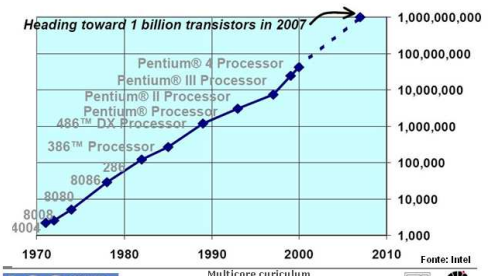
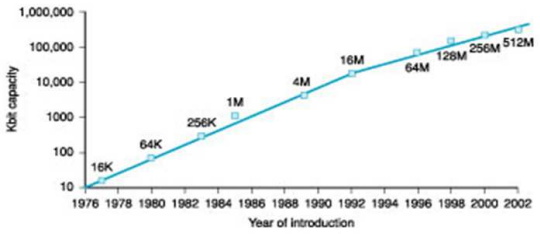
Multicore

Motivation

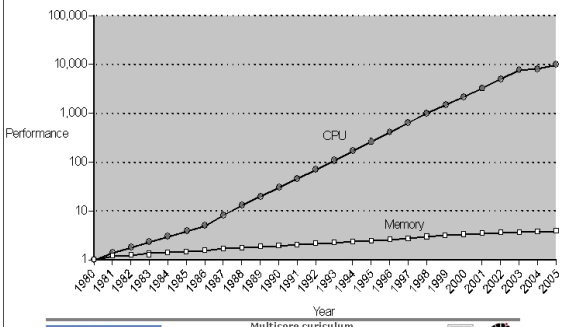
Moore's Law: the number of transistors double every 18 months



Memory capacity also increases



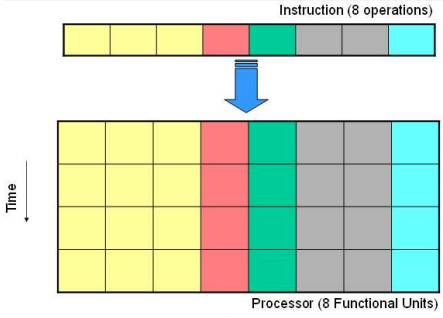
The Memory Wall



How to go parallel?

- VLIW Processors
- Superescalar Processors
  - Hyperthread
- Multi-core

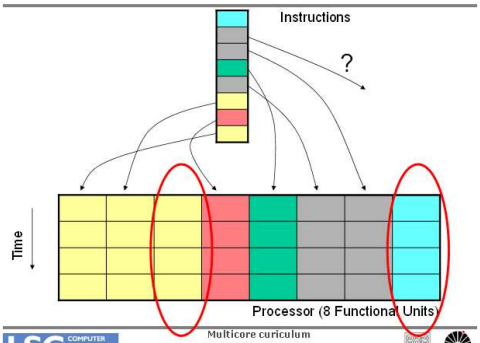
Very Long Instruction Word



VLIW

- Advantages
  - Easy to implement in hardware
    - Several similar tiles
    - Do not require a huge control logic
- Disadvantages
  - Difficult to generate good code

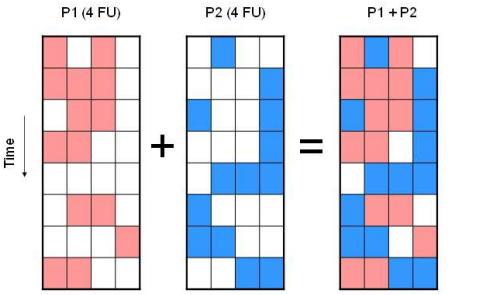
Superscalar Processor



Superscalar Processor

- Advantage
  - Transparent to the software
  - The processor is able to use dynamic information to find the parallelism
  - Speculative code execution
- Disadvantage
  - Can not always find instruction for each functional unit
  - Detecting parallelism in hardware requires a lot of area

Hyperthreading Technology



Hyperthreading Technology

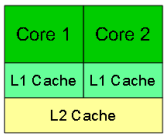
- Requirements
  - 2 Different
    - Program counter
    - Register banks
    - Status registers
  - The same
    - Functional units
    - Caches

Hyperthreading Technology

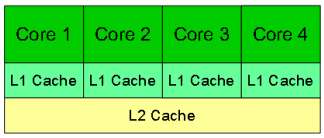
- Advantage
  - Uses the available functional units to execute a second thread
  - Capable of executing code during a stall of the other thread (cache miss, etc)
- Disadvantage
  - Threads usually need the same functional unit
  - 2 threads at the same time, but only 30% of typical speedup

Chip Multiprocessing (CMP)

2 Cores

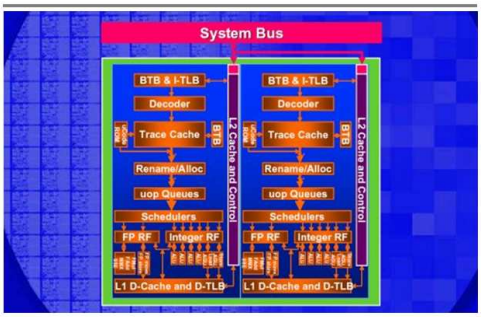


4 Cores



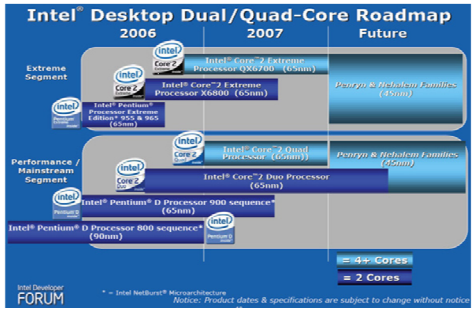
A cache L2 também pode ser dividida!

Pentium D Processor Diagram

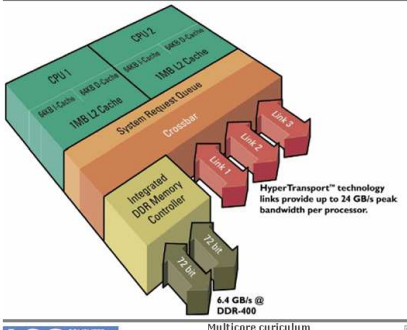


Intel Dual Core Pentium

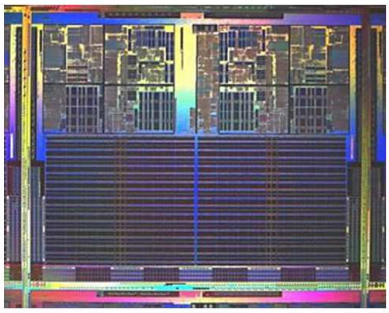
Intel Roadmap



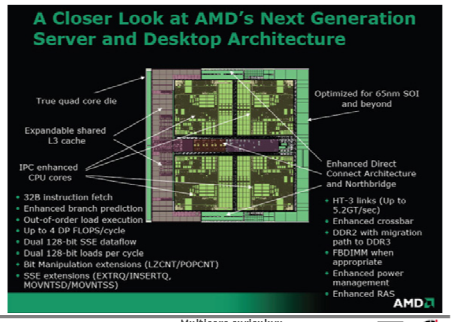
AMD Dual Core



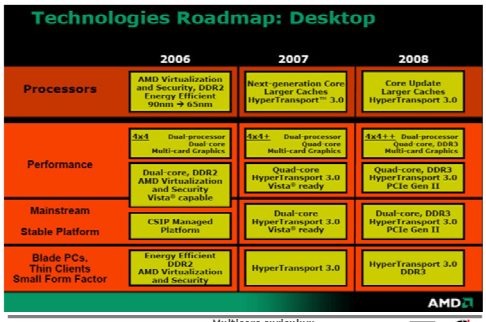
AMD Dual Core



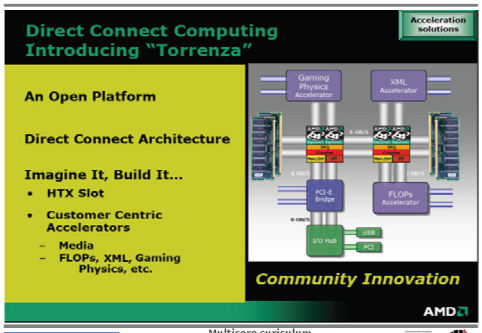
AMD Quad Core



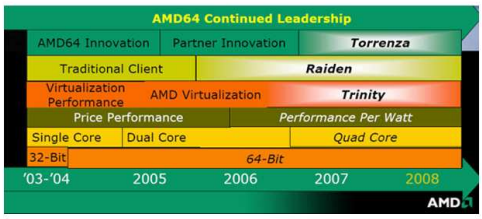
AMD Roadmap



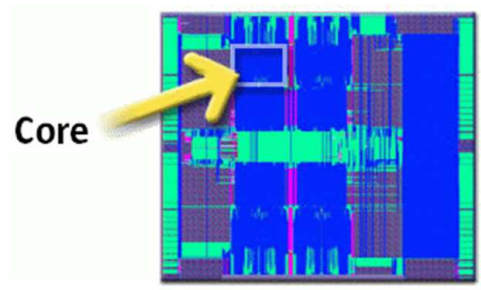
AMD abre o padrão de barramento



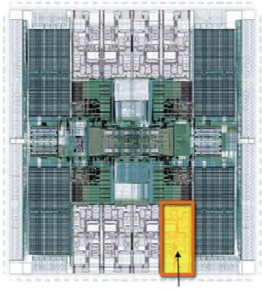
AMD Vision Roadmap



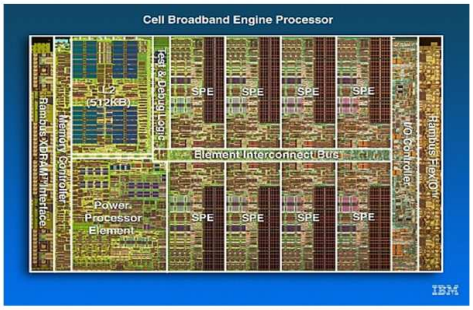
SUN Niagara



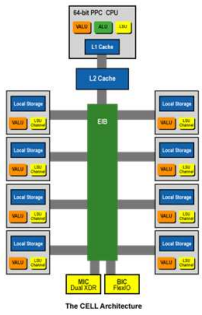
SUN Niagara



IBM Cell



IBM Cell

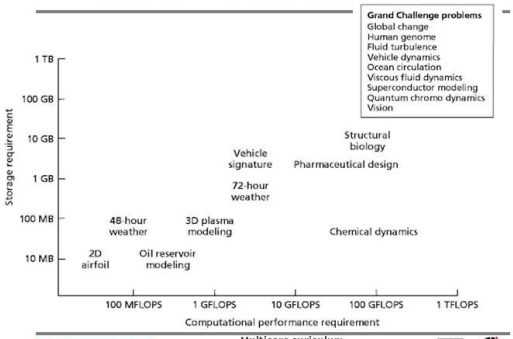


The CELL Architecture

Chip Multiprocessing

- Advantages
  - Simpler cache coherency circuit
  - Simpler chip packaging
  - Sharing L2 reduces the total chip area
  - Easier to replicate the same, already tested, cores several times in the circuit
- Disadvantages
  - Requires software redesign
  - The memory wall problem increased

Scientific Computing Demand



Análise de Desempenho

- Como medir o desempenho de um software?
  - Tempo
    - Usuário
    - Sistema
    - Real
- Como medir o desempenho de um software multithread?
  - Tempo?

Como medir o tempo?

- Exemplo:
  - Processo totalmente paralelizável

	1 Proc.	2 Proc.	4 Proc.
Usuário	20s		
Sistema	2s		
Real	22s		

Como medir o tempo?

- Exemplo:
  - Processo 60% paralelizável

	1 Proc.	2 Proc.	4 Proc.
Usuário	20s		
Sistema	2s		
Real	22s		

Espaço de Endereçamento

- Como garantir o compartilhamento de dados através da memória?
- Espaço de endereçamento único
  - O endereço 1000 para processador 1 é exatamente o mesmo para qualquer outro
  - Mecanismo fortemente dependente da interconexão
  - Não escala muito bem no caso de barramentos

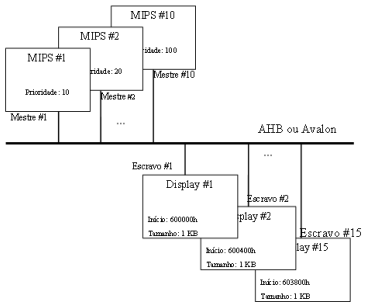
Endereçamento

- Multiprocessamento simétrico
  - Capacidade de qualquer processador executar qualquer tarefa, acessando os mesmos dados
- Arquiteturas NUMA
  - Cada processador tem seu espaço de endereçamento
  - São necessárias operações especiais para transferir dados entre os processadores

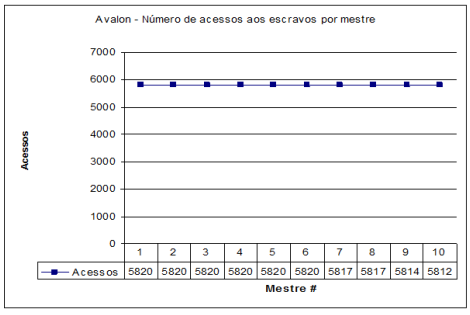
Barramentos

- Fonte de contenção de sistemas grandes
- Experimentos com simulação
  - Vários mestres (processadores)
  - Mestres com prioridades diferentes
  - Diferentes quantidades de escravos (memórias)
  - Variação na quantidade de memórias

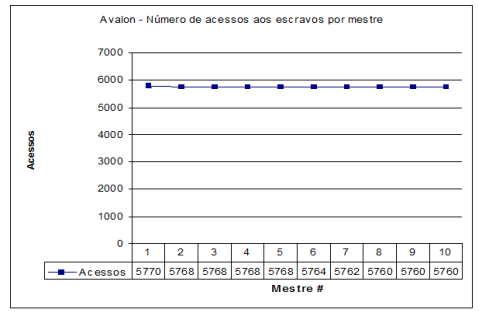
Esquema do Experimento



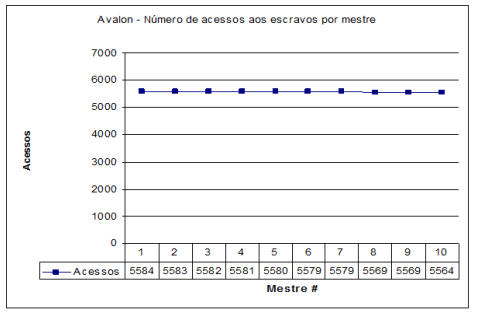
10 mestres e 15 escravos



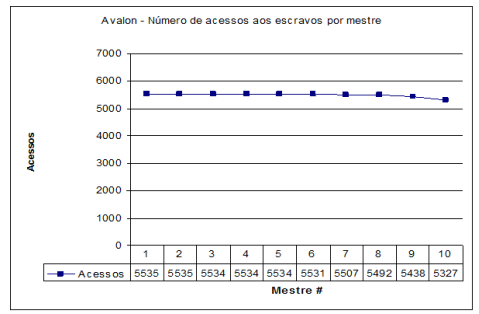
10 mestres e 10 escravos



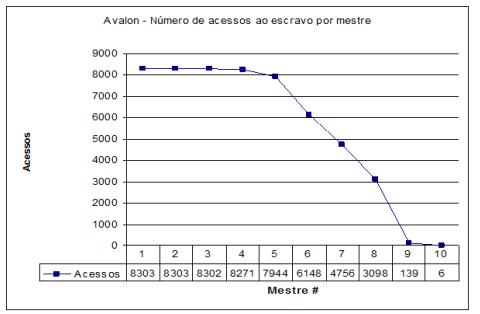
10 mestres e 5 escravos



10 mestres e 2 escravos



10 mestres e 1 escravo



How to program?

- Operating systems support
  - Linux, µcLinux, VxWorks, WinCE, etc.
  - Every process/thread in a different core
  - Several programs can execute at the same time → real parallel execution
- Thread libraries
  - pthreads, Windows Threads, OpenMP
  - Take advantage of multiprocessing inside the same program

Programming

- Operating systems concepts
  - Process
    - Basic code execution unit
    - Memory protection between two different process
  - Threads
    - Process sub-unit
    - One process can have more than one thread
    - No memory protection between two threads of the same process
    - No overhead to share memory between two threads

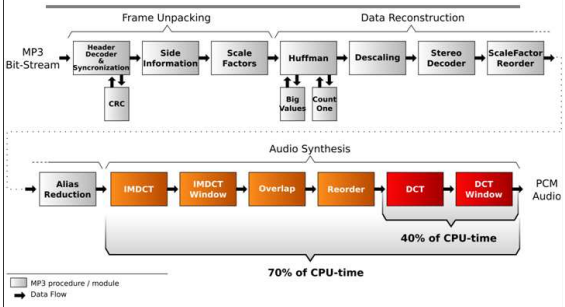
New Programming Issues

- Identify the concurrency
  - Break the program into several parts that can be run in parallel
- Algorithm design
  - Describe, efficiently, the program parts to run in parallel
- Communication
  - Efficiently share data between every program part

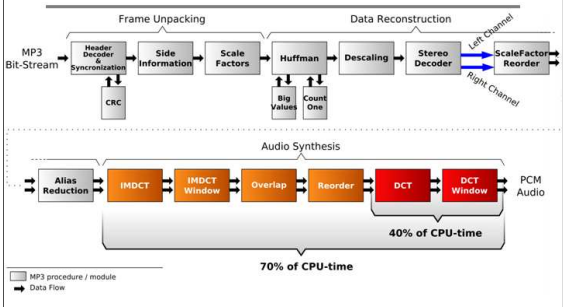
Program Classification

- Trivially parallelizable (10%~20%)
  - Just replicate the code for each core and execute the same algorithm
  - Ex.: Web server, some multimedia algorithm
- Parallelizable (~60%)
  - Requires some program, or data structure, adaptation to work in parallel
- Difficult to parallelize (20%~30%)
  - The algorithm needs to be redesigned

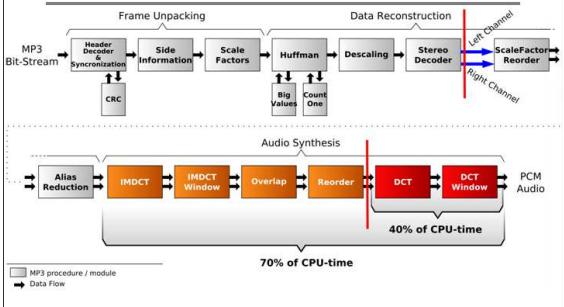
Exemplo: Como paralelizar?



