Programação Paralela para Aplicações Científicas

Celso L. Mendes

LAC /INPE

Email: celso.mendes@inpe.br

WORCAP-2019

Programação Paralela para Aplicações Científicas

Tópicos:

1. **Introdução**
2. **Programação com Memória Compartilhada**
3. **Programação com Memória Distribuída**
4. **Programação de Aceleradores**
5. **Conclusão & Referências**

Introdução

- Processamento *Paralelo*:
  - Uso simultâneo de mais que uma CPU em *uma* tarefa
  - Deve haver coordenação entre as várias CPUs
- Processamento Paralelo ≠ Processamento Distribuído
  - Em Proc. Distribuído, tipicamente usa-se várias CPUs (em geral espalhadas geograficamente) para várias tarefas, uma por CