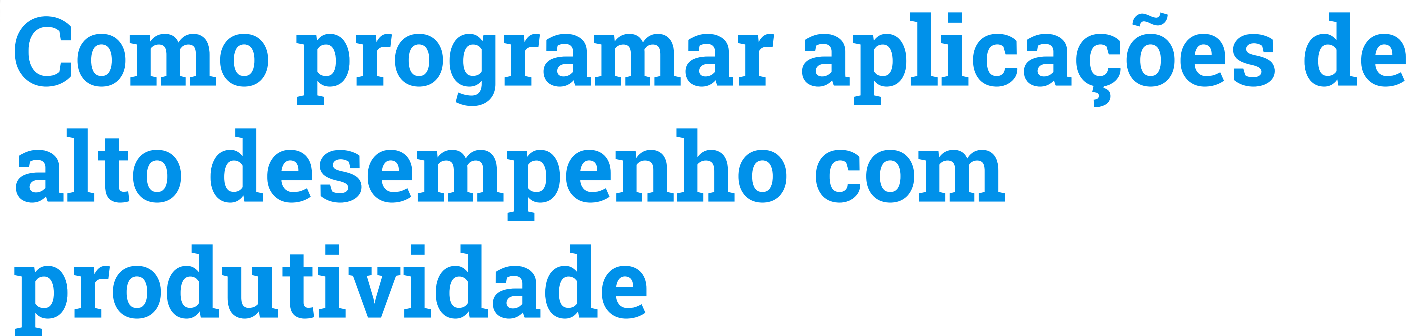


Como programar aplicações de alto desempenho com produtividade

Álvaro Fazenda e Denise Stringhini

CSBC 2019 - JAI 2

Sumário

- Introdução
- Modelo hierárquico de sistemas computacionais de alto desempenho
- Paralelização de aplicações
- Programação OpenMP
- Programação para GPUs

Objetivo

Apresentar princípios da programação paralela baseada em **diretivas**, aplicáveis a problemas encontrados em aplicações diversas.

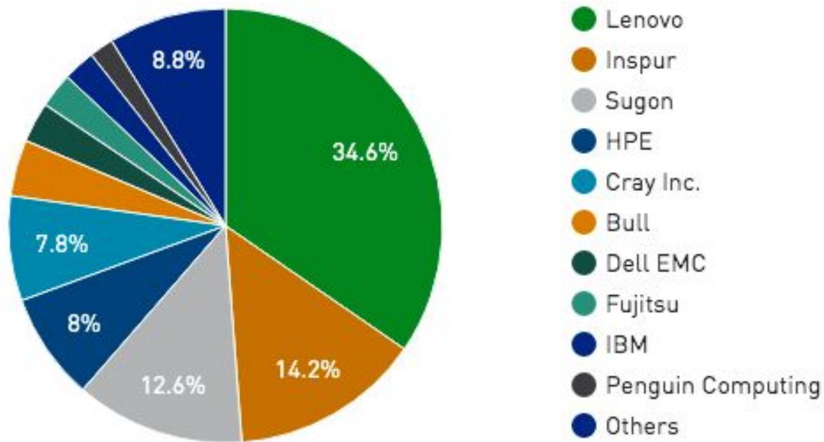
Introdução

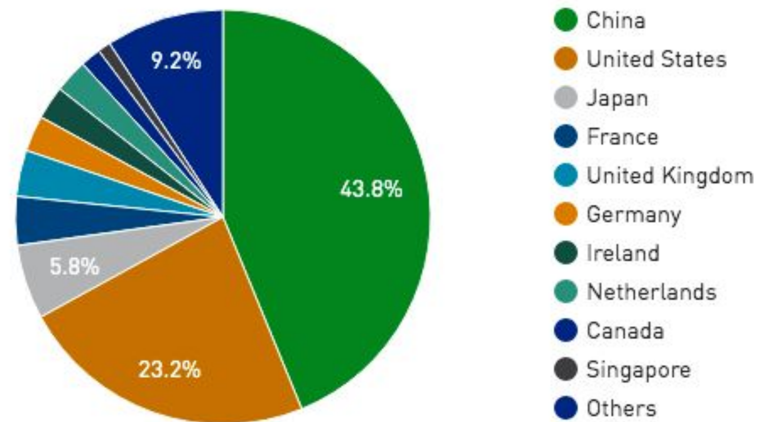
O que é alto desempenho?

Top500.org (junho/2019)

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	DOE/SC/Oak Ridge National Laboratory United States	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband IBM	2,414,592	148,600.0	200,794.9	10,096
2	DOE/NNSA/LLNL United States	**Sierra** - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband IBM / NVIDIA / Mellanox	1,572,480	94,640.0	125,712.0	7,438
3	National Supercomputing Center in Wuxi China	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	125,435.9	15,371
4	National Super Computer Center in Guangzhou China	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 NUDT	4,981,760	61,444.5	100,678.7	18,482
5	Texas Advanced Computing Center/Univ. of Texas United States	**Frontera** - Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR Dell EMC	448,448	23,516.4	38,745.9	

Top500.org - estatísticas (junho 2019)





Brasil no Top500

Brasileiros no Top 500 (junho 2016)

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
265	Laboratório Nacional de Computação Científica Brazil	**Santos Dumont GPU** - Bullx B710, Intel Xeon E5-2695v2 12C 2.4GHz, Infiniband FDR, Nvidia K40 Bull, Atos Group	10,692	456.8	657.5	
323	SENAI CIMATEC Brazil	**CIMATEC Yemoja** - SGI ICE X, Intel Xeon E5-2690v2 10C 3GHz, Infiniband FDR SGI	17,200	405.4	412.8	
364	Laboratório Nacional de Computação Científica Brazil	**Santos Dumont Hybrid** - Bullx B710, Intel Xeon E5-2695v2 12C 2.4GHz, Infiniband FDR, Intel Xeon Phi 7120P Bull, Atos Group	24,732	363.2	478.8	
433	Laboratório Nacional de Computação Científica Brazil	**Santos Dumont CPU** - Bullx B71x, Intel Xeon E5-2695v2 12C 2.4GHz, Infiniband FDR Bull, Atos Group	18,144	321.2	348.4	

Brasileiros no Top 500 (junho 2017)

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
471	**Santos Dumont GPU** - Bullx B710, Intel Xeon E5-2695v2 12C 2.4GHz, Infiniband FDR, Nvidia K40 , Bull, Atos Group Laboratório Nacional de Computação Científica Brazil	10,692	456.8	657.5	371.3
480	Cluster Platform DL360, Xeon E5-2673v3 12C 2.4GHz, 10G Ethernet , HPE Cloud Provider Brazil	17,136	450.2	658.0	571.2

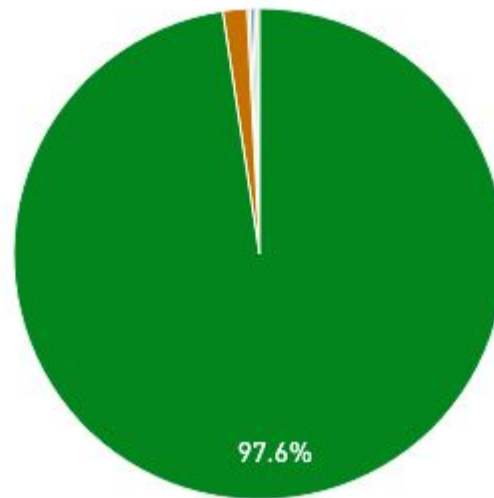
Brasileiros no Top 500 (novembro 2018)

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
331	**BC1** - Lenovo C1040, Xeon E5-2673v4 20C 2.3GHz, 40G Ethernet , Lenovo Cloud Provider Brazil	38,400	1,123.2	1,413.1	

Brasileiros no Top 500 (junho 2019)

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
142	**Fênix** - SYS-1029GQ-TRT, Xeon Gold 5122 4C 3.6GHz, Infiniband EDR, NVIDIA Tesla V100 , Bull, Atos Group Petróleo Brasileiro S.A Brazil	48,384	1,836.0	4,297.4	287
419	**BC1** - Lenovo C1040, Xeon E5-2673v4 20C 2.3GHz, 40G Ethernet , Lenovo Cloud Provider Brazil	38,400	1,123.2	1,413.1	
431	**BC2** - Lenovo C1040, Xeon E5-2673v4 20C 2.3GHz, 40G Ethernet , Lenovo Software Company (M) Brazil	38,400	1,123.2	1,413.1	

Aplicações – Top 500 (junho 2019)



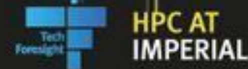
- Healthcare and Medicine
- Modeling and simulation
- Security
- Fintech and Engineering
- Materials and manufacturing industries

KEY

- MODELLING & SIMULATION
- HEALTHCARE & MEDICINE
- SECURITY
- FINTECH
- ENGINEERING, MATERIALS & MANUFACTURING

CURRENT & FUTURE APPLICATIONS OF HIGH PERFORMANCE COMPUTING

Conceived and created by Richard Watson and Jolanta Leonaite at Imperial Tech Foresight. Design execution by Wond.co.uk
Further information can be obtained from techforesight@imperial.ac.uk



https://toptrends.nowandnext.com/2017/02/22/roadmap-for-high-performance-computing-more-interesting-than-it-sounds/

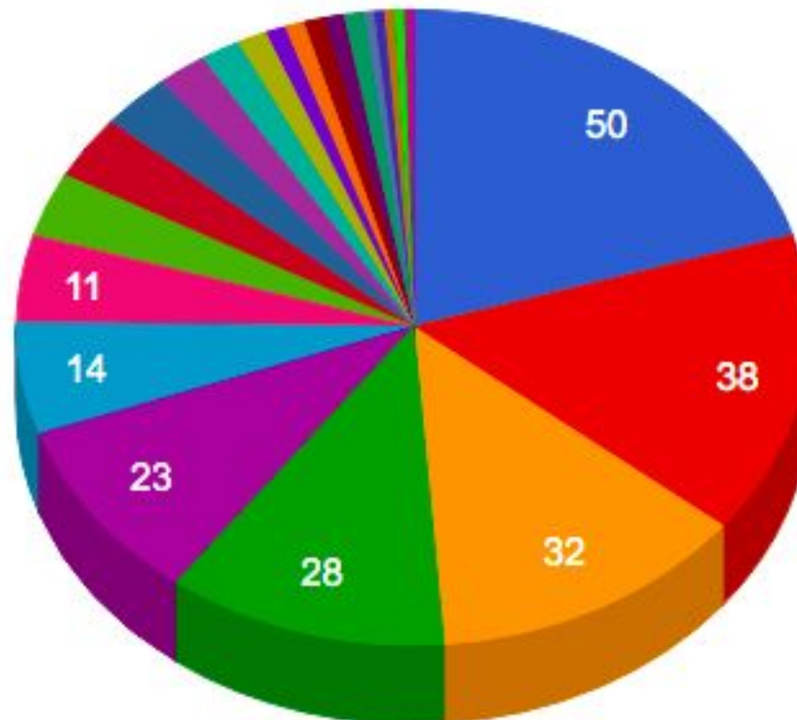
- 13 projetos científicos e tecnológicos em andamento
- 21 projetos encerrados
- Instituições de todas as regiões
- Diferentes áreas do conhecimento
- Engenharias, Física, Ciências Biológicas, Química, Computação, Meteorologia, Saúde, outras

Projetos em andamento:

http://sdumont.lncc.br/projects_ongoing.php?pg=projects#

SINAPAD - Áreas do conhecimento

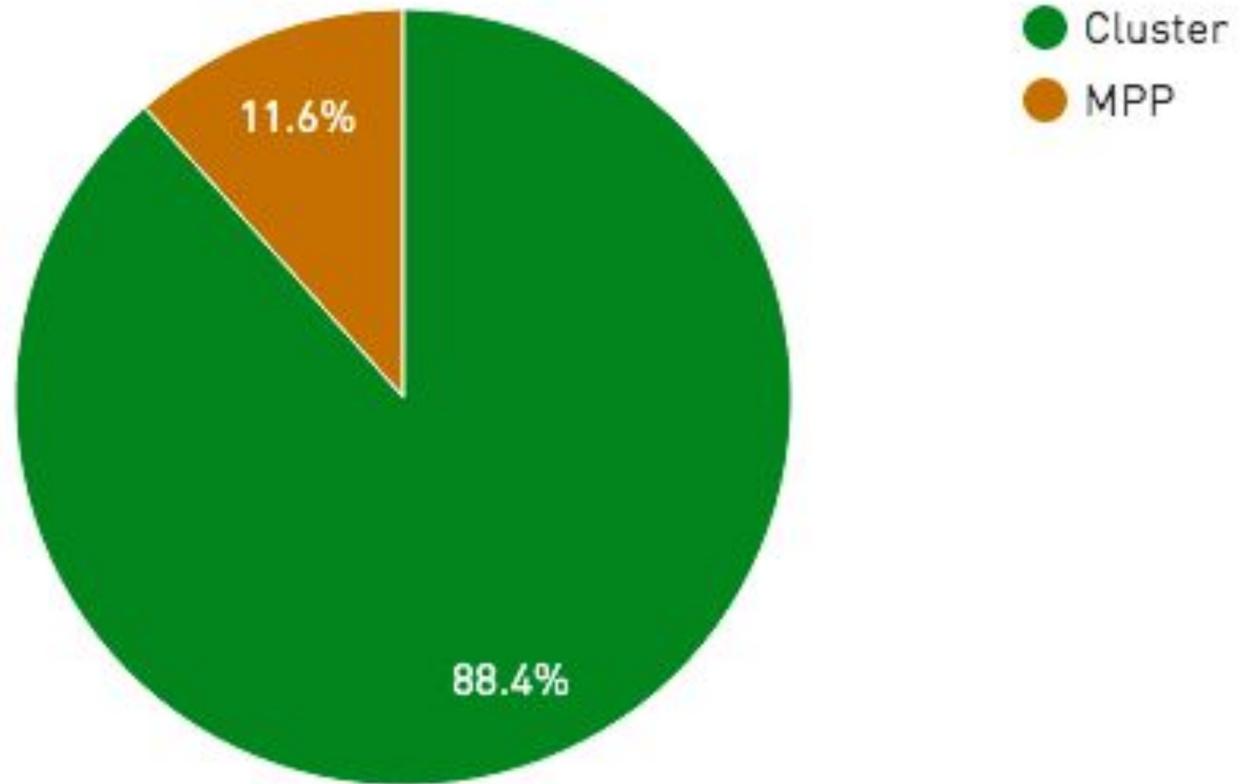
Química · Física · Engenharias · Ciências biológicas · Ciência da computação
Ciências da saúde · Geociências · Astronomia · Matemática · Meteorologia
Ciência dos materiais · Biodiversidade · Probabilidade e estatística · Oceanografia
Linguística, letras e artes · Farmácia · Ciências agrárias · Economia · Educação
Ciências humanas · Bioinformática · Administração · Astroquímica



Modelo hierárquico de SCAD

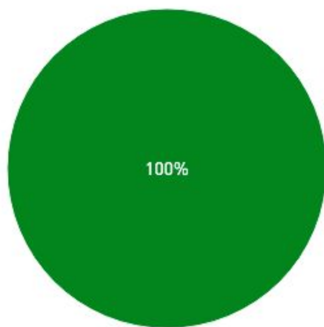
Considerações sobre arquiteturas de sistemas computacionais de alto desempenho

Arquitetura – Top 500



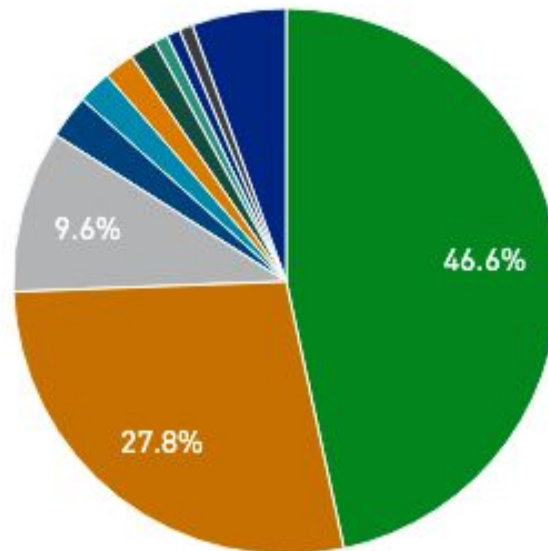
Sistema operacional – Top 500

Operating system Family System Share



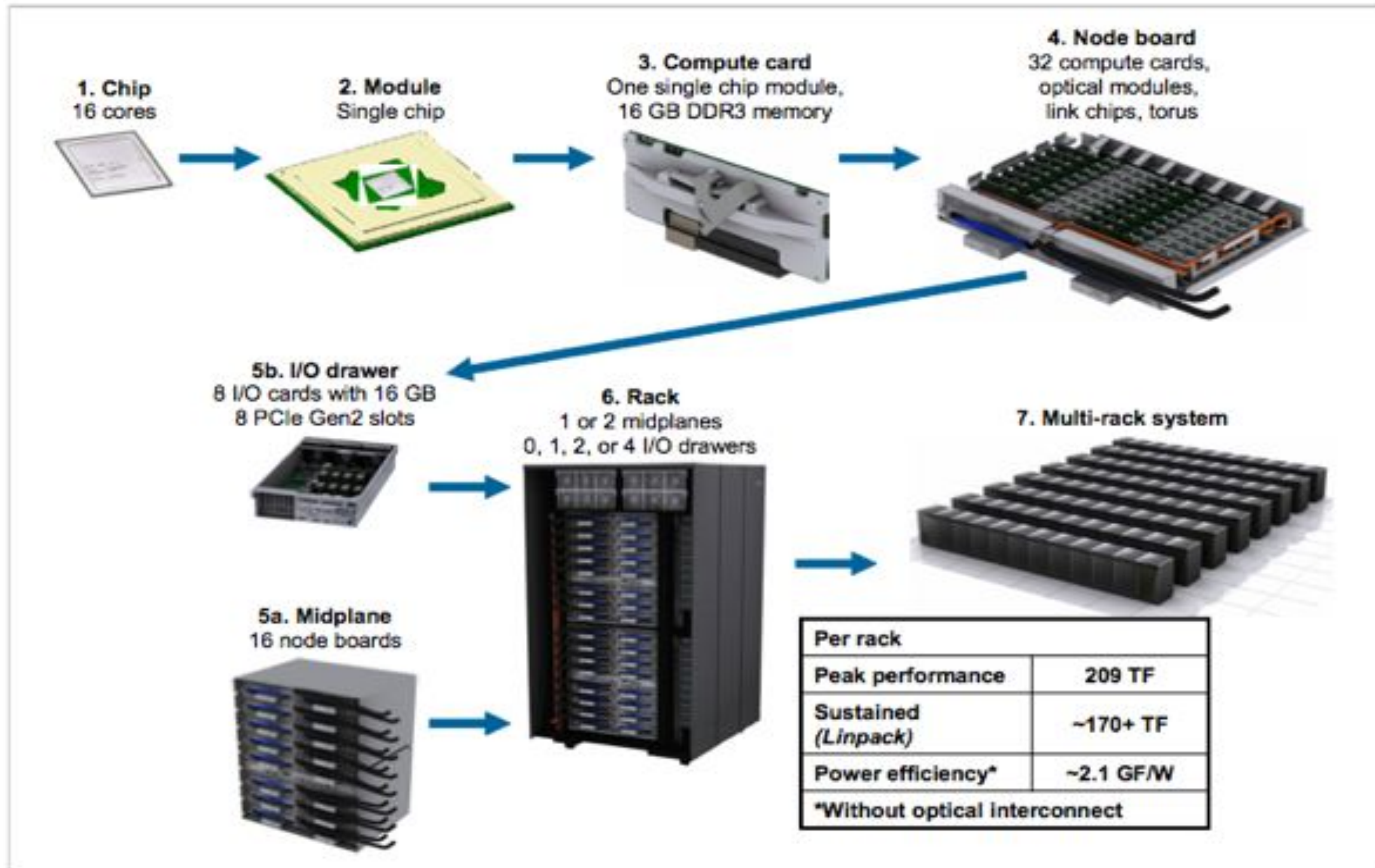
Linux

Operating System System Share



- Linux
- CentOS
- Cray Linux Environment
- bullx SCS
- SUSE Linux Enterprise Se...
- TOSS
- Red Hat Enterprise Linux
- RHEL 7.4
- RHEL 7.2
- Ubuntu Linux
- Others

Exemplo: IBM Blue Gene



Per rack	
Peak performance	209 TF
Sustained *(Linpack)*	~170+ TF
Power efficiency*	~2.1 GF/W
*Without optical interconnect	

Figure 1-2 Blue Gene/Q hardware overview

Memória compartilhada

- Multicore
- Programação: variáveis compartilhadas entre *threads*.
  - OpenMP, Pthreads

Aceleradores

- GPU (ou outros aceleradores)
- Programação: *offloading* do código e dados para a placa
  - CUDA, OpenACC

Memória distribuída

- Cluster, MPP
- Programação: troca de mensagens entre processos.
  - **MPI**, PVM

MPP

3. Paralelização de aplicações

Técnicas básicas de projeto e extração de paralelismo considerando o paradigma de memória compartilhada

Quatro passos da paralelização

- Análise da versão sequencial
- Projeto e implementação
- Testes de correção
- Análise de desempenho

Análise da versão sequencial

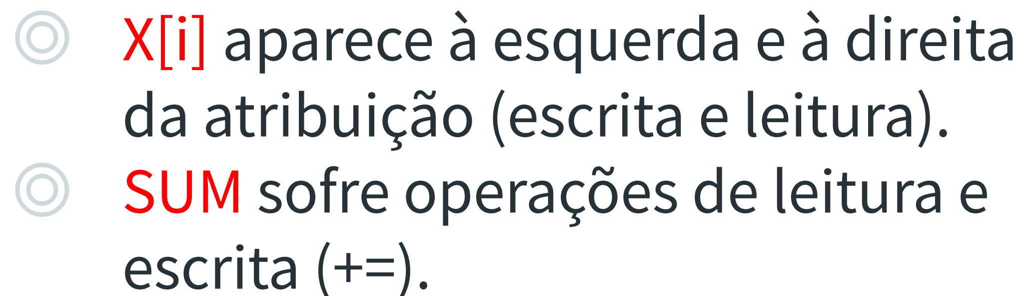
1. Identificar pontos onde se possa implementar concorrência.
2. Definir tipo de decomposição de domínio.
3. Encontrar *hot spots*.
   - Normalmente laços.

(1) Análise de dependências

```
for(i=0; i<N; i++) {
    X[i] = Y[i] + Z[i];
    A[i] = Y[i] + delta;

}
```

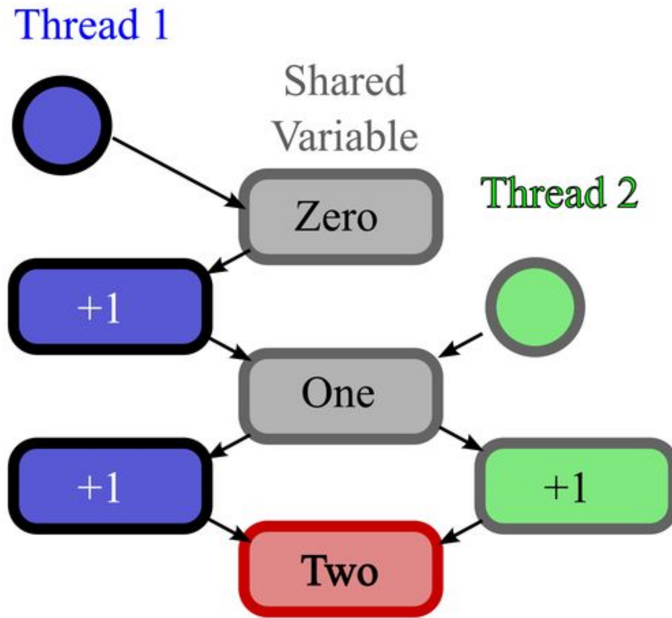
```
SUM = 0;
for(i=0; i<N; i++) {
    X[i] = X[i] + Z[i];
    A[i] = X[i] + delta;
    SUM += A[i];
}
```

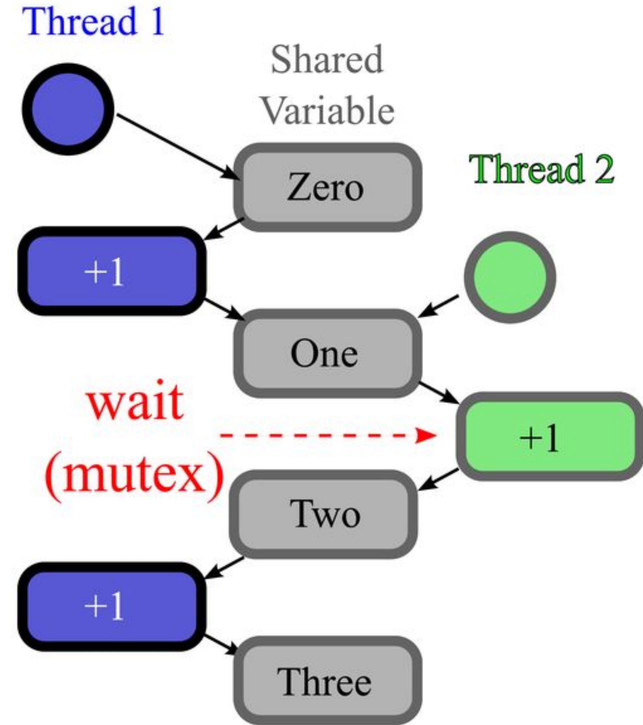
Algoritmo 2.1 - Exemplo de laços sem e com dependência de dados



- X[i] aparece à esquerda e à direita da atribuição (escrita e leitura).
- SUM sofre operações de leitura e escrita (+=).