Separar canais direito e esquerdo



Quebrar em fases



Using threads

- Two usual thread libraries
  - pthreads
  - Windows Threads
- Advantages
  - Control of the parallelism
  - Explicit code division
  - Explicit API available

OpenMP

- Library for code parallelization
- Uses **#pragma** to define parallel regions
  - The programs can still be executed serially
- Automatically detects the number of processors
  - The programmer is not required to know the number of processors available
- Needs compiler support

Example

```
#include <stdio.h>

main()
{
#pragma omp parallel
  {
    puts("Hello
World");
  }
}
```

- 1 processor
  Hello World
- 4 processors
  Hello World
  Hello World
  Hello World
  Hello World

Example: (primo.c)

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

#define LIMIT 5000000

int prime(int n)
{
  int root, f;

  root = (int) sqrt((double)
n);

  for(f = 2; f <= root; f ++)
    if (n % f == 0)
      return 0;

  return 1;
}
```

```
int main()
{

  int quant = 0, n;
  for(n = 2; n < LIMITE; n ++) {
    quant += prime(n);

  printf("Number of primes lower
than %d: %d\n", LIMIT, quant);
}
```


Com OpenMP

```
int main()
{

  int quant = 0, n;

#pragma omp parallel for
  for(n = 2; n < LIMITE; n ++) {
    int p = prime(n);
#pragma omp critical(quant)
    quant += p;
}

  printf(" Number of primes lower than
%d: %d\n", LIMIT, quant);
}
```


Results

- Both code executed dual Xeon with hyperthread (4 logic processors)
- Without OpenMP
  - 16,49s
- With OpenMP
  - 6,84s

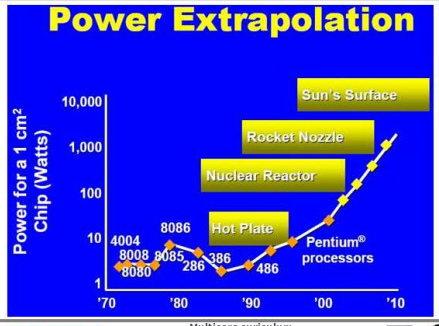
Outras técnicas

- Outras técnicas de aumentar o desempenho tirando proveito do paralelismo:
  - Biliotecas especiais SIMD
    - Intel IPP
    - Intel MKL
  - Helper threads

De onde tirar desempenho?

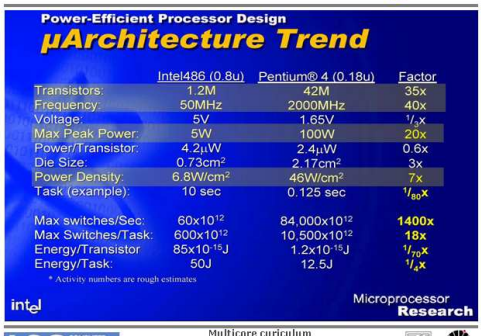
- Sistemas convencionais estão chegando no limite de freqüência
  - Consumo de energia está aumentando
- Muitos transistores disponíveis por chip
  - Maior capacidade de produção de hardware?
- Uma possível solução
  - Sistemas multicores

Consumo de energia



Fonte: Intel

Eficiência no uso de energia

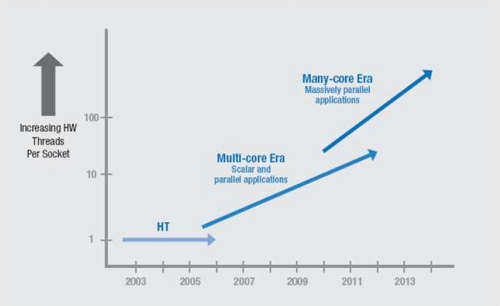


Fonte: Intel

Por que reduzir o consumo?

- Prolongar a duração de bateria
- Evitar problemas de aquecimento
- Minimizar tamanho de dissipador
  - Redução de área
  - Redução de peso
- Garantir usabilidade
  - Ex.: Sensação de teclado quente

Paralelismo: Visão da Intel



Paralelismo em software

- Sistemas operacionais
  - Linux, µcLinux, VxWorks, WinCE, etc.
- Bibliotecas de threads
  - pthreads, Windows Threads, OpenMP
- Funcionalidade mínima
- Detalhes mais adiante

Motivação

- Dificuldade em aumentar a freqüência de operação sem aumentar o consumo de energia
- A quantidade de transistores continua aumentando
- O que fazer com eles?
  - Novos processadores no mesmo circuito integrado

Tipos de multiprocessamento

- Homogêneo
  - Vários processadores idênticos
  - Exemplo:
    - Dual ARM, Pentium 4 Duo, etc.
- Heterogêneos
  - Processadores diferentes
  - Exemplo:
    - RISC + DSP dentro de um celular ou tocador de MP3

Novos problemas

- Identificar a concorrência
  - Quebrar o problema em várias partes que podem ser atacadas em paralelo
- Projeto do algoritmo
  - Descrever, de forma eficiente, em várias partes paralelas, a solução para o problema
- Comunicação
  - Usar meios eficientes para compartilhar dados entre as várias partes do programa

Abordagens

- Com sistema operacional
  - Dois conceitos já existentes: Processos e Threads
  - Processos
    - Unidade de execução de código
    - Proteção de memória entre processos
  - Threads
    - Sub-unidade de execução
    - Um processo pode possuir várias threads
    - Sem proteção de memória entre threads de um mesmo processo

Usando threads

- Duas bibliotecas usuais
  - pthreads para Unix
  - Windows Threads
- Vantagens
  - Controle do paralelismo
  - Divisão explícita do código
  - APIs disponíveis na forma de chamadas de funções
- São mesmo vantagens?

OpenMP

- Biblioteca para paralelização de código
- Utiliza **#pragma** para definir regiões paralelas
  - Os programas devem continuar funcionando na forma serial
- Detecta o número de processadores
  - O programador não precisa definir o número de threads
- Suporte de apenas alguns compiladores

Exemplo de uso

```
#include <stdio.h>

main()
{
#pragma omp parallel
   {
     puts("Hello
World");
   }
}
```

- Saída com 1 processador
  Hello World
- Saída com 4 processadores
  Hello World
  Hello World
  Hello World
  Hello World

Exemplo: (primo.c)

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

#define LIMITE 5000000

int primo(int numero)
{
  int raiz, fator;

  raiz = (int) sqrt((double)
numero);

  for(fator = 2; fator <= raiz;
      fator++)
    if (numero % fator == 0)
      return 0;

  return 1;
}
```

```
int main()
{

  int quantidade = 0, numero;
  for(numero = 2;
      numero < LIMITE;
      numero ++) {
    quantidade += primo(numero);

  printf("Total de numeros primos
ate %d: %d\n", LIMITE,quantidade);
}
```


Com OpenMP

```
int main()
{

  int quantidade = 0, numero;

#pragma omp parallel for schedule(static, 8)
  for(numero = 2; numero < LIMITE; numero ++) {
    int p = primo(numero);
#pragma omp critical(quantidade)
    quantidade += p;
}

  printf("Total de numeros primos ate %d: %d\n",
LIMITE, quantidade);
}
```


Resultados

- Executado num dual Xeon com hyperthread (4 processadores lógicos)
- Sem OpenMP
  - 16,49s
- Com OpenMP
  - 6,02s
- Sem schedule(static, 8)
  - 6,84s

Tipos de programas

- Facilmente paralelizáveis (10%~20%)
  - Basta replicar o código entre vários processadores e/ou fragmentar os dados
  - Ex.: Servidores web, alguns algoritmos multimídia
- Paralelizáveis (~60%)
  - Exigem modificação no algoritmo para funcionarem em paralelo
- Difíceis (20%~30%)
  - O algoritmo tem que ser repensado, reprojetado ou mesmo substituído