Explorando o Paralelismo no nível de *Threads*

Marcio Machado Pereira
Luiz A. Biazotto Filho
UNICAMP, BRAZIL , jun 2011

Roteiro

- ✓ Motivação
- ✓ Arquiteturas *Multithreading*
  - ➢ SMT
  - ➢ CMP
- ✓ Extração Automática de *Threads*
  - ➢ *Coarse-Grained* TLP
  - ➢ *Fine-Grained* TLP
  - ➢ Paralelização Semi-Automática
  - ➢ Programação Funcional
- ✓ Conclusão

Motivação [1]

- ✓ Limites do ILP
  - ➢ Unidades Funcionais “sub-utilizadas”.
- ✓ Barreira térmica
  - ➢ Arquiteturas “*multicores*”

→ Exploração do Paralelismo em um nível mais alto de abstração (e.g., Processos, *Threads*). Como?

- ➢ Aplicações em Linguagens de Programação Paralela

  (concorrência, custos de comunicação, localidade de dados)
- ➢ Extração de Paralelismo em Aplicações Sequenciais

  (eficaz em aplicações científicas e de multimídia)

Motivação [2]

Considere o seguinte programa que calcula o fatorial de um inteiro positivo:

```
factorial :: Int -> Int
factorial 1 = 1
factorial n = n * factorial ( n – 1 )
```

Um algorítimo paralelo para calcular o fatorial usa uma abordadem de divisão-e-conquista (menos intuítiva), como mostrado abaixo:

```
factorial :: Int -> Int
factorial n = product ( 1, n )
    where product ( lo, hi )
          | ( lo == hi ) = lo
          | otherwise  = product ( lo, mid ) * product ( mid + 1, hi )
                where mid = ( lo + hi ) `div` 2
```

Arquiteturas Multithreading

✓Diferentemente do ILP, que explora implicitamente operações paralelas dentro de laços e blocos de um programa, **TLP é explicitamente** representado pelo uso de várias *threads* de execução que são essencialmente paralelas;

✓Permite que múltiplas *threads* compartilhem as unidades funcionais de um único processador de maneira intercalada;

- processador duplica o estado independente de cada *thread* (*register files, PC, page table*);
- memória compartilhada (virtual, multiprogramação);

✓Hardware tem que suportar que a troca de *threads* seja feita mais rapidamente que a troca de processos (lenta).

Arquiteturas SMT

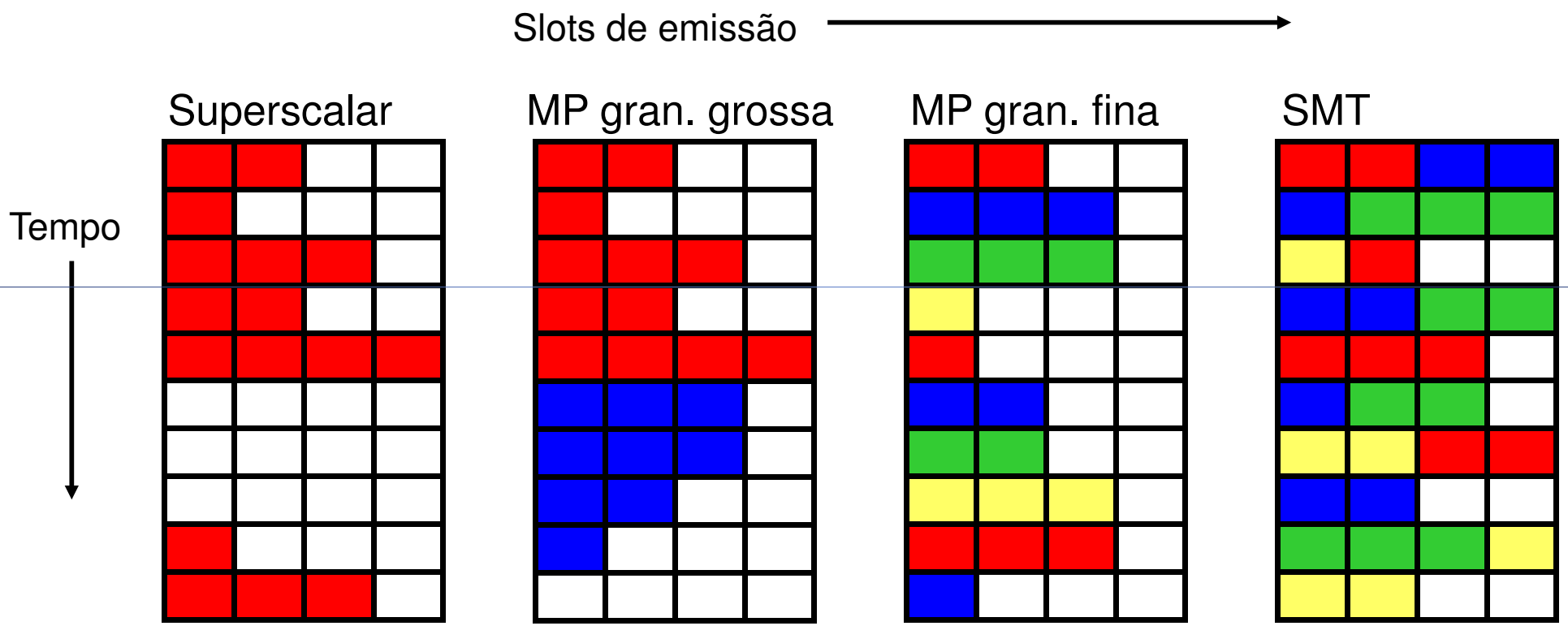
✓Herda características da granularidade fina, porém, a cada ciclo, a CPU pode despachar instruções de diferentes *threads* ao mesmo tempo (*schedule* dinâmico permite executar instruções de cada *thread* se possível);

✓Outras *threads* passam a utilizar as unidades de execução no caso da *thread* corrente ficar em longa latência;

✓A dependência entre *threads* é resolvida pelo mecanismo de despacho;

✓Utilizado pela tecnologia *HyperThreading* da Intel, introduzido nos processadores **Pentium IV e Xeon**. Recentemente, presente na família da micro-arquitetura **Nehalem**.

Utilização de unidades funcionais



Considerações:
cada cor representa um diferente *thread* nos processadores *multithreading*;
a figura não está considerando a perda de *throughput* para enchimento de *pipeline* na MP de granularidade grossa.

Arquiteturas CMP

✓Chip Multi-Processing: um grupo de processadores "*cores*" fisicamente integrados em um mesmo chip (*multicores*), onde cada um pode executar *threads* simultaneamente;

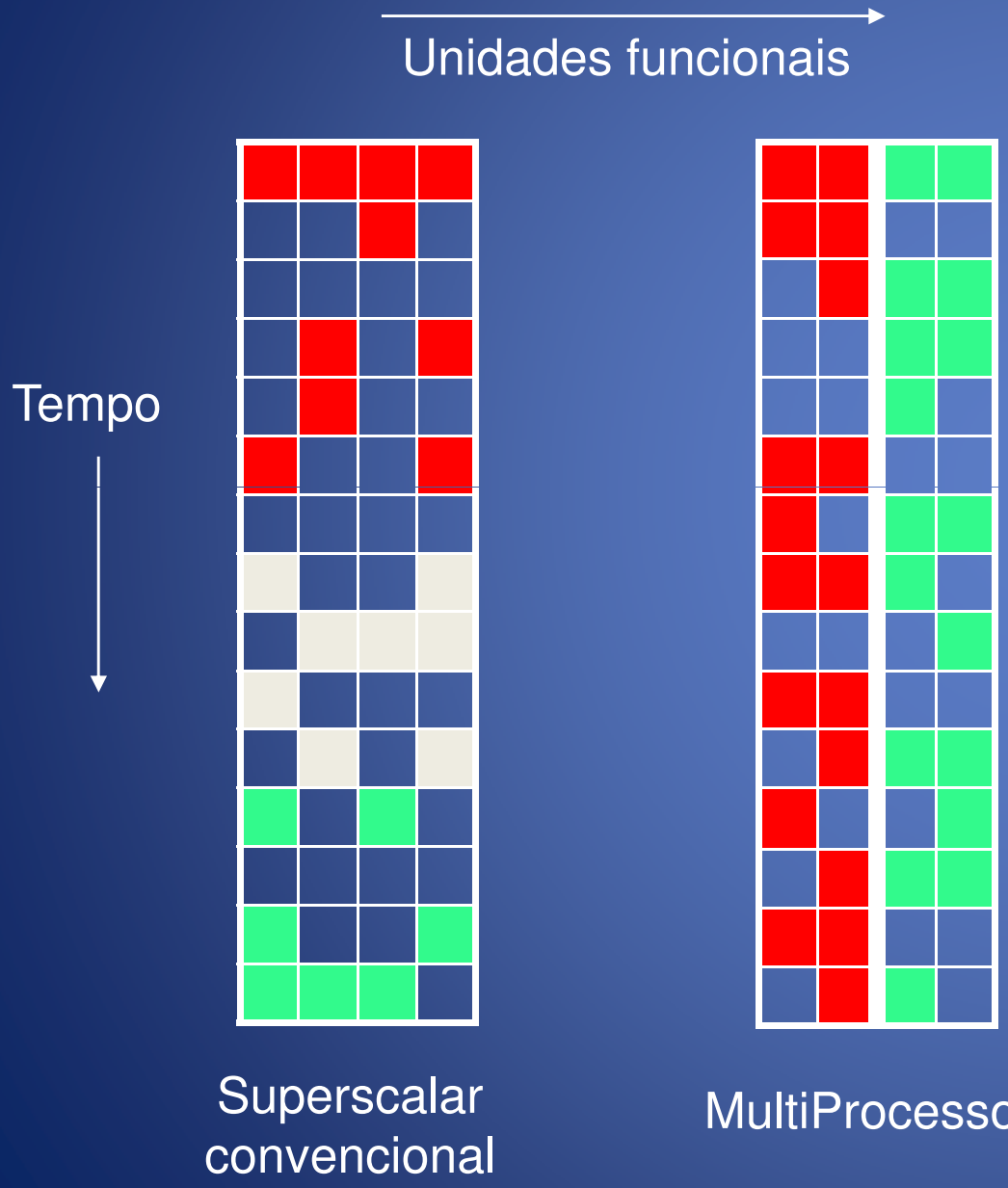
✓"Fácil" de ser implementado (engenharia);

✓Cada *core* é visto pelo programador como uma unidade de processamento separada;

- ➢ Obriga substituição do modelo computacional de Von Neumann por um novo modelo de programação paralela;

✓Sun T1 (**Niagara**) utiliza a granularidade fina como abordagem para *multithreading*. É capaz de despachar apenas uma instrução por vez, mas gerencia até quatro *threads*.

Arquiteturas CMP



✓CMP tenta eliminar a “perda horizontal” de unidades funcionais;

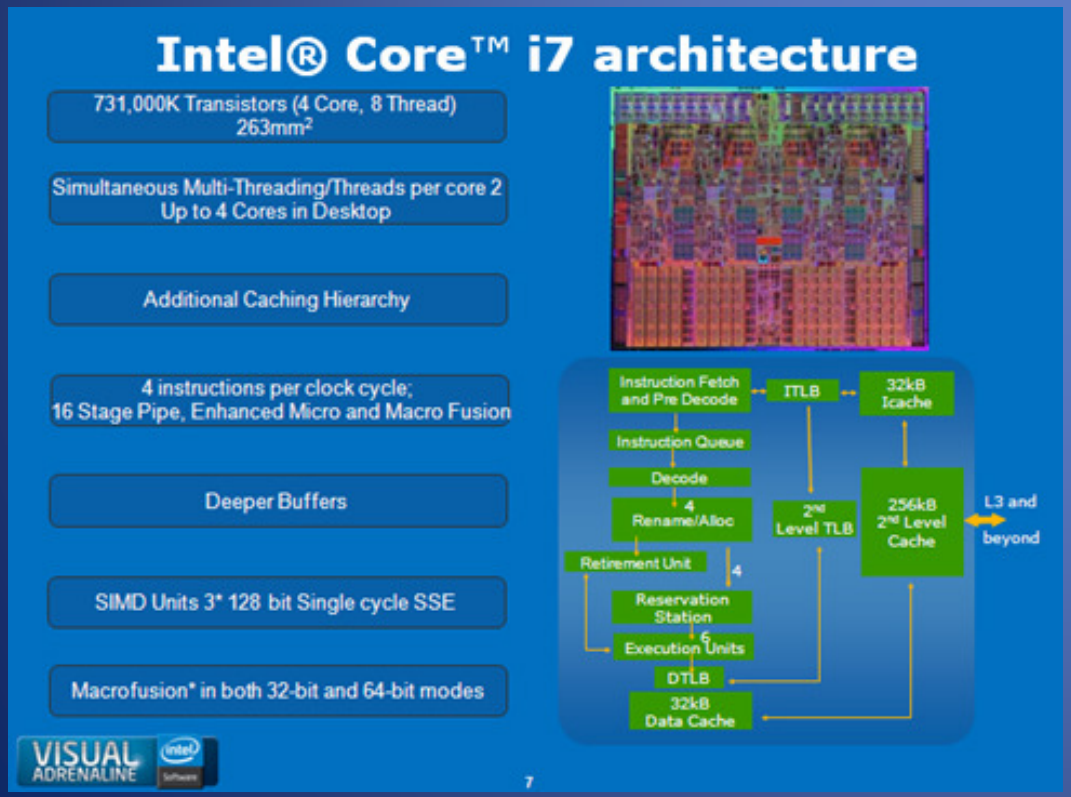
✓CMP é menos flexível para uso geral, pois caso haja apenas um *thread* em execução, ele só tem metade dos recursos disponíveis para utilizar;

✓Cor vermelha e verde representam threads diferentes; Cor cinza claro representa tempo de troca de contexto (inicio de interrupção, exceção, RPC até a volta dela).

Evolução CMP = CMT

✓CMT = CMP com *multithreading* em *hardware*, por exemplo, **Niagara II** e os atuais **Intel Core Nehalem**.

➢"Aproveita" o potencial dos chips *multicore*, implementando técnicas de troca de *threads* em hardware, que podem variar desde a granularidade grossa, fina até SMT.



• In 2008 the latest processor microarchitecture from Intel was introduced with the Intel Core i7 processor (Figure). It featured a native, monolithic quad-core processor for the first time from Intel and used fewer transistors than the Intel® Core™ 2 Quad processor (731 million versus 820 million). **Intel HTTechnology last seen on the Pentium 4 was reintroduced on the quad-core processor to allow the Intel Core i7 processor support for up to eight simultaneous threads together with increased resources in the out-of-order engine to support the increased number of threads.**

Fonte: http://software.intel.com/en-us/articles/who-moved-the-goal-posts-the-rapidly-changing-world-of-cpus/

Extração Automática de *Threads*

- ✓ Bem sucedido em domínios de aplicação restrita, principalmente onde o paralelismo de dados é abundante:
  - ➢ *Flat data parallel*
  - ➢ *Nested data parallel*

- ✓ Extração de Paralelismo em laços de programas
  - ➢ Granularidade Grossa (*Coarse-Grained* TLP)
  - ➢ Granularidade Fina (*Fine-Grained* TLP)

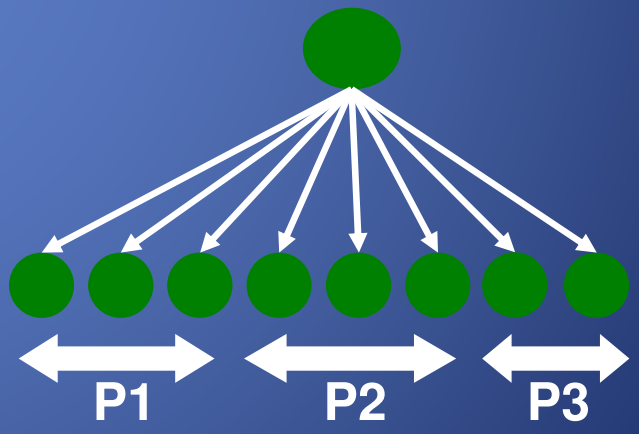
Flat data parallel

e.g. HPF Fortran, C/C++ OpenMP

- ✓ O líder de mercado: amplamente utilizado, bem entendido, com bom suporte

```
foreach i in 1..N {
    ...do something to A[i]...
}
```

