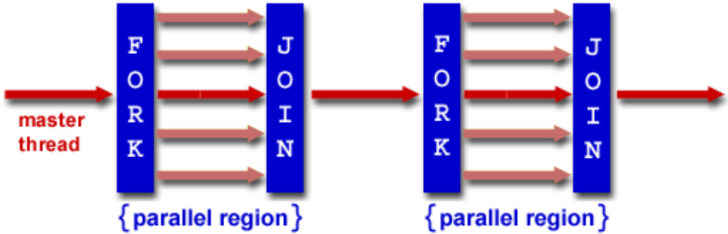
```
1       for (i = 0; i < size; i++)
2           if (array[i] == 3)count++;
```

- Agora faça a versão paralela!
- Ajuda dos universitários! ▸ Baixar Código
- Código completo ▸ Baixar Código

OpenMP

- OpenMP (*Open Multi-Processing*) é uma API multiplataforma para multiprocessamento, usando memória compartilhada.
- Composta por um conjunto de diretivas para o compilador, funções de biblioteca e variáveis de ambiente as quais especificam a implementação de um programa paralelo em C/C++.

Região Paralela



- O programa começa com uma única thread: **master thread**.
- fork: o master thread inicia um conjunto de threads.
- join: somente o master thread continua a executar.

Região Paralela

```
/* sequential, master thread ... */
#pragma omp parallel [clause...]
{
/* parallel, all threads ... */
}
/* sequential, master thread ... /*
```

- A estrutura é sempre **#pragma omp diretiva clausulas**.
- A diretiva **Parallel** cria novas threads.
- A quantidade de threads é igual ao número de núcleos, quando não especificado a quantidade de threads.

Região Paralela

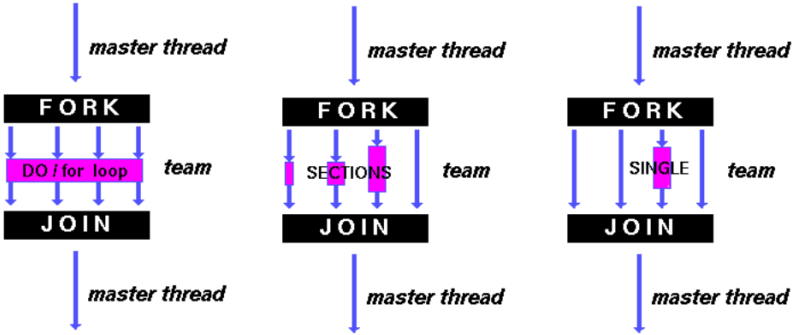
- Cláusulas:
  - **num_threads**(integer-expression)
  - **default**(shared | none)
  - **private**(list)
  - **firstprivate**(list)
  - **shared**(list)
  - **reduction**(operator: list)

Hello World!

```
#include <omp.h>

int main ()
{
  int nthreads, tid;
  /* Fork a team of threads with each thread
    having a private tid variable */
  #pragma omp parallel private(tid)
  {
    /* Obtain and print thread id */
    tid = omp_get_thread_num();
    printf("Hello World from thread = %d\n", tid);
    /* Only master thread does this */
    if (tid == 0) {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
  }
  /* All threads join master thread and terminate */
}
```

Divisão de Tarefas



Divisão de Tarefas
OpenMP For

```
#pragma omp for [clause...]
for (init;test;incr){
/* ... */
}
```

- Deve estar dentro de um bloco paralelo.
- Restrições sobre a variável de laço e expressões de controle:
  - Variável do laço deve ser int, iterator ou ponteiro.
  - O teste deve ser um dos <, <=,>,> =
  - O incremento pode ser ++, -, + =, - =, +, -
  - O teste e o incremento deve ser invariante dentro do laço.
  - pode haver vários laços aninhados.
  - O programador é responsável por dependências dentro das iteração.

Divisão de Tarefas
OpenMP For

```
#pragma omp for [clause...]
for (init;test;incr){
/* ... */
}
```

- Cláusulas:
  - **private**(list)
  - **firstprivate**(list): variavel privada a uma task. Iniciada com o valor corrente.
  - **lastprivate**(list): variavel privada a uma task. Atualizada no final da região paralela.
  - **reduction**(operator: v): Atualiza a instância sequêncial da lista de variáveis listada com a combinação de valor de todas as threads de acordo com a operação.
  - **schedule**(kind[, chunk_size]): static, dynamic, guided, auto
  - **Nowait**: Retira barreira implícita.

Divisão de Tarefas
OpenMP Section

```
#pragma omp sections [clause ...]
{
  #pragma omp section
  {
  /* ... */
  }
  #pragma omp section
  {
  /* ... */
  }
}
```

- Várias seções são executadas em paralelo.
- Cada seção é executado uma vez por uma thread.