(1) Análise de dependências: condições de corrida





http://www.techenablement.com/intel-xeon-phi-optimization-part-1-of-3-multi-threading-and-parallel-reduction/

(2) Decomposição de domínio

- Decomposição funcional
  - Dividir o trabalho entre tipos de tarefas (ou funções) diferentes
- Decomposição de domínio
  - Dividir os dados entre tarefas que executam o mesmo código

(3) Detecção de *hot spots*

- Detecção de trechos mais demorados (“gargalos”)
- Laços são bons candidatos
  - Tarefas repetitivas sobre um conjunto de dados homogêneos (decomposição de domínio)
- Escolhe-se os laços candidatos e efetua-se a medição de tempo.

(3) Detecção de *hot spots*

```
#include <stdio.h>
#include <sys/time.h>
#define N 1000000

int main(void) {
   int k;
   double p = 1, x;
   struct timeval inicio, final;
   long long tmili;

   gettimeofday(&inicio, NULL);
   x = 1.0 + 1.0/N;
   for(k=0; k<N; k++)
      p = p*x;
   gettimeofday(&final, NULL);

   tmili = (int) (1000*(final.tv_sec - inicio.tv_sec) +
           (final.tv_usec - inicio.tv_usec) / 1000);

   printf("PI aproximado: %g\n", p);
   printf("tempo decorrido: %lld ms\n",tmili);
   return 0; }
```

Algoritmo 2.2 - Exemplo de código-fonte com instrumentação para medida de tempo de execução de um trecho específico.

Projeto e implementação

1. Observar plataforma de hardware disponível (arquitetura paralela)
2. Considerar uso de padrões conhecidos de programação paralela
3. Realizar testes de correção
4. Realizar uma análise de desempenho

- Observar plataforma de hardware disponível e definir paradigmas de programação:
  - Memória compartilhada
  - Aceleradores
  - Memória distribuída

Padrões de programação

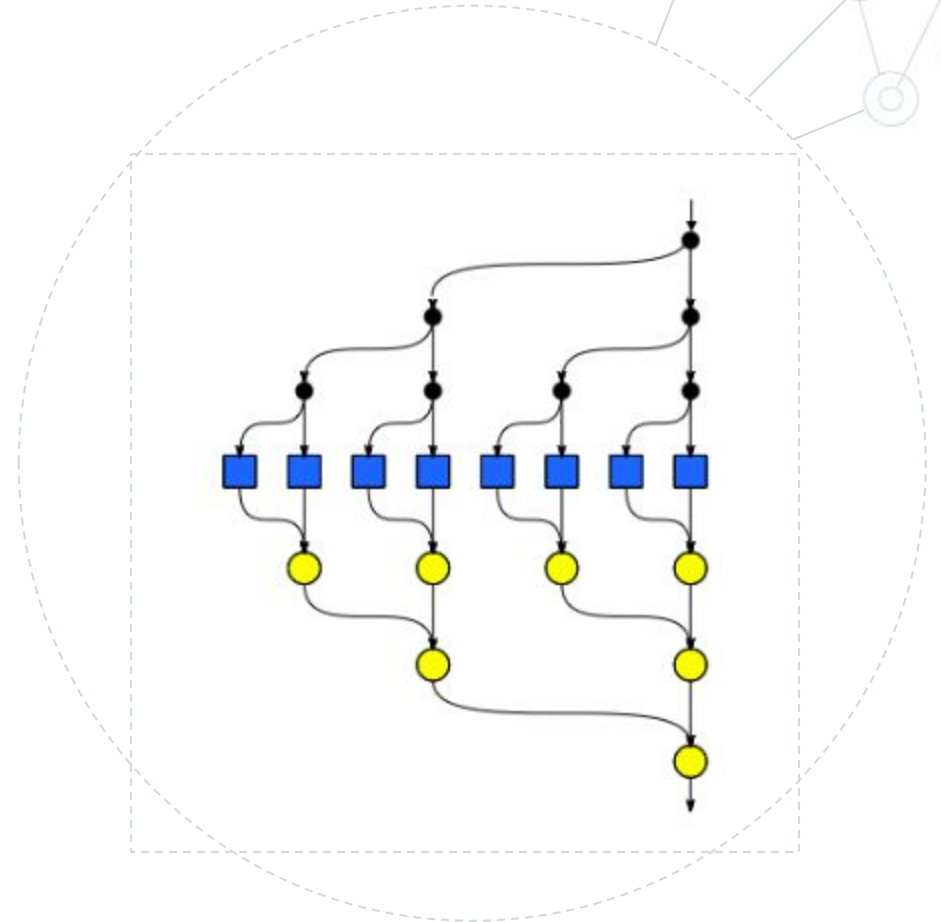
Padrões de código são soluções genéricas e reutilizáveis para problemas comuns em determinados contextos.

- O projeto de código paralelo pode usar algum padrão de código conhecido para este fim
- Exemplo: McCool et al (2012).

(2.1) Padrão fork-join

A operação ***fork*** permite que o fluxo de controle seja dividido em múltiplos fluxos de execução paralelos

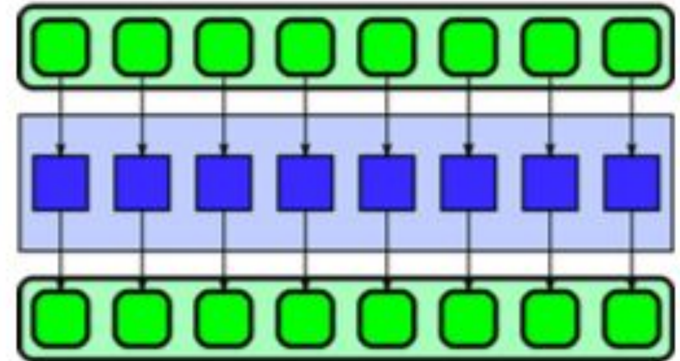
Na operação ***join*** estes fluxos se reunirão novamente no final de suas execuções, quando apenas um deles continuará.



(2.2) Padrão map

Replica uma mesma operação sobre um conjunto de elementos indexados.

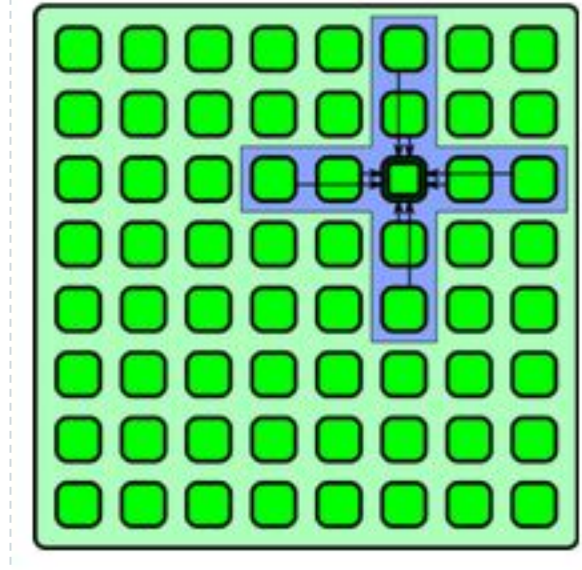
Este padrão se aplica à paralelização de laços, nos casos onde se pode aplicar uma função independente a todo o conjunto de elementos.



(2.3) Padrão stencil

É uma generalização do padrão map, onde a função é aplicada sobre um conjunto de vizinhos.

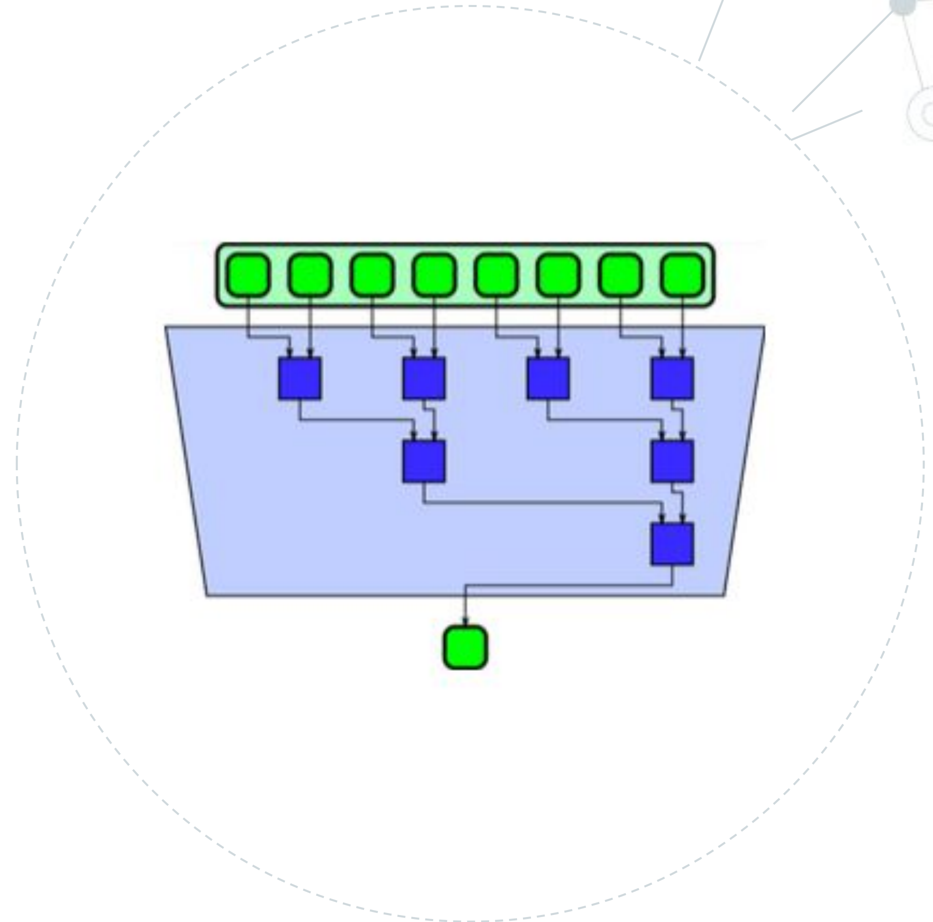
Os vizinhos são definidos a partir de um conjunto offset relativo a cada ponto do conjunto.



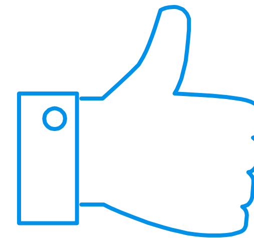
(2.4) Padrão reduction

Combina todos os elementos de uma coleção em um único elemento a partir de uma função combinadora associativa.

Uma operação muito comum em aplicações numéricas é realizar um somatório ou encontrar o máximo de um conjunto de elementos.



Testes de correção



Atenção a aspectos da execução paralela que se diferencia da sequencial e podem ocasionar erros ou inconsistências de execução.

(3) Testes de correção

- Código multithreading é altamente sujeito a erros e não-determinismo
- Atenção a:
  - Erros causados pela alteração concorrente de dados compartilhados,
    - Ex: condições de corrida (*race conditions*).
  - Mau-uso dos mecanismos de controle de acesso à seções críticas (por exemplo, mutex)
  - Erros de sincronização entre threads.

(3) Testes de correção

- Existem ferramentas de depuração disponíveis
  - Maioria não é gratuita
  - Uso requer certo treinamento
- Teste básico
  - Comparar saída da versão paralela com saída de uma versão sequencial correta
  - Problema: não determinismo (executar várias vezes…)
- Recomenda-se testes sistemáticos principalmente em problemas críticos.

Análise de desempenho

É necessário que o desempenho paralelo seja melhor que o sequencial, pois o custo e a complexidade deste processo devem valer à pena.

(4) Análise de desempenho

- Medidas de tempo e comparações com a versão sequencial do programa são usadas para verificar se houve ganho de desempenho.
- Caso contrário, o programador deverá verificar pontos de gargalo que estejam atrapalhando o desempenho.

(4) Análise de desempenho: problemas frequentes

- Situações onde haja contenção na sincronização de recursos compartilhados (por exemplo, uso sincronizado de variáveis compartilhadas),
- Desbalanceamento de carga de trabalho entre as threads
- Quantidade excessiva de chamadas à biblioteca de threads

(4) Análise de desempenho: objetivos

Desempenho

Capacidade de reduzir o tempo de resolução do problema à medida que os recursos computacionais aumentam

Escalabilidade

Capacidade de aumentar ou manter o desempenho à medida que os recursos computacionais aumentam

(4) Análise de desempenho: fatores limitantes

Limites arquiteturais

Latência e a largura de banda da camada de interconexão.

Capacidade de memória da máquina utilizada.

Limites algorítmicos

Falta de paralelismo inerente ao algoritmo.

Frequência de comunicação.

Frequência de sincronização.

Sistema de execução

Escalonamento deficiente.

Balanceamento de carga.

(4) Análise de desempenho: medidas

Medida básica: Tempo de Execução

O sistema B é n vezes mais rápido que A quando:

$$Texec(A) / Texec(B) = n$$

Maior desempenho → Menor tempo de execução

(4) Análise de desempenho: speedup

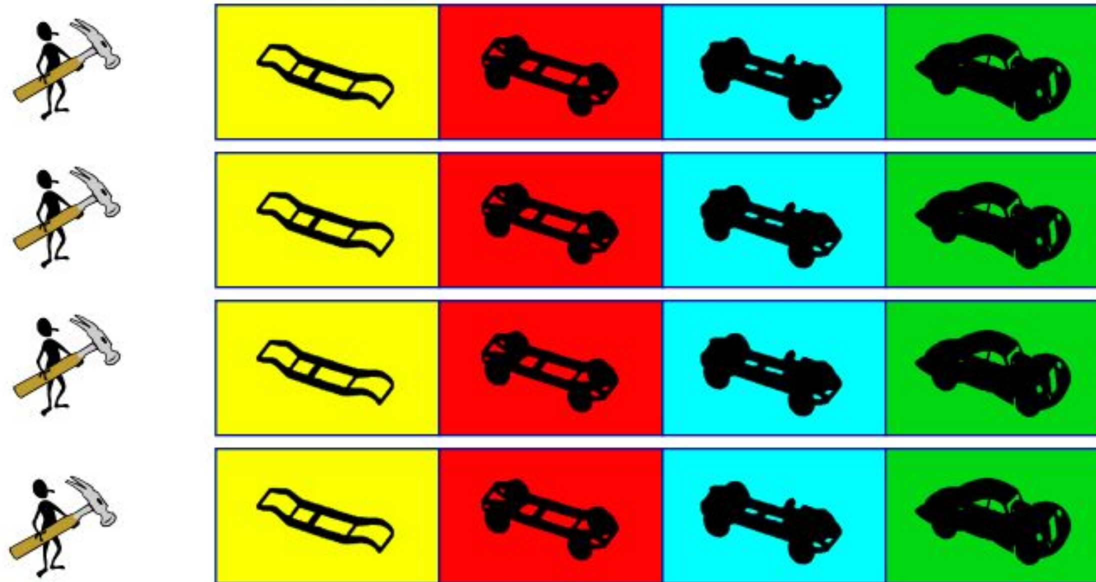
Speedup: Medida de ganho em tempo

Speedup(P) = T (1 proc) / T (P proc)

Onde P = número de processadores

$1 \leq Speedup \leq P$

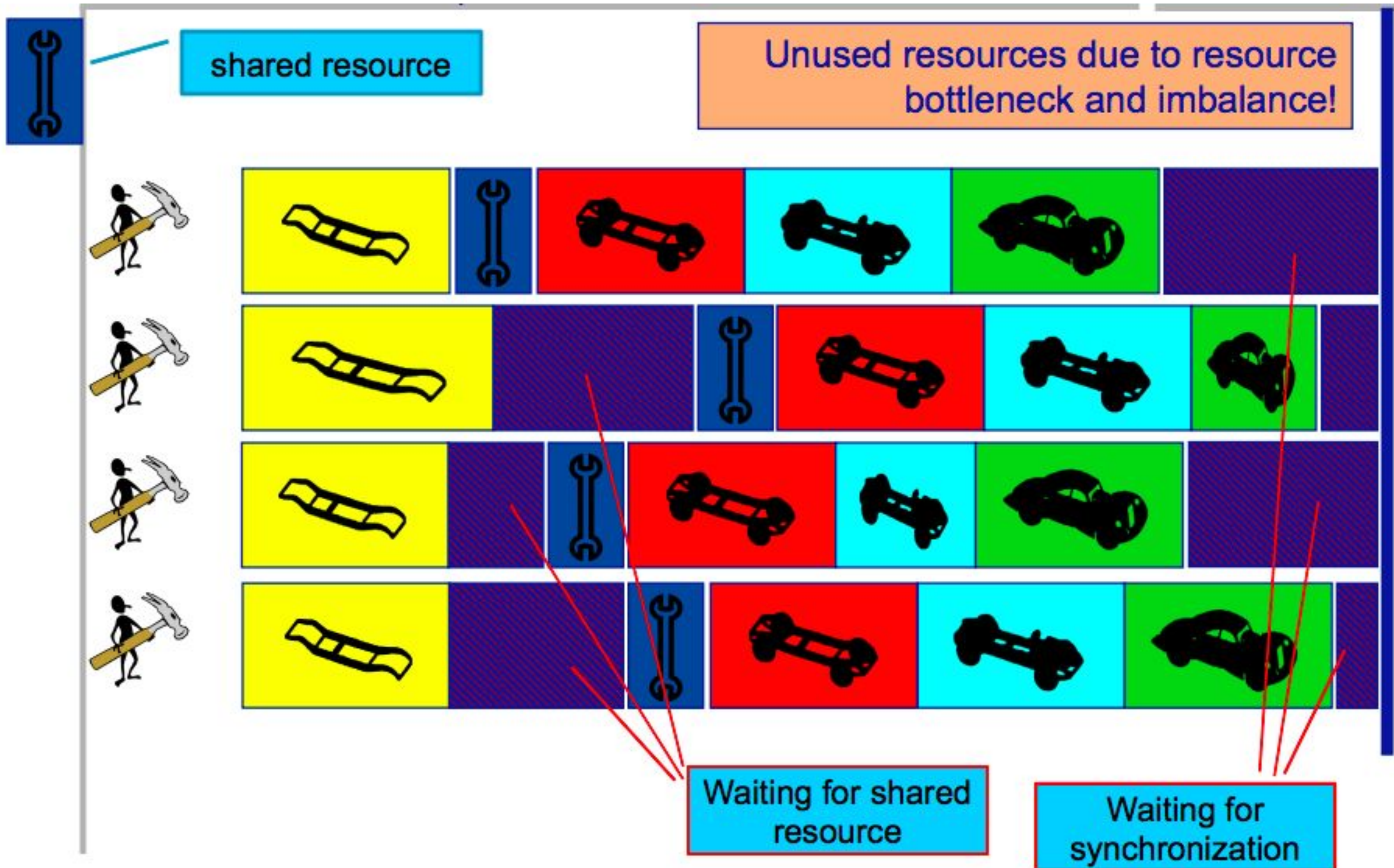
(4) Análise de desempenho: speedup



Após 4 passos: 4 carros

"speedup perfeito" = speedup linear

(4) Análise de desempenho: speedup



Eficiência: Medida de uso dos processadores

Eficiência(P) = Speedup(P) / P

$0 < \text{Eficiência} \leq 1$

(4) Análise de desempenho: Lei de Amdhal

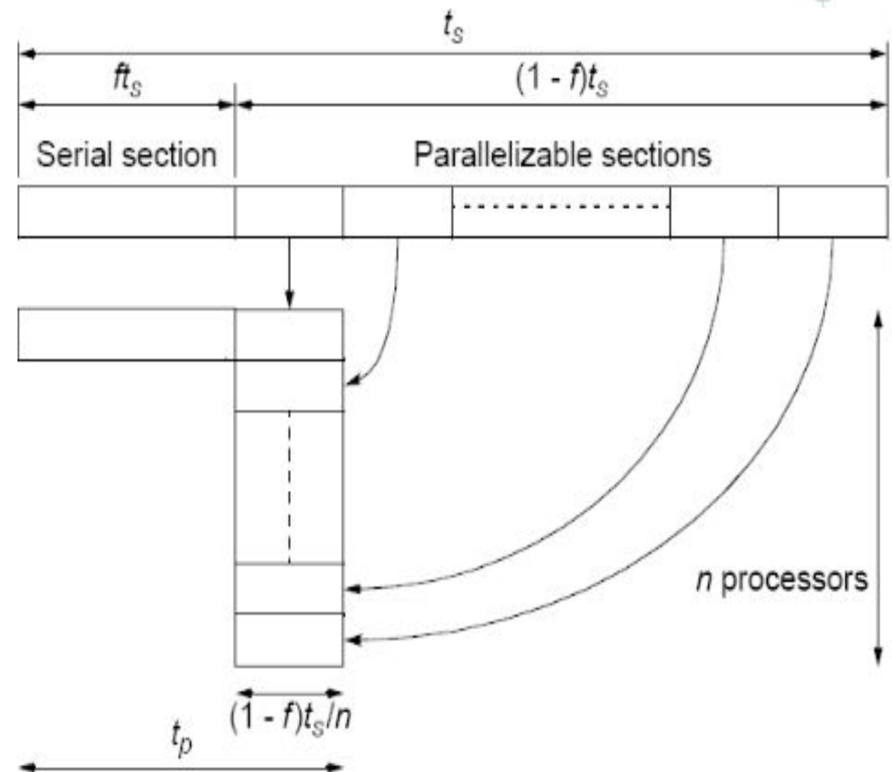
- Limitação teórica para os ganhos de desempenho

Programa serial: $Tserial = T0 = (s+q)T0$

- Onde:
  - $s+q = 1$
  - s corresponde a fração serial do código (impossível de ser paralelizada)
  - q corresponde a fração paralelizavel do código

(4) Análise de desempenho: Lei de Amdhal

- Supondo paralelização ideal:
  - $Tpar = sT0 + (q/p)T0$
- $Speedup = Tserial/Tpar = (s+q)/(s+q/P) = 1 / (s+q/P)$
- $Speedup = 1 / [s+(1-s)/P]$



(4) Análise de desempenho: Lei de Amdhal

- **Lei de Amdhal**: considera apenas a escalabilidade dos recursos computacionais, sendo o tamanho do problema fixo.
  - Neste caso, temos limitação do speedup.
- **Lei de Gustafson-Barsis**: considera o aumento do tamanho da aplicação ou do domínio a medida que se aumentam o número de recursos computacionais.

https://www.youtube.com/watch?v=NaUqvKFj4Oo

(4) Análise de desempenho: tipos de escalabilidade

Escalabilidade forte

Mantém-se o tamanho do problema e escala-se o número de processadores.

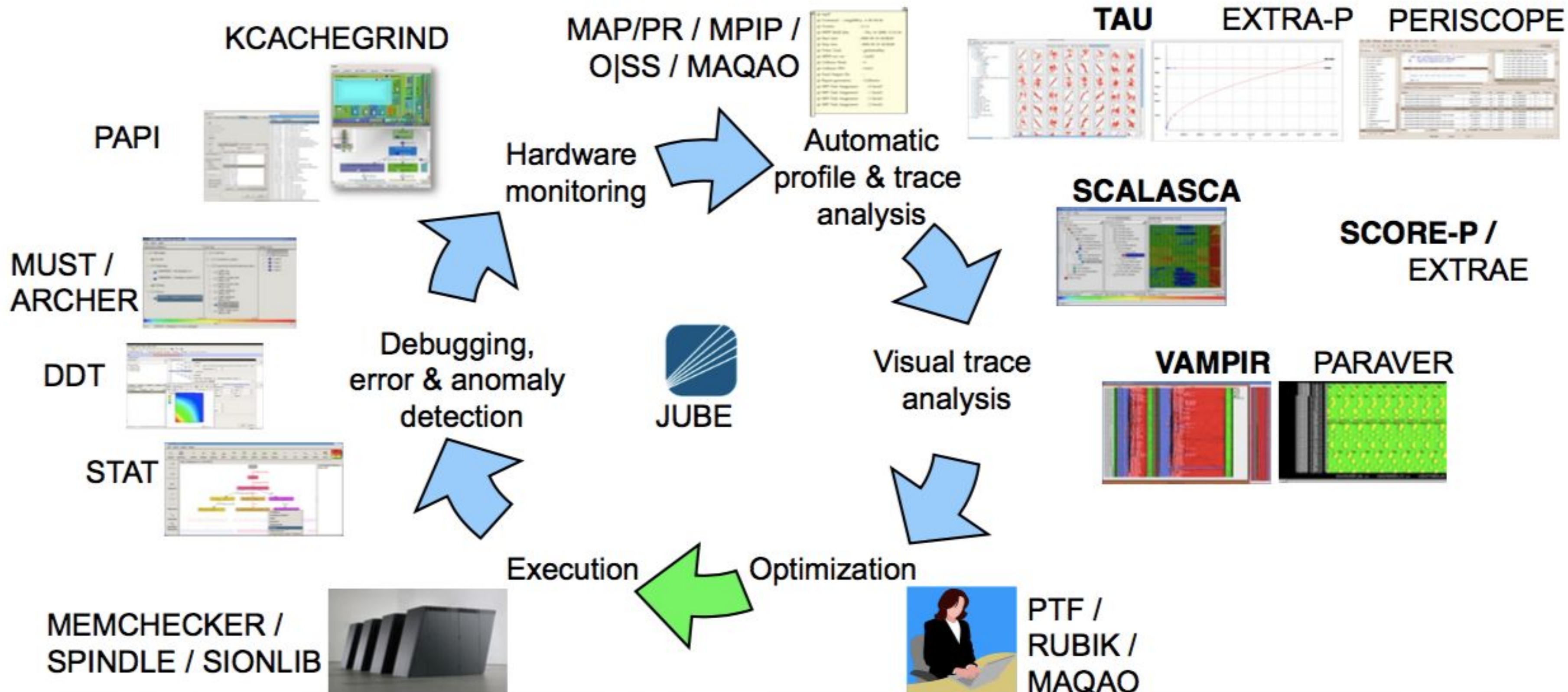
É a capacidade de executar aplicações n vezes mais rápidas, onde n é a quantidade de processadores utilizados (speedup).

Escalabilidade fraca

Escala-se o tamanho do problema juntamente com o número de processadores.

É a capacidade de aumentar a carga de trabalho e a quantidade de processadores por um fator de n e manter o tempo de computação.

(4) Análise de desempenho: ferramentas e ciclo de desenvolvimento



Bugs and Speed in HPC Applications: Past, Present, and Future
Jeffrey Hollingsworth - ISC 2018

Programação OpenMP

Programação por diretivas

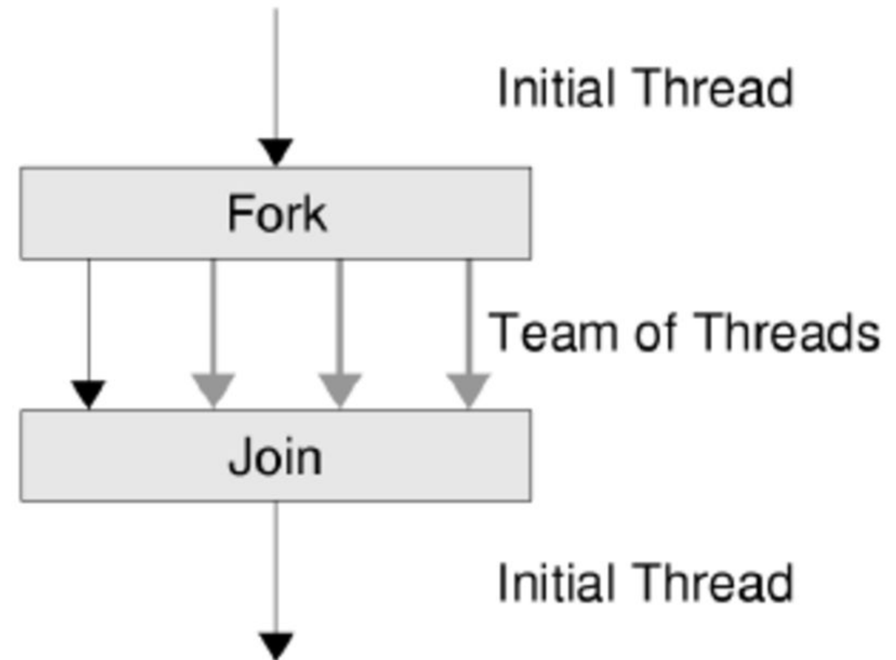
Análise

- Concorrência: paralelismo em potencial
- Identificação dos possíveis pontos onde se possa implementar concorrência
- Identificar hot-spots
  - Trechos de alta demanda computacional
  - Comum em laços (loops)

OpenMP
Enabling HPC since 1997

- Padrão
  - Diretivas
  - pequeno conjunto de funções
  - variáveis de ambiente
- suporta C/C++ e Fortran
  - C/C++: **`#pragma`**
- Várias opções de compiladores
  - GCC/GFortran

Modelo *Fork-Join*



Modo de operação do OpenMP

- Uma **região paralela** é a estrutura básica do paralelismo OpenMP
- Inicia a execução com uma única *thread*
  - Denominada ***thread master***
- **Criação automática de *threads*** em regiões paralelas
  - *Overhead* para criação e destruição
  - Importante **minimizar o número de regiões paralelas** abertas
- Existência de **variáveis privadas ou compartilhadas** nessas regiões

Diretivas e Sentinelas

- Uma diretiva é uma linha especial de código fonte com significado especial apenas para determinados compiladores
- Uma diretiva se distingue pela existência de uma sentinela no começo da linha
    - C/C++: **`#pragma omp`**
    - Seguem o padrão de diretivas de compilação para C/C++
    - Cada diretiva se aplica no máximo a próxima instrução, que deve ser um bloco estruturado

Regiões Paralelas

- Um código dentro da região paralela é executado por todas as *threads*
    - Exige incluir a biblioteca: `<omp.h>`

```
#include <omp.h>
...
#pragma omp parallel
   {
   // bloco executado em paralelo
   }
```

Cláusulas

- Especificam informação adicional na diretiva de região paralela:
- Em C/C++:

```
#pragma omp parallel [clausulas]
```

- Cláusulas são separadas por vírgula ou espaço no Fortran, e por espaço no C/C++

Principais cláusulas para atributos de dados em uma região paralela

- **`shared(list)`** - Variáveis compartilhadas - todas as *threads* acessam o mesmo endereço dos dados
- **`private(list)`** - Variáveis privadas - cada *thread* tem a sua própria cópia local
  - Valores são indefinidos na entrada e saída
- **`default(shared|none)`** - Define valores *default*:
  - `Shared` - todas os dados são compartilhados
  - `None` - nada será definido por *default*

OpenMP Team := Master + Workers

- Uma região paralela é um bloco de código executado por todas as *threads* concorrentemente
  - A *thread* Mestre tem `ID = 0`
  - Ao iniciar uma região paralela todos os *threads* estão sincronizados
  - Regiões paralelas podem ser aninhadas, mas esta característica é dependente da implementação
  - Regiões paralelas podem ser condicionadas por "`if`"

Principais funções da API OpenMP

Função	Utilidade
omp_get_num_threads()	Retorna o número de threads em execução
omp_set_num_threads(n)	Define o número de threads (n)
omp_get_thread_num()	Retorna o ID da Thread
omp_get_num_procs()	Retorna o número de processadores do sistema
omp_in_parallel()	Retorna verificação se no dado ponto do programa está-se ou não em uma região paralela
omp_get_wtime()	Retorna o *Wall Clock Time* do sistema
omp_get_wtick()	Retorna o número de *ticks* ou intervalos regularmente espaçados ditados pela freqüência do *clock* entre um segundo, no sistema

Hello World em OpenMP

```
#include <omp.h>
#include <stdio.h>
int main (int argc, char *argv[]) {
  int th_id, nthreads;
#pragma omp parallel private(th_id)
  {
    th_id = omp_get_thread_num();
    printf("Hello World from thread %d\n", th_id);
    if (th_id==0) {
      nthreads = omp_get_num_threads();
      printf("There are %d threads\n", nthreads); }
  }
  Return 0; }
```

Definindo a quantidade de *threads* a executar

- Padrão: quantidade de threads = quantidade de processadores

- Dentro do código:
  - `#pragma omp parallel num_threads(N)`
  - `omp_set_num_threads(N)`

- Fora do código com variável de ambiente:
  - `export OMP_NUM_THREADS=N`

Compilação de programas em C/C++ com OpenMP

- Compiladores disponíveis: Intel C/C++ e Fortran-95, GCC e GFORTRAN, G95, PGI, PathScale, Absoft

- Usando GCC:

```
gcc -fopenmp -o <executavel> <fonte.c>
```

Laços Paralelos em OpenMP

- Principal fonte de paralelismo em muitas aplicações
  - Checar se as Iterações são independentes
    - podem ser executadas em qualquer ordem
  - Exemplo: considerando 2 *threads* e o laço a a seguir, pode-se fazer as iterações 0-49 em uma *thread* e as iterações 50-99 na outra

```
for (i = 0; i<100; i++) {
    a[i] = a[i] + b[i]; }
```

Laços Paralelos em OpenMP (cont.)

- `#pragma omp for [clausulas]`

- Sem cláusulas adicionais, a diretiva DO/FOR particionará as iterações o mais igualmente possível entre as *threads*
  - Contudo, isto é dependente de implementação e ainda há alguma ambigüidade:
  - Ex:. 7 iterações, 3 *threads*. Pode ser particionado como 3+3+1 ou 3+2+2

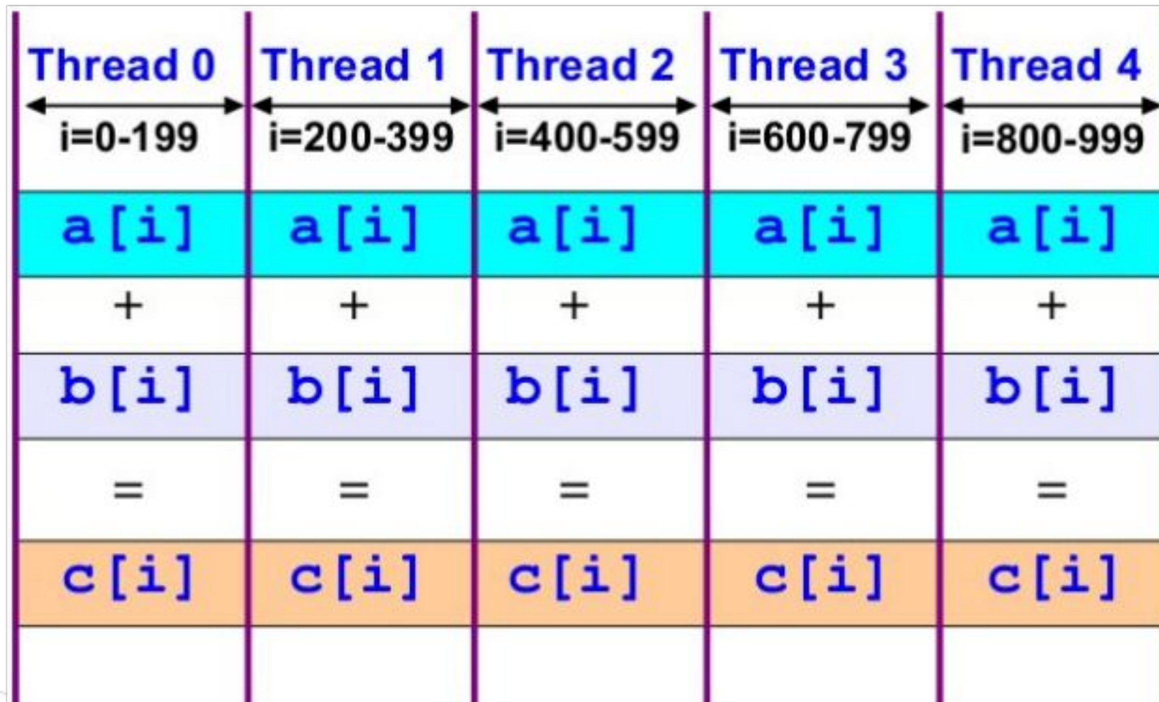
Exemplos de laços paralelos em OpenMP

For-loop with independent iterations

```
for (int i=0; i<n; i++)
  c[i] = a[i] + b[i];
```

For-loop parallelized using an OpenMP pragma

```
#pragma omp parallel for
for (int i=0; i<n; i++)
    c[i] = a[i] + b[i];
```



Paralelizando laço em OpenMP

```
/* Laco perfeitamente paralelizavel */
#pragma omp parallel
#pragma omp for
for (i=0; i<n; i++) {
    b[i]= (a[i] - a[i-1)]])*0.5; }
/* end parallel for */
```

A diretiva DO/FOR paralela

- Construção comum
- Existe uma forma que combina a região paralela e a diretiva DO/FOR:

```
#pragma omp parallel for [clausulas]
```

```
#pragma omp parallel
#pragma omp for
   for (...)
```

```
#pragma omp parallel for
for (....)
```

Single PARALLEL loop

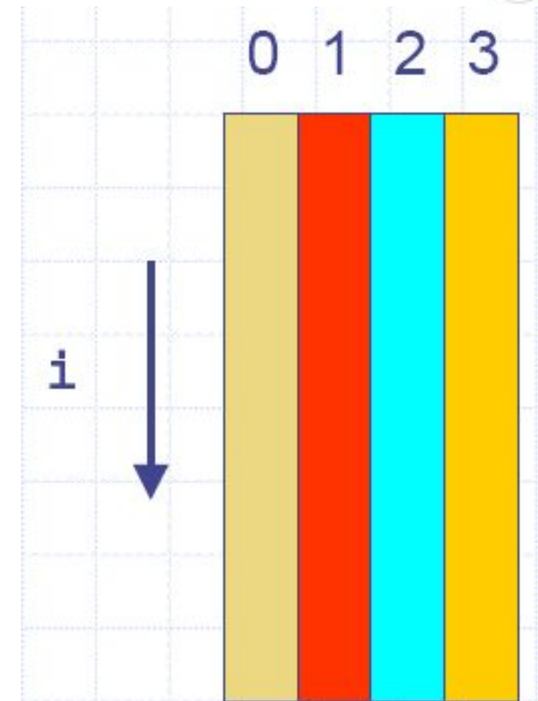
Mais exemplos de uso da diretiva DO/FOR

```
#pragma omp parallel for
for (i=0; i<n; i++) {
    b[i]= (a[i] - a[i-1)])*0.5; }
/* end parallel for */
```

```
#pragma omp parallel for
for (i=0; i < n; i++) {
    z[i] = a * x[i] + y; }
```

Variáveis Privadas e Compartilhadas (exemplo)

```
#pragma omp parallel \
    default(none) \
    private (i, myid) \
    shared(a,n)
{
myid = omp_get_thread_num();
for(i = 0; i < n; i++){
   a[i][myid] = 1.0; }
} /* end parallel */
```



Quais variáveis devem ser compartilhadas e privadas?