- ✓ "**something**" é sequential
- ✓ Único ponto de concorrência
- ✓ Fácil de implementar: usa "*chunking*"
- ✓ Baixo custo



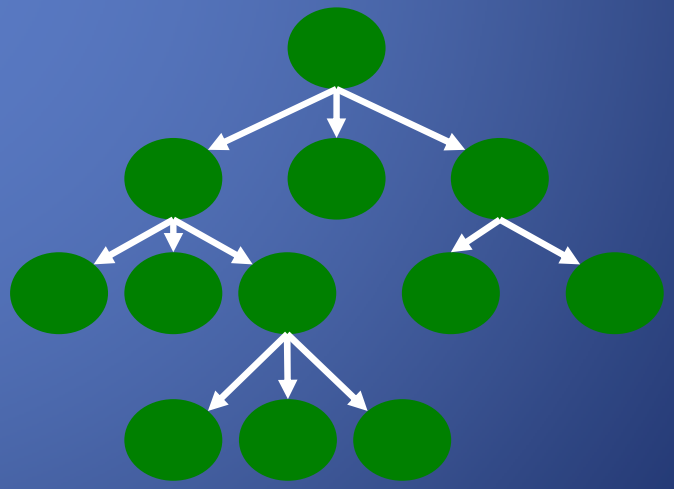
1,000,000's of (small) work items

Nested data parallel

- ✓ Idéia Central: permitir que **"something"** possa ser paralelo

```
foreach i in 1..N {
    ...do something to A[i]...
}
```

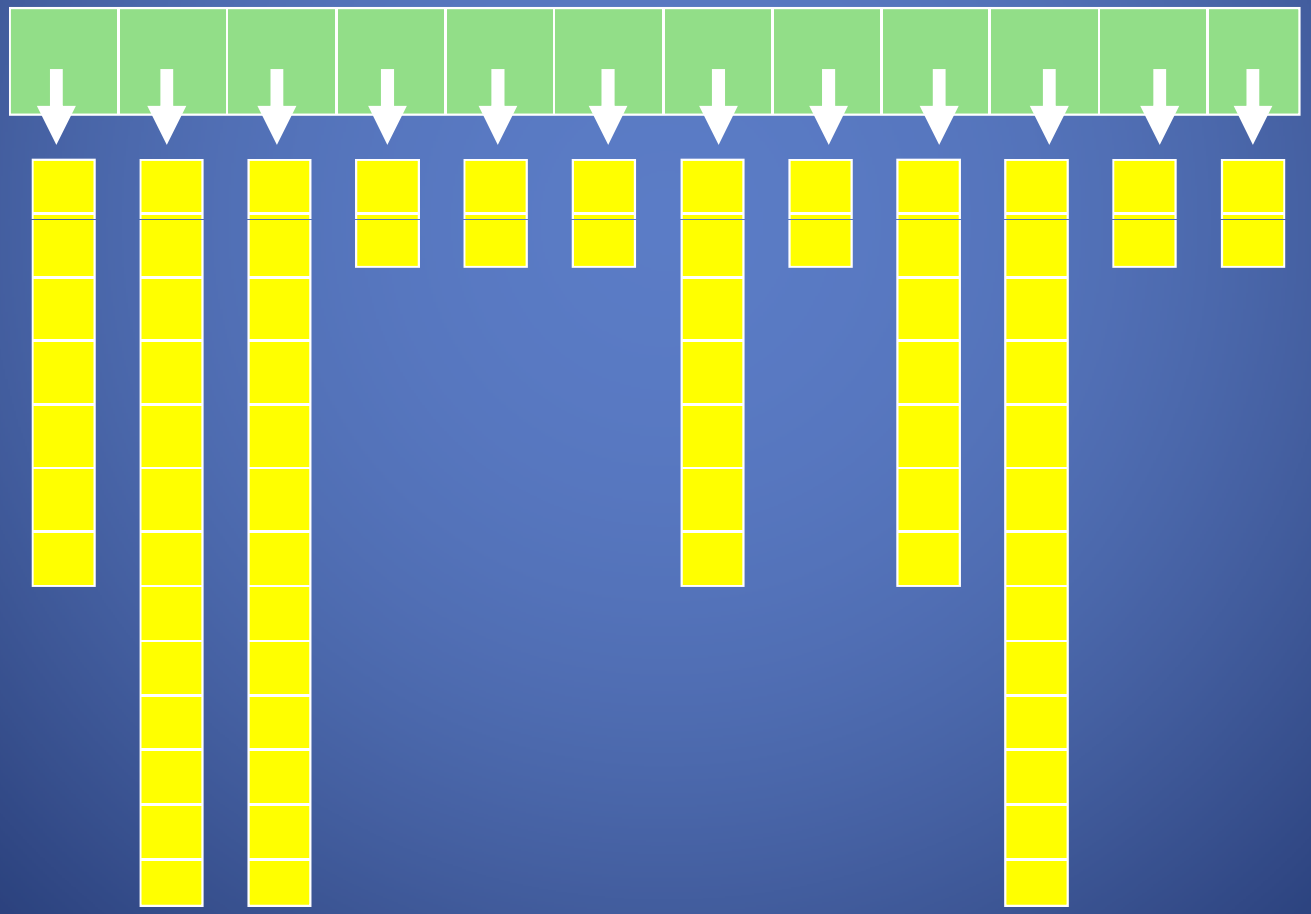
- ✓ Agora, a estrutura de paralelismo é recursiva, e não-balanceada
- ✓ O modelo de custos ainda é bom.



Still 1,000,000's of (small) work items

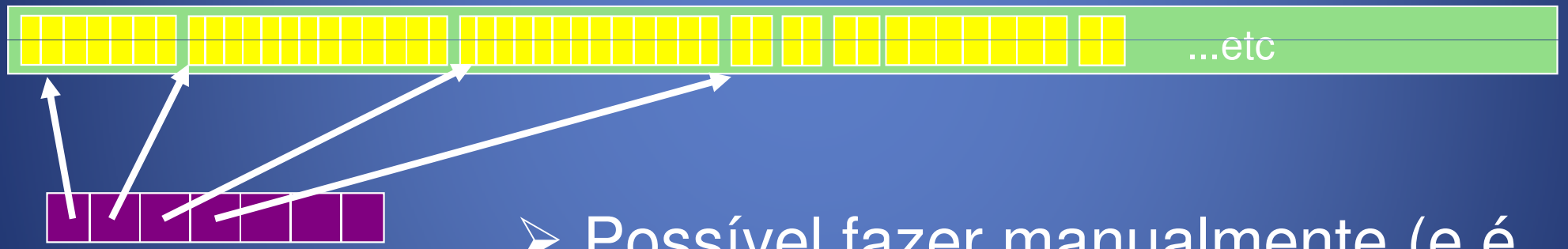
...difícil de implementar bem

- ✓Considere a seguinte estrutura de dados (que pode respresentar um grafo, uma matriz esparsa, etc.)



The flattening transformation

- ✓Concatena os sub-vetores em um grande vetor
- ✓Opera em paralelo neste grande vetor
- ✓ vetor auxiliar mantém índices dos sub-vetores



- ➢ Possível fazer manualmente (e é feito na prática), mas muito difícil de acertar
- ➢ Blelloch mostrou que poderia ser feito de forma sistemática

Extração de Paralelismo em Laços

- ✓ Granularidade Grossa (*Coarse-Grained* TLP)
  Concentra a grande maioria das técnicas de paralelização automática.
  - ➢ DOALL,
  - ➢ DOACROSS

- ✓ Granularidade Fina (*Fine-Grained* TLP)
  Explora mecanismos especializados do hardware. (bandwidth de comunicação mais elevado).
  - ➢ DSWP

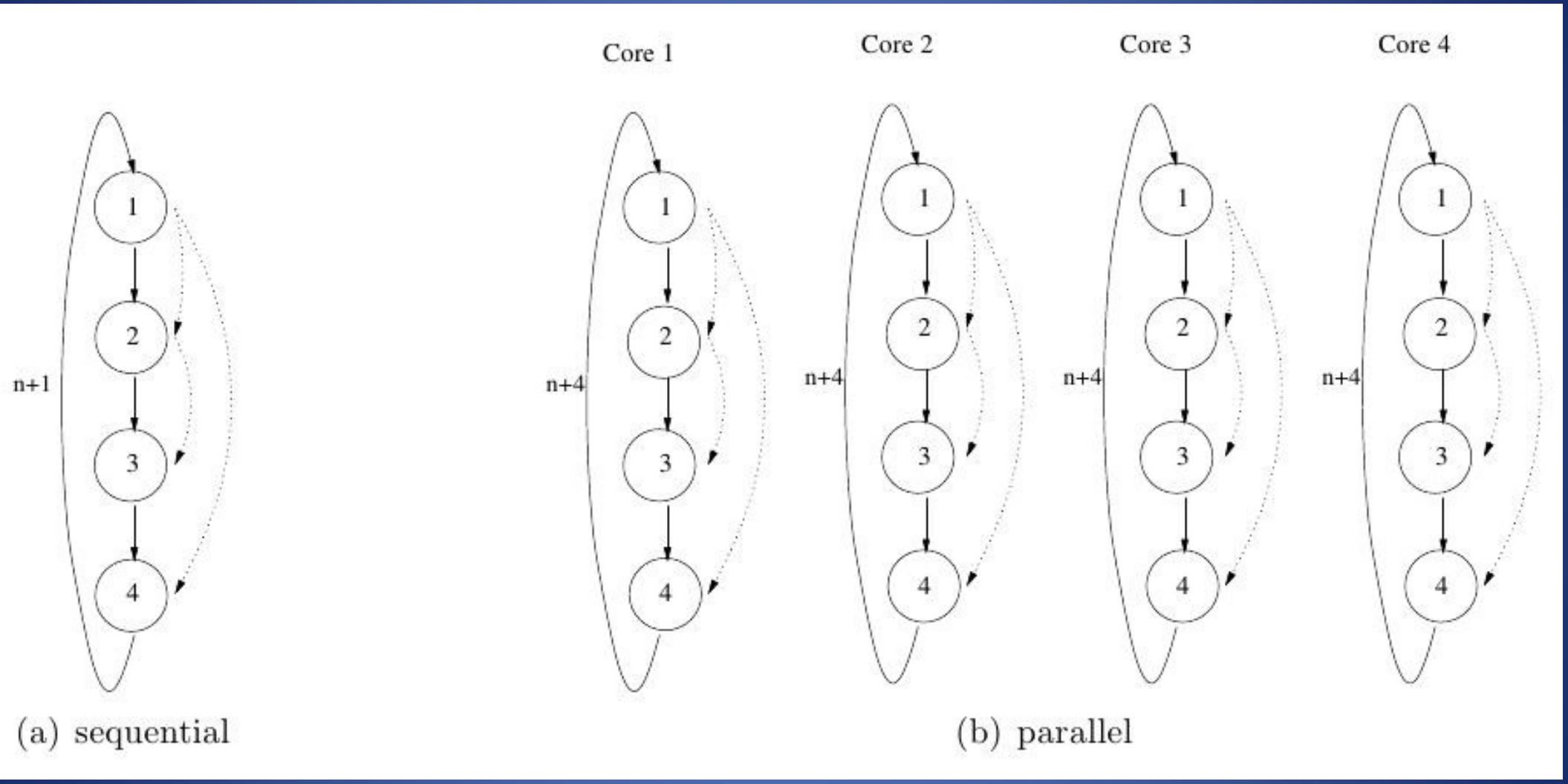
Coarse-Grained TLP [1]

- ✓ DOALL – laços sem dependência entre as iterações.

  Desafios:
  - ➢ Analisar um laço aninhado para provar que suas iterações são independentes;
  - ➢ Transformar um laço aninhado não DOALL em um equivalente DOALL.

  Técnicas propostas:
  - ➢ Testes GCD e Ômega;
  - ➢ *Polyhedral methods*;
  - ➢ *Loop skewing*;
  - ➢ *Loop distribution*.

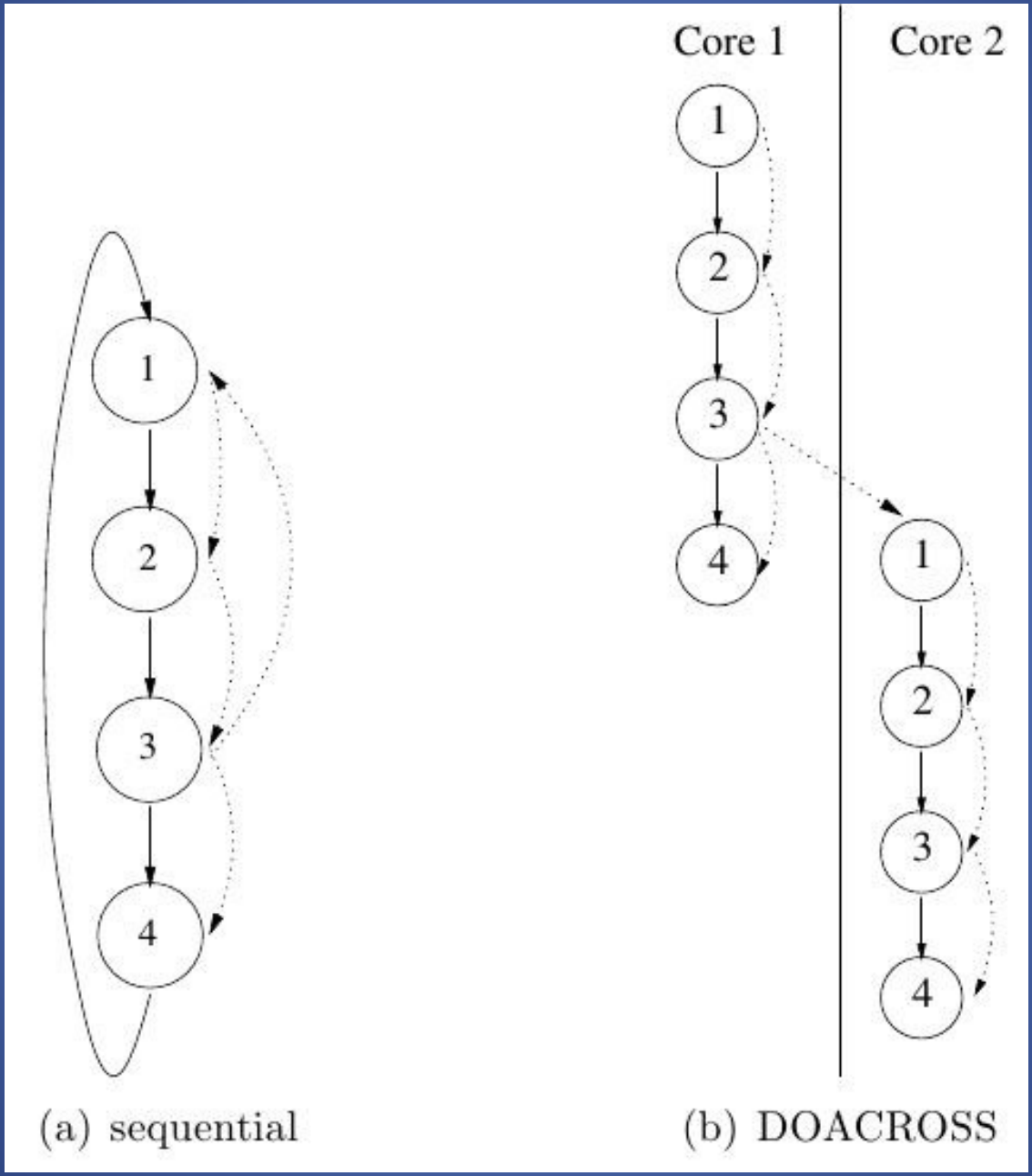
Exemplo: DOALL



Coarse-Grained TLP [2]

- ✓DOACROSS – laços com dependências de malha (*loop-carried dependence*)
  - ➢Iterações executadas na forma de rodízio (*round-robin fashion*)
  - ➢Primitivas de sincronização são inseridas no código para respeitar as dependências de malha
    - Limita a quantidade de paralelismo;
    - Insere sincronizações no caminho crítico de execução do laço.

Exemplo: DOACROSS



Fine-Grained TLP [1]

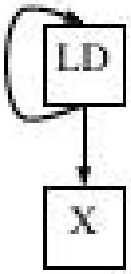
- ✓ *Scalar Operand Network (on-chip inter-core communication)*
  - ➢ Presente no Processador RAW
  - ➢ Malhas interligadas entre os *cores*
  - ➢ Co-processador de roteamento em cada *core*
  - ➢ Para o SW equivale a um conjunto de filas em HW com registros especializados de leitura e gravação (*register-mapped queues*)

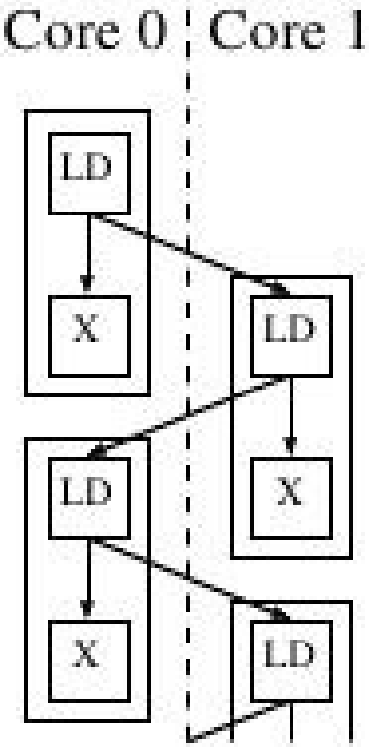
Fine-Grained TLP [2]

- *Synchronization Array*
  - ISA estendido com duas novas instruções:
    - ✓ *Produce & Consume*
  - Apoiam a paralelização de laços que percorrem estruturas de dados recursiva ou ligadas
  - Dividido em 2 segmentos:
    - ✓ Execução do percurso
    - ✓ Execução do corpo do laço
  - e.g. *Decoupled Software Pipelining* - DSWP

Exemplo: DSWP

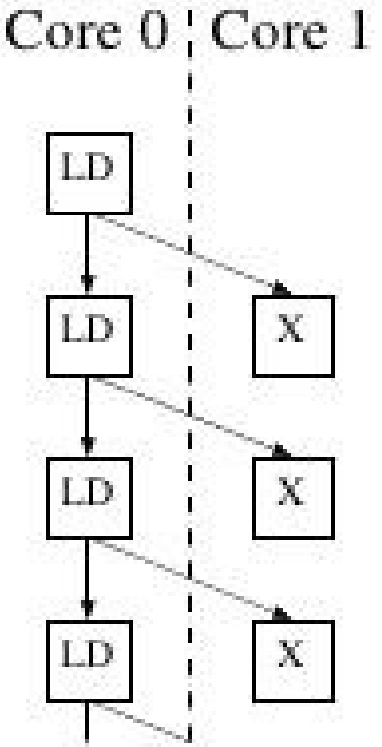
```
while (ptr = ptr->next) {
   ptr->val = ptr->val + 1; // X
}
```





Iters * (LD Latency + Comm Latency)

DOACROSS



Iters * LD Latency

DSWP

Paralelização Semi-Automática

O paralelismo completamente implícito é ainda um objetivo distante

Threads Explícitas

- ✓ Não-determinísticas por construção
- ✓ Sincronização via *locks*, messagens, STM ...

Semi-implícitas

- ✓ Situa-se entre o paralelismo implícito e a abordagem explícita;
- ✓ Determinística (a semântica permanece a mesma);
- ✓ "anotações" no código indicam quando o paralelismo é útil;
- ✓ Tem-se mostrado eficaz.

Exemplo: quicksort [1]

Primeiro, temos uma função normal (em Haskell) que classifica uma lista usando uma abordagem de divisão-e-conquista:

```
quicksort :: Ord a => [ a ] -> [ a ]
quicksort [ ]       = [ ]
quicksort ( p:xs ) = losort ++ p : hisort
    where
        losort = quicksort [ y | y <- xs, y < p ]
        hisort = quicksort [ y | y <- xs, y >= p ]
```

- ✓ escolhe um elemento da lista (pivô). Qualquer elemento serve como pivô, o primeiro é apenas o mais fácil para fazer o "pattern matching".
- ✓ Cria-se uma sub-lista de todos os elementos menores que o pivô, e recursivamente ordena esta sub-lista.
- ✓ Idem para os elementos maiores ou iguais ao pivô.
- ✓ Concatena as duas sub-listas já ordenadas.

Exemplo: quicksort [2]

Agora, o código transformado em código paralelo, usando as anotações par and seq de Haskell:

```
quicksort :: Ord a => [ a ] -> [ a ]
quicksort [ ]       = [ ]
quicksort ( p:xs ) = losort `par` hisort `seq` ( losort ++ p : hisort )
    where
        losort = quicksort [ y | y <- xs, y < p ]
        hisort = quicksort [ y | y <- xs, y >= p ]
```

O padrão aqui é: `x` par (y `seq` e). O efeito deste padrão é fazer com que x seja avaliada em paralelo com y. Quando a avaliação de x e y forem concluídas, prossegue-se a computação por meio da avaliação da expressão e. seq é então usada para controlar a ordem de avaliação.

Por que Linguagens de Programação Funcional?

- ✓ Não requerem construções especiais para representar paralelismo;
- ✓ Paralelismo está implícito no algorítimo;
- ✓ computações independentes podem ser realizadas concorrentemente;
- ✓ Sem a necessidade de mecanismos especiais para proteger os dados que são compartilhados entre tarefas concorrentes, mas…

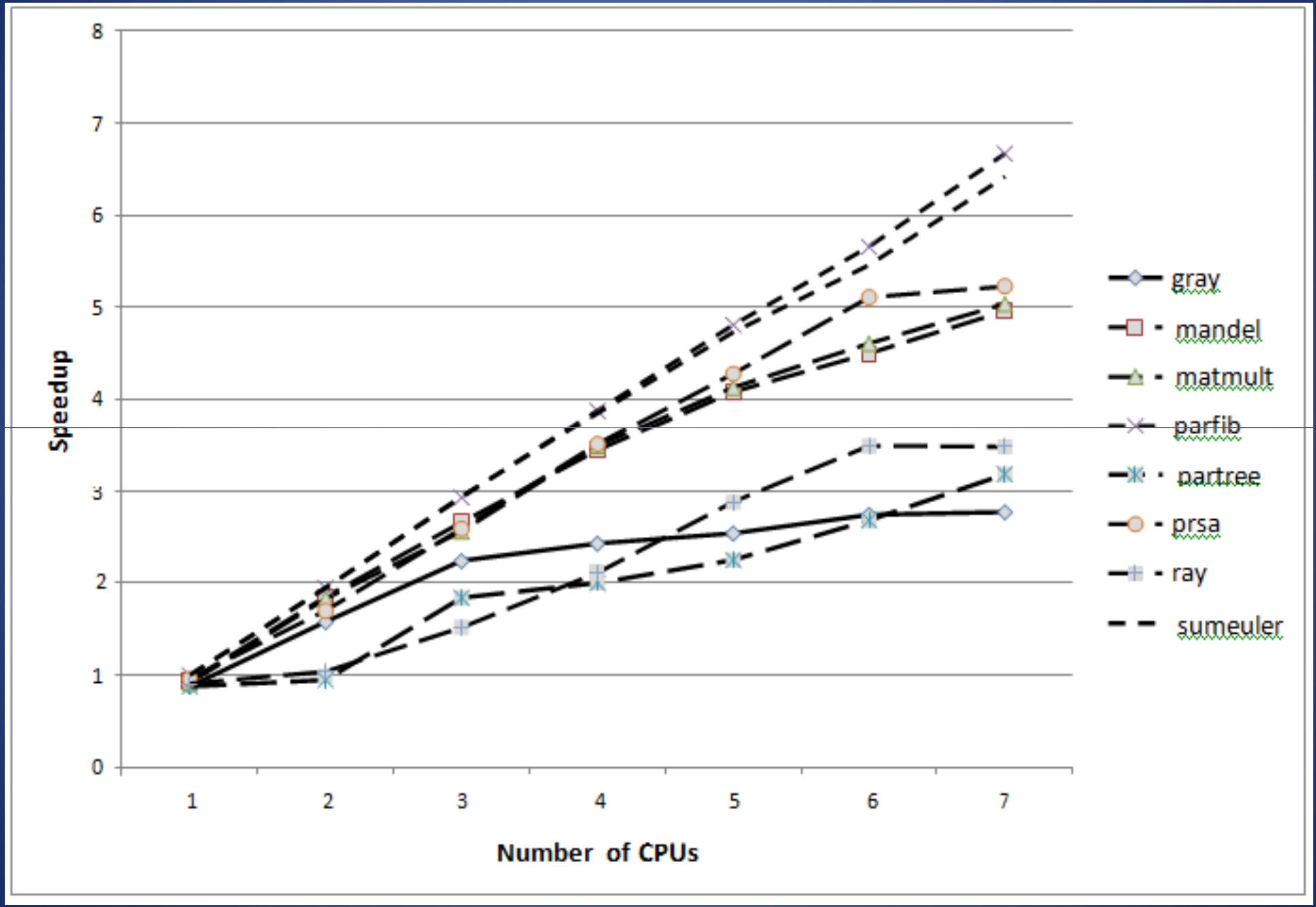
… difícil de realizar este potencial na prática.

Haskell is ...

- ✓ A purely functional language
- ✓ Strongly statically typed
- ✓ 20 years old
- ✓ Open source
- ✓ Used in research and industry
- ✓ Built for parallel programming

Haskell has a head start in the race to find an effective way to program parallel hardware.

Benchmark


Speedup
0
1
2
3
4
5
6
7
8
1
2
3
4
5
6
7
Number of CPUs
gray
mandel
matmult
parfib
partree
prsa
ray
sumeuler

Conclusão

✓ TLP não é um problema trivial:

- ➢ falta de abstrações para expressar o paralelismo no nível das linguagens de programação;
- ➢ falta de modelos de programação paralela de fácil uso;
- ➢ limitações impostas aos compiladores na paralelização automática;

✓O que está sendo feito (Software e Hardware)?

- ➢ (SW) Desenvolvimento de novas linguagens de programação que capturam de forma eficiente o paralelismo inerente no ínicio do ciclo de desenvolvimento (e.g., X10 da IBM, Fortress da Sun, Chapel da Cray).
- ➢ (HW) Especulação no nível de threads (TLS) e Memória Transacional (TM).

✓Nós vimos também que a utilização de programas puramente funcionais, como Haskell, são alternativas viáveis para resolver a programação paralela e para explorar a arquitetura *multicore.*

Obrigado!