Divisão de Tarefas

OpenMP Section

```
omp_set_num_threads(2);
#pragma omp parallel sections
{
  #pragma omp section
  {
    printf("secao 1\n");
    for (i=0;i<20000;i++){
      printf("a");
      fflush(stdout);
    }
  }

  #pragma omp section
  {
    printf("secao 2\n");
    for (i=0;i<20000;i++){
      printf("b");
      fflush(stdout);
    }
  }
```

Divisão de Tarefas
OpenMP Single

```
#pragma omp parallel [clause...]
{
  /* ... */
  #pragma omp master
  {
  /* ... */
  }
  #pragma omp single
  {
  /* ... */
  }
}
```

- Declaração ou bloco executado por uma única thread.
- A diferença é que
  - a master é uma thread específica, single é qualquer thread.
  - existe uma barreira implícita depois de bloco single, porém não há depois do master.

Divisão de Tarefas

OpenMP Single

```
#pragma omp parallel
{
  printf("TESTE %d\n",omp_get_thread_num());
  fflush(stdout);
  #pragma omp single
  {
    printf("SINGLE %d\n",omp_get_thread_num());
    fflush(stdout);
  }
  printf("XXX\n");
  fflush(stdout);
}
```

OpenMP barrier

```
/* before */
#pragma omp barrier
/* after */
```

- Nenhuma thread cruza barreira até as outras threads atingirem a barreira.
- Algumas diretivas omp possui uma barreira implícita.
- A cláusula nowait pode remover a barreira implícita.

OpenMP critical

```
#pragma omp critical [name]
{
/* ... */
}
```

- Declara uma região crítica.
- Todas as regiões críticas com o mesmo `name` são mutuamente exclusivos.
- Regiões críticas sem nome indicam a mesma região crítica.

Práticas

- Refaça em OpemMP:
  - O `Count 3`.
  - O incremento de uma variável - Exemplo 2.

- Mutex em OpenMp.

Desempenho de Algoritmos Paralelos

- É importante entender até que ponto é vantajoso utilizar programas paralelos.
- O objetivo é determinar os benefícios do paralelismo aplicados a um problema considerado.
- E também o quanto o algoritmo é capaz de continuar eficiente.

Speedup e Eficiência

$$S = \frac{T_S}{T_P}$$

- O ***speedup*** S diz quantas vezes o algoritmo paralelo é mais rápido que o algoritmo serial.

Speedup e Eficiência

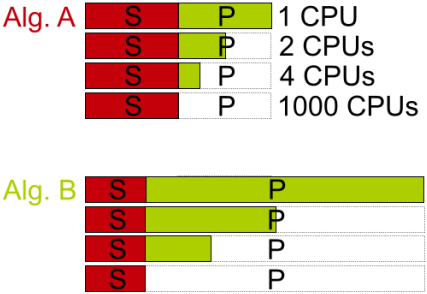
$$E = \frac{S}{P} = \frac{\frac{T_S}{T_P}}{P} = \frac{T_S}{PT_P}$$

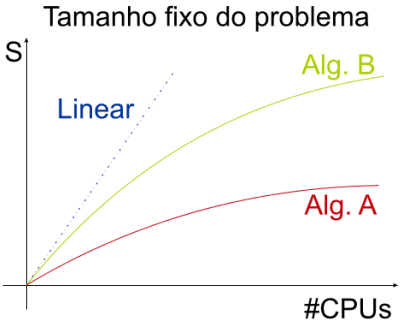
- A eficiência é uma medida normalizada de *speedup* que indica o quão efetivamente cada processador é utilizado.
- P é número de *threads*.
- Um *speedup* com valor igual a P, tem uma eficiência igual 1, ou seja, todos processadores são utilizados e a eficiência é linear.

Lei de Amdahl

- Segundo a Lei de Amdahl (do inglês, *Amdahl's law*), a velocidade de processamento paralelo é limitada a porção sequencial do programa.
- Essa porção do programa que não pode ser paralelizada limitará o aumento de velocidade disponível com o paralelismo.

Lei de Amdahl

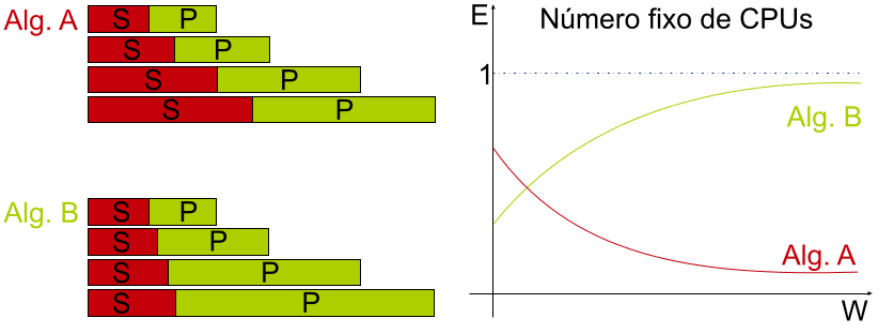




Escalabilidade Paralela

- A palavra escalável tem uma grande variedade de significados em diversas áreas.
- Informalmente, a tecnologia é escalável quando ela pode lidar com problemas cada vez maiores sem perdas de desempenho.

Escalabilidade Paralela



Fim

Obrigado!