- A maioria das variáveis são compartilhadas
  - *Defaults*:
    - O índices dos laços são privados
    - Variáveis temporárias dos laços são compartilhadas
    - Variáveis apenas de leitura são compartilhadas
    - Arrays principais são compartilhadas
    - Escalares do tipo *write-before-read* são usualmente privados
  - A decisão pode ser baseada em fatores de desempenho

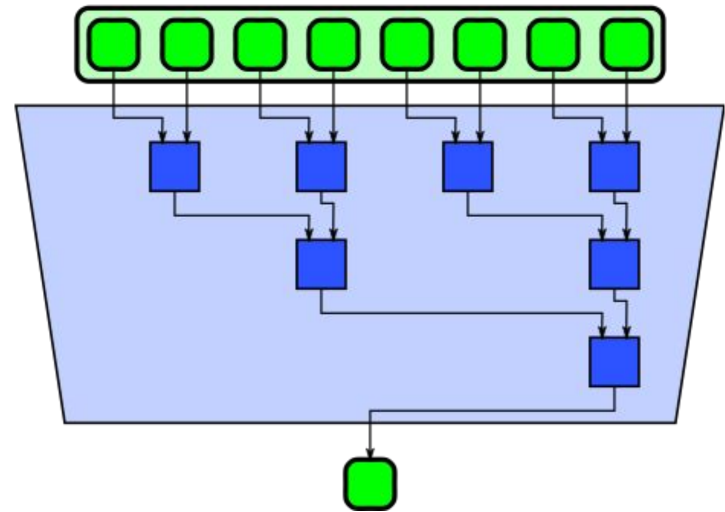
Valor inicial de variáveis privadas

- Variáveis privadas não tem valor inicial no início da região paralela.
- Para levar o valor atual para o valor inicial dentro da região paralela usa-se: **FIRSTPRIVATE**

```
b = 23.0;
. . . . .
#pragma omp parallel firstprivate(b) private(i,myid)
{
  myid = omp_get_thread_num();
  #pragma omp for
  for (i=0; i<n; i++) {
      b += c[myid][i]; }
  c[myid][n] = b;
}
```

Reduções

- combina todos os elementos de uma coleção em um único elemento a partir de uma função combinadora associativa
- Exemplos: somatório, produtório, média, máximo, mínimo, etc.



- Uso da cláusula REDUCTION:
  - C/C++: **`reduction(op:list)`**
    - Onde op pode ser:

<table>
<tr><th>Operador (op)</th><th>Operação</th><th>Valor inicial</th></tr>
<tr><td>+</td><td>adição</td><td>0</td></tr>
<tr><td>-</td><td>subtração</td><td>0</td></tr>
<tr><td>*</td><td>Multiplicação</td><td>1</td></tr>
<tr><td>&</td><td>E lógico</td><td>Todos os bits em 1</td></tr>
<tr><td>|</td><td>OU lógico</td><td>Todos os bits em 0</td></tr>
<tr><td>^</td><td>Equivalente (lógica)</td><td>Todos os bits em 0</td></tr>
<tr><td>&&</td><td>Não equivalente (lógico)</td><td>Todos os bits em 1</td></tr>
<tr><td>||</td><td>Máximo</td><td>0</td></tr>
</table>

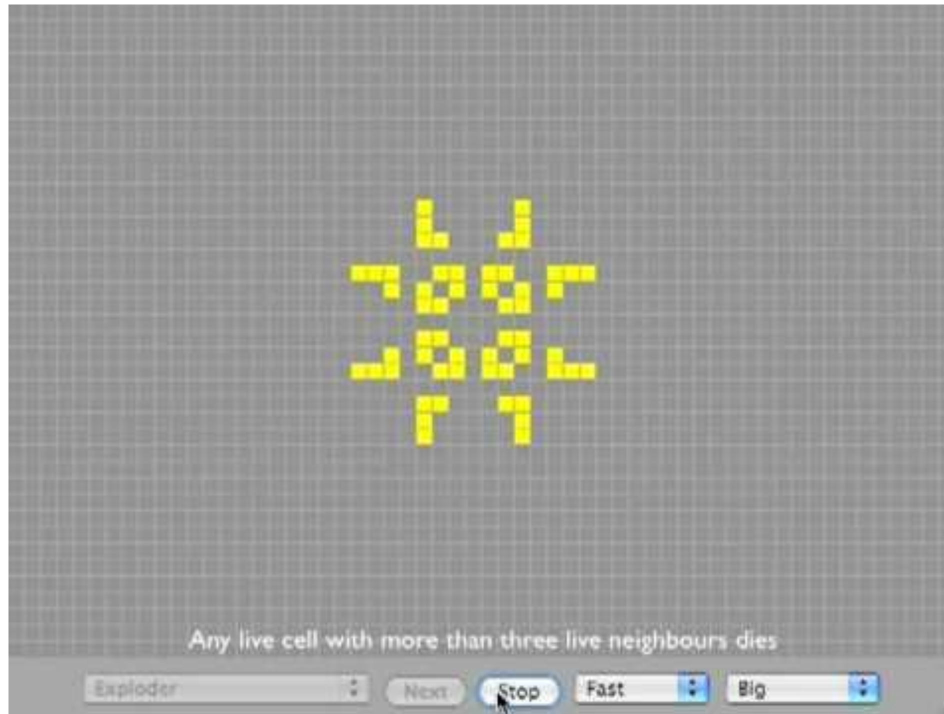
Exemplo de redução em OpenMP

```
soma = 0;
#pragma omp parallel private (i, myid) \
        reduction (+:soma)
{
myid = omp_get_thread_num();
#pragma omp for
for (i = 0; i < n; i++) {
    soma = soma + c[i][myid]; }
}
```

Estudo de Caso: **John Conway's Game of Life**

GOL

- Princeton - 1960 - John Conway
- jogo em tabuleiro similar ao xadrez/damas
- conjunto de regras pré-definido
  - formas geométricas bidimensionais
  - replicação autonoma
  - Modificação de forma a cada evolução
-



Regras:

1. Cada célula viva com menos de dois vizinhos morre de solidão;
2. Cada célula viva com dois ou três vizinhos deve permanecer viva para a próxima geração;
3. Cada célula viva com quatro ou mais vizinhos morre por superpopulação.
4. Cada célula morta com exatamente 3 vizinhos deve se tornar viva.

Código para evolução das células - GOL (Fonte: OLCF/ORNL)

```
for (i = 1; i<=dim; i++) {
    for (j = 1; j<=dim; j++) {
        int numNeighbors = getNeighbors(grid, i, j);
        if (grid[i][j]==1 && numNeighbors<2)        //R1
            newGrid[i][j] = 0;
        else if (grid[i][j]==1 &&                    //R2
                    (numNeighbors==2 || numNeighbors==3))
            newGrid[i][j] = 1;
        else if (grid[i][j]==1 && numNeighbors>3)   //R3
            newGrid[i][j] = 0;
        else if (grid[i][j]==0 && numNeighbors == 3)//R4
            newGrid[i][j] = 1;
        else
            newGrid[i][j] = grid[i][j];
        }
}
```

Repetir laço principal até limite de iterações ou atingir critério de parada

```
for (iter = 0; iter<maxIter; iter++) {
    // calcular nova geração do GOL
    ...
}
```

- Abrange todo o procedimento do slide anterior

Implementação da versão concorrente do GOL em OpenMP

- Tabuleiro bidimensional 2048x2048 células (4.194.304 células)
  - Condições de contorno periódica
    - Colunas e linhas de contorno
    - Face esquerda ligada a face oposta (direita)
    - Face superior ligada a face inferior
- 10.000 iterações (10 mil novas gerações)
- Tempo de execução: ~ 5 minutos e 30 segundos (330,474 segundos)
  - *C PGI community edition compiler*

GOL OpenMP - Versão 1

```
for (iter = 0; iter<maxIter; iter++) {
    // Laco para nova geracao
    #pragma omp parallel for
     for (i = 1; i<=dim; i++) {
         for (j = 1; j<=dim; j++) {
             ... }
         }
      ... // restante do codigo }
```

Idem para os laços que tratam as condições de contorno

Somatório das células do tabuleiro ao final

```
int total = 0;
   #pragma omp parallel for \
                  reduction(+:total)
for (i = 1; i<=dim; i++) {
   for (j = 1; j<=dim; j++) {
      total += grid[i][j];
   }
}
```

Desempenho da versão 1 em OpenMP (dual Intel Xeon E5-2660v4 @ 2.00GHz, onde cada CPU conta com 14 núcleos (cores))

Nº threads	Tempo(s)	*Speedup*	Eficiência
Serial	330,474	1	100%
2	256,37	~1,289	~64,453%
4	132,531	~2,494	~62,339%
8	70,377	~4,696	~58,697%
16	39.945	~8,273	~51,708%
28 (max)	27.365	~12,076	~43,130%

Problemas com eficiência paralela em OpenMP

- Causas diversas. Investigar:
  - Operações sequenciais (Lei de Amdhal)
    - alocação, iniciação de memória, controle do laço principal = 0,02%
  - **Comunicações e tarefas de sincronização**
    - **abertura e fechamento de threads de3 regiões paralelas ocorrem 10.000*3 vezes**
    - **medida de tempo exata difícil de ser estimada**
  - Desbalanceamento de carga
    - tarefas concorrentes homogêneas

Otimizando: minimizando a quantidade de *threads* criadas e destruídas dentro do laço que controla as gerações

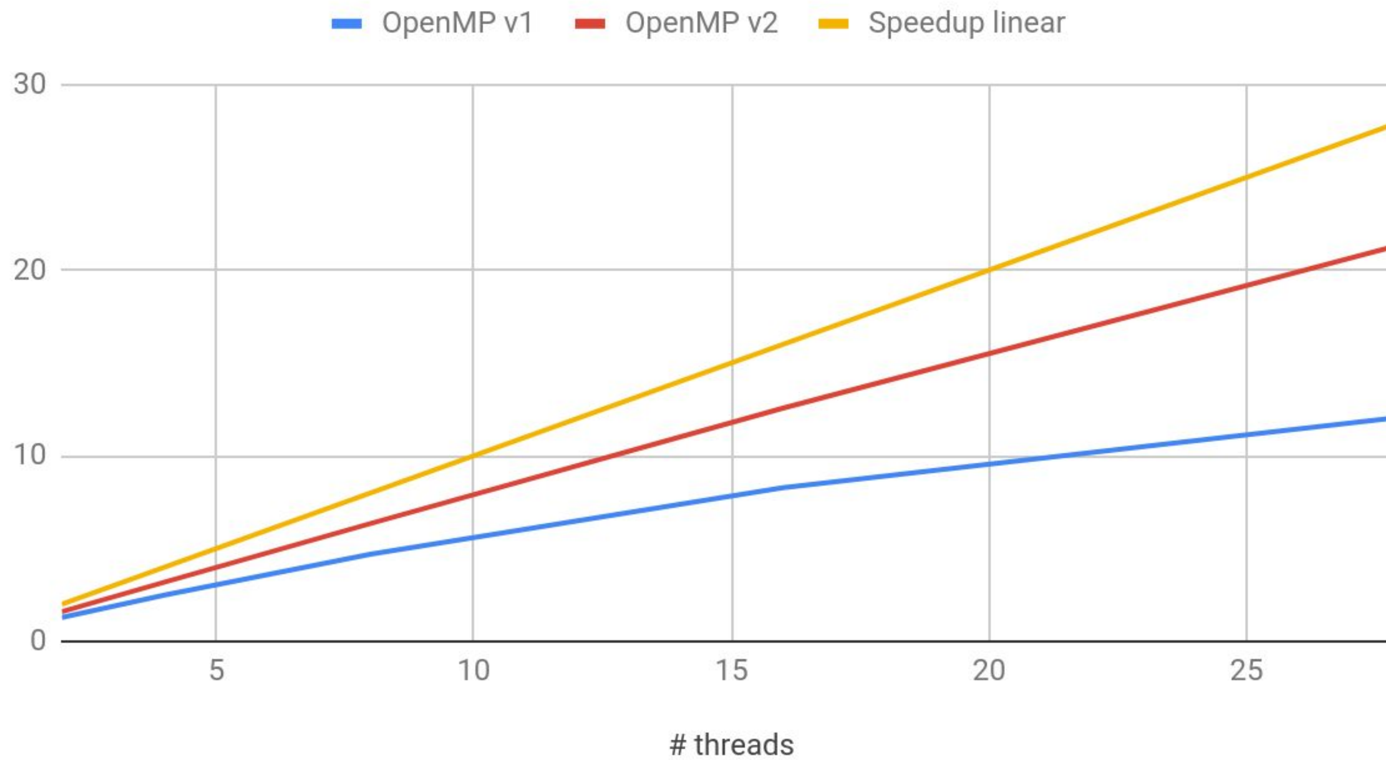
```
// loop principal - evolucao de geracoes
#pragma omp parallel \
    shared(grid,newGrid,maxIter,dim) private(iter,i,j)
{ // principal regiao paralela
for (iter = 0; iter<maxIter; iter++) {
     // Laco para nova geracao
     #pragma omp for
     for (i = 1; i<=dim; i++) {
           for (j = 1; j<=dim; j++) {
                 ... }
           }
    // restante do codigo
} // fim da região paralela
```

Threads criadas aqui

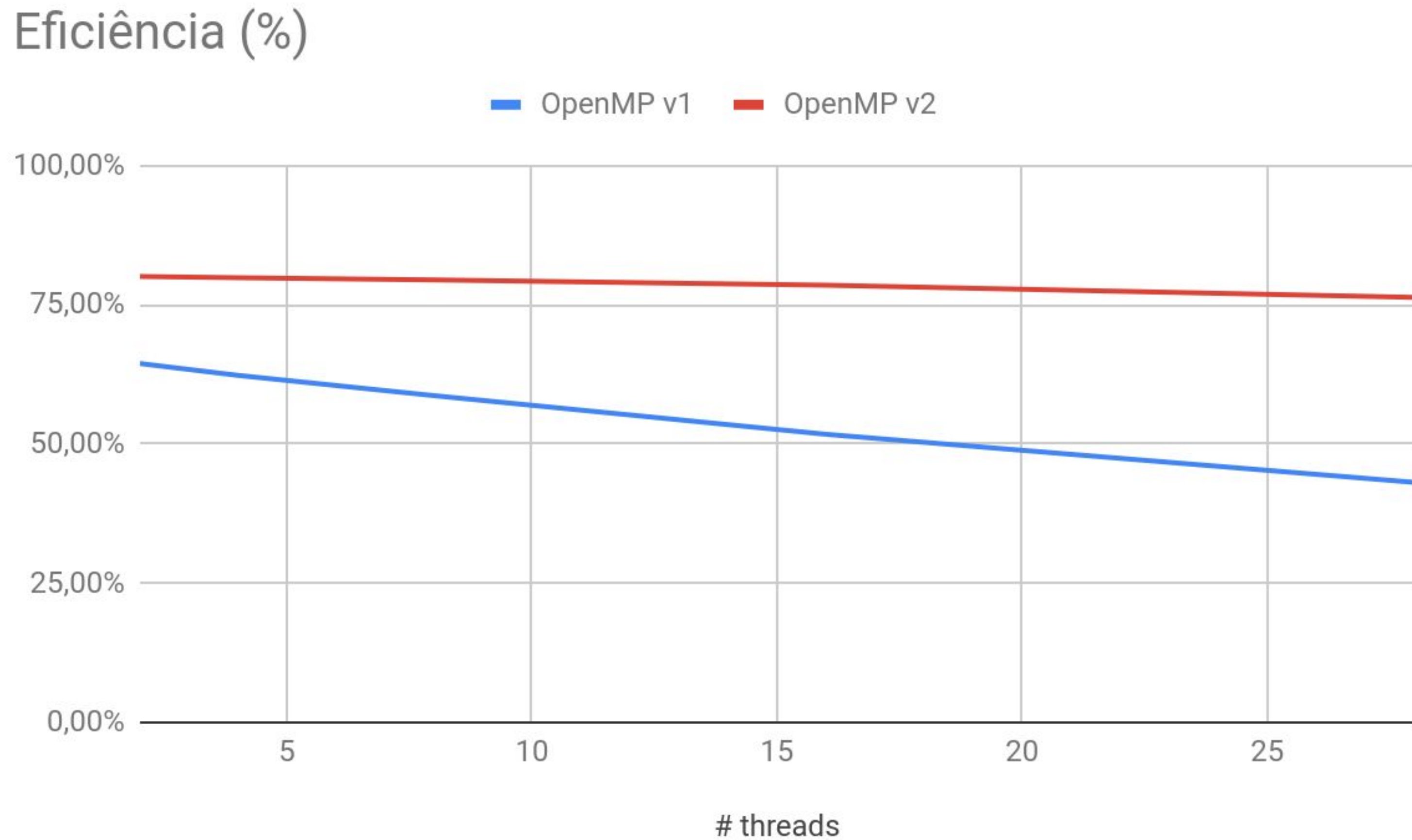
Paraleliza laço

Destroi threadas

Speedup das versões em OpenMP



Eficiência paralela das versões em OpenMP



5. Programação para GPUs

Programação com diretivas do padrão OpenACC

OpenACC

More Science, Less Programming

OpenACC: Open Programming Standard for Parallel Computing

- *https://www.openacc.org/*
- Diretivas de compilação para especificar regiões paralelas em C, C++, Fortran
- Permite programação de alto-nível para plataformas heterogêneas
- Modelo simplificado de programação
  - Similar ao OpenMP

Casos de sucesso (Fonte: NVIDIA)

Large Oil Company

3x in 7 days

Solving billions of equations iteratively for oil production at world's largest petroleum reservoirs

Univ. of Houston

Prof. M.A. Kayali

20x in 2 days

Studying magnetic systems for innovations in magnetic storage media and memory, field sensors, and biomagnetism

Uni. Of Melbourne

Prof. Kerry Black

65x in 2 days

Better understand complex reasons by lifecycles of snapper fish in Port Phillip Bay

Ufa State Aviation

Prof. Arthur Yuldashev

7x in 4 Weeks

Generating stochastic geological models of oilfield reservoirs with borehole data

GAMESS-UK

Dr. Wilkinson, Prof. Naidoo

10x

Used for various fields such as investigating biofuel production and molecular sensors.

Programação para GPUs com diretivas

OpenMP

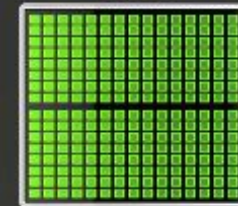
CPU

```
main() {
  double pi = 0.0; long i;

  #pragma omp parallel for reduction(+:pi)
  for (i=0; i<N; i++)
  {
    double t = (double)((i+0.05)/N);
    pi += 4.0/(1.0+t*t);
  }

  printf("pi = %f\n", pi/N);
}
```

OpenACC

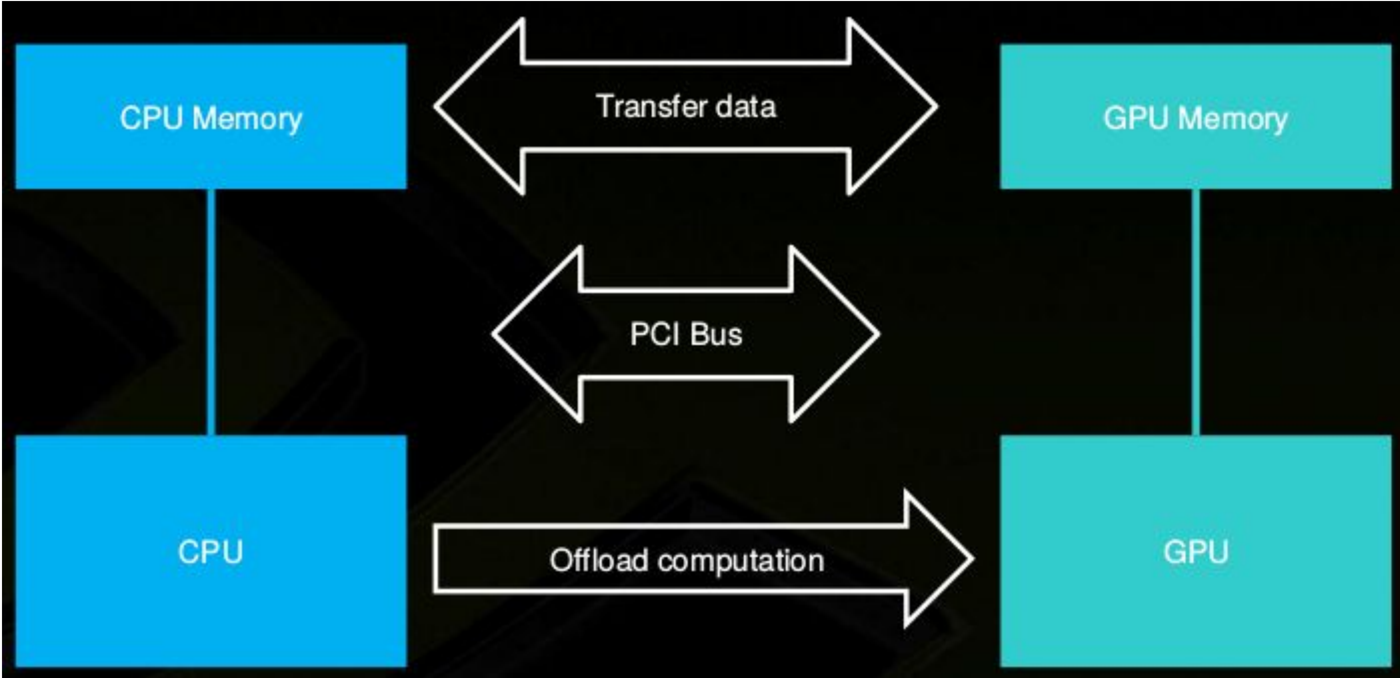
CPU

GPU



```
main() {
  double pi = 0.0; long i;

  #pragma acc parallel
  for (i=0; i<N; i++)
  {
    double t = (double)((i+0.05)/N);
    pi += 4.0/(1.0+t*t);
  }

  printf("pi = %f\n", pi/N);
}
```

Conceitos básicos



Sintaxe do OpenACC

C/C++

```
#pragma acc directive clauses
<code>
```

- Um “#pragma” instrui o compilador como compilar um trecho de código
- A palavra sentinela “acc” informa o compilador que uma diretiva do OpenACC deverá vir na sequência
- Diretivas são comandos em OpenACC
- Cláusulas adicionam características ou diferenciam diretivas

Criação de regiões paralelas aceleradas com OpenACC

`#pragma acc kernels`	`#pragma acc parallel`

- Ambas permitem definir um trecho paralelo de código
- `kernels` - mais conservadora
  - análise do compilador do trecho delimitado
  - Checa possíveis dependências de dados
  - decide se o trecho deve ou não ser paralelizado
- `parallel` - região paralela definitiva
  - garantia da não existência de dependências deve ser dada pelo programador.

Exemplo de aplicação

```
void saxpy(int n, float a,
        float *x,
        float *restrict y) {
  #pragma acc kernels
  #pragma acc loop
  for (int i=0; i<n; ++i)
      y[i] = a*x[i] + y[i];
}
```

```
void saxpy(int n, float a,
        float *x,
        float *restrict y) {
  #pragma acc parallel
  #pragma acc loop
  for (int i=0; i<n; ++i)
      y[i] = a*x[i] + y[i];
}
```

- `float *restrict y` - endereço exclusivo para `*y`
  - Sem dependência entre os arrays `*x` e `*y`
- desempenho semelhante
  - `kernels` não identifica dependências
- `Loop` - paralelizar um laço

Exemplo de operação de redução com OpenACC

```
float saxpy_sum(int n, float a, float *x,
                float *restrict y) {
     float total = 0.;
     #pragma acc parallel loop reduction (+:total)
     for (int i = 0; i < n; ++i) {
          y[i] = a * x[i] + y[i];
          total += y[i]; }
     return total;
     }
```

- `loop` usado de forma contraída
- Sumariza vários valores em um único
- Cada thread calcula uma parte
- Saída como um único valor global
  - somatórios, produtórios, extração de máximo valor, etc,

Operações possíveis em reduções com OpenACC

<table>
<tr><th>Operator</th><th>Description</th><th>Example</th></tr>
<tr><td>+</td><td>Addition/Summation</td><td>reduction(+:sum)</td></tr>
<tr><td>*</td><td>Multiplication/Product</td><td>reduction(*:product)</td></tr>
<tr><td>max</td><td>Maximum value</td><td>reduction(max:maximum)</td></tr>
<tr><td>min</td><td>Minimum value</td><td>reduction(min:minimum)</td></tr>
<tr><td>&</td><td>Bitwise and</td><td>reduction(&:val)</td></tr>
<tr><td>|</td><td>Bitwise or</td><td>reduction(|:val)</td></tr>
<tr><td>&&</td><td>Logical and</td><td>reduction(&&:val)</td></tr>
<tr><td>||</td><td>Logical or</td><td>reduction(||:val)</td></tr>
</table>

Compilação com PGI community edition (livre uso acadêmico)

```
pgcc   -acc   [-Minfo=accel]   [-ta=nvidia]   [-o   exemplo.x]
exemplo.c
```

- `-acc` habilita o padrão OpenACC
- `-Minfo=accel|opt|all`
  - Accel - informações sobre a parte acelerada
  - opt - informações sobre otimizações
  - all - todo tipo de feedback
- `-ta=<device>`
  - especifica o dispositivo alvo
  - `-ta=multicore` – acelera o código para CPUs multicore
  - `-ta=tesla` – acelera para placas NVIDIA Tesla GPUs

Opções para geração de código

```
-ta=host|multicore|tesla:{cc30|cc35|cc50|cc60|cc70|ccall|cudaX.Y|fastmath|[no]flushz|[no]fma|keep|[no]lineinfo|llc|zeroinit|[no]llvm|deepcopy|loadcache:{L1|L2}|maxregcount:<n>|pinned|[no]rdc|safecache|[no]unroll|managed|beta|autocompare|redundant}
```

`Host` - execução serial no Host

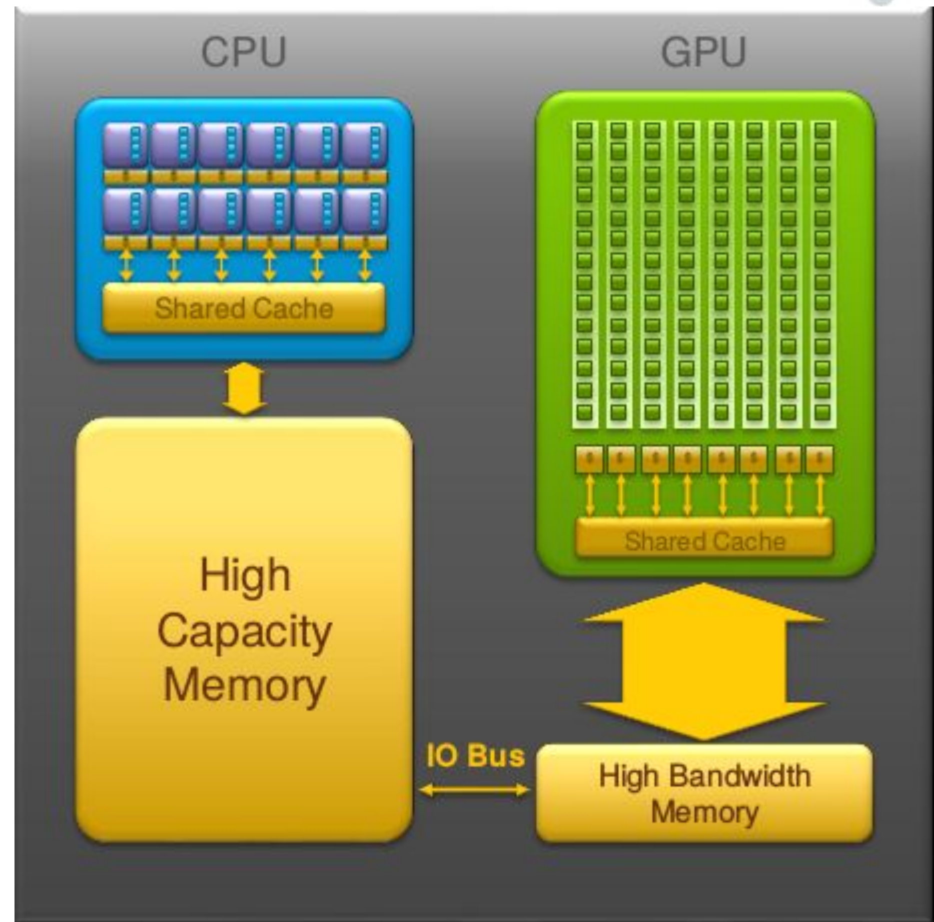
`Multicore` - Execução paralela na CPU

`Tesla` - Execução paralela na GPU Tesla

- `ccXX` - NVIDIA compute capability
- `Managed` - CUDA Managed Memory

Transferências de memória entre CPU e GPU em OpenACC

- Memória da CPU é geralmente grande
- na GPU tem maior largura de banda
- Memórias de CPU e GPU normalmente separadas
  - Conectada por I/O bus (PCI-e)
- ***Bandwith >> I/O Bus***
- GPU precisa de dados em sua própria memória



Gerenciamento de transferências de memória em OpenACC

Permite definir região de transferencia de dados com operações específicas (**`#pragma acc XXXXX`**):

- **`copy(vars)`** - Aloca memória no acelerador, faz cópia *host→ device* no início e *device→host* no final
- **`copyin(vars)`** - Aloca memória no acelerador, copia *host→ device* no início
- **`copyout(vars)`** - Aloca memória no acelerador, copia *device→host* no final
- **`create(vars)`** - Aloca memória no acelerador (dados temporários)
- **`present(vars)`** - Indica que os dados já estão presentes (ou alocados) no acelerador

Exemplo de código com transferência de dados explicitada

```
float saxpy_sum(int n, float a, float *x,
                float *restrict y) {
     float total = 0.;
     #pragma acc parallel copyin(x[0:n]) copy(y[0:n])\
                   copyout(total)
          {
          #pragma acc loop reduction (+:total)
          for (int i = 0; i < n; ++i) {
               y[i] = a * x[i] + y[i];
               total += y[i]; }
          } // fim da região com transf. de dados
     return total;
     }
```

Estudo de caso com o programa **Jogo da Vida** em OpenACC Implementação “ingênua”.

```
for (iter = 0; iter<maxIter; iter++) {
   // Loop para nova geracao
   #pragma acc parallel
   #pragma acc loop
   for (i = 1; i<=dim; i++) {
      for (j = 1; j<=dim; j++) {
         ... }
      }
      ... // restante do codigo }
```

Demais laços para preenchimento de fronteiras esquerda, direita, superior e inferior receberam diretivas similares

Atualização de dados entre gerações em GPU

```
// troca de arrays para proxima geracao
#pragma acc parallel loop
for(i = 1; i <= dim; i++) {
    for(j = 1; j <= dim; j++) {
        grid[i][j] = newGrid[i][j];
        }
    } // copia de dados entre grid e newGrid
```

- Troca de ponteiros entre grid e newGrid não é possível em GPU.
- Substituída por uma simples cópia entre os dois arrays

Redução no cálculo da somatória do tabuleiro

```
// Somatorio das celulas do tabuleiro
int total = 0;
#pragma acc parallel loop reduction(+:total)
for (i = 1; i<=dim; i++) {
    for (j = 1; j<=dim; j++) {
        total += grid[i][j]; }
    }
```

Laço ocorre posterior ao laço principal, ao final da execução, executando uma única vez

Compilação com PGI

```
pgcc GOL-openacc.c  -acc   -ta=nvidia:kepler
-Minline    -Minfo   -o teste.x
```

- **-acc**: Ativa OpenACC
- **-ta=nvidia:kepler**: Especifica dispositivo alvo
- **-Minline**: Permite “incorporar” funções onde são chamadas (necessária para acelerar a função “getNeighbors”)
- **-Minfo**: Retorno com informações sobre a compilação

Problema inesperado - transferência de dados com erro

```
80, Accelerator kernel generated
  Generating Tesla code
  82, #pragma acc loop gang /* blockIdx.x */
  83, #pragma acc loop vector(128) /* threadIdx.x */
  80, Generating implicit copyin(grid[1:2048][1:2048])
      Generating implicit copy(newGrid[1:2048][1:2048])
  85, getNeighbors inlined, size=16, file GOL-openacc.c (8)
          83, Loop is parallelizable
```

Ignorou as bordas do tabuleiro. A faixa de valores deveria ser de 0 até 2050 em ambos os eixos.

Função “getNeighbors” incorporada ao trecho acelerado

A transferência incompleta de dados, que não contempla todos os valores necessários, acabou gerando um erro de execução!!

Solução para corrigir a transferência de dados incompleta

```
// Loop sobre as celulas para nova geracao
#pragma acc parallel \
        copyin(grid[0:fullSize][0:fullSize]) \
        copy(newGrid[0:fullSize][0:fullSize])
#pragma acc loop
for (i = 1; i<=dim; i++) {
    for (j = 1; j<=dim; j++) {
        ...
        }
    }
```

Definiu-se explicitamente a faixa de valores a ser transferida para os arrays envolvidos

Laço paralelo corrigido em OpenACC para cálculo de nova geração do tabuleiro, contemplando toda a matriz

Desempenho da primeira versão paralela em GPU por OpenACC
(NVIDIA TitanBlack - 2880 cuda cores @ 889 MHz, 6GB mem.)

Versão	Tempo (s)	Speedup
Serial	330,474	1
OpenMP v2 - 28 threads	15,463	21,37
OpenACC v1	1208,54	0,27

Desempenho considerado inesperado e pífio

Razões para o baixo desempenho da Versão 1

- Tempo com transferências de dados entre a CPU e GPU
- As três primeiras regiões paralelizadas estão dentro de um laço que se repete 10.000 vezes (para calcular 10.000 gerações)
- Transferências entre CPU e GPU ocorrem 70.000 vezes
  - 20.000 para o primeiro laço - array `grid` (10.000 CPU→ GPU no início e 10.000 GPU→ CPU ao término)
  - 20.000 para o segundo laço de forma semelhante
  - 30.000 para o terceiro laço (CPU→ GPU de `grid` e `newGrid` no início, e GPU→ CPU de `newGrid` ao término)

Versão 2 - diminuindo a quantidade de transferências de dados desnecessárias

```
#pragma acc data copy(grid[0:fullSize][0:fullSize]) \
                 create(newGrid[0:fullSize][0:fullSize])
{ // principal regiao paralela acelerada
    for (iter = 0; iter<maxIter; iter++) {
        #pragma acc parallel loop
        for (i = 1; i<=dim; i++) { // Fronteiras esq. e dir.
            ...  }
        // Linhas superior e inferior
        #pragma acc parallel loop
        for (j = 0; j<=dim+1; j++) { // Fronteiras sup. e inf.
            ...   }
        #pragma acc parallel loop
        for (i = 1; i<=dim; i++) { // Nova geracao
            for (j = 1; j<=dim; j++) {
              ...    } }
    } // Fim do laco principal
} // Fim da regiao paralela principal
```

Transferência de dados ocorre apenas no início e fim do algoritmo principal do Jogo da Vida

Desempenho da versão 2 em OpenACC

Versão	**Tempo (s)**	***Speedup***
Serial	330,474	1
OpenMP v2 - 28 threads	15,463	21,37
OpenACC v1	1208,54	0,27
OpenACC v2	6,50	50,84 (2,37 em relação a versão OpenMP v2)

- Speedup:
  - > 50 vezes em relação a versão serial
  - 2,37 vezes a versão OpenMP

6. Considerações finais

Considerações finais

- A programação paralela ainda é considerada difícil.
- O uso de diretivas é um bom ponto de partida para projetos de paralelização.

→ Esperamos ter ajudado :)

Álvaro Fazenda: alvaro.fazenda@unifesp.br
Denise Stringhini: dstringhini@unifesp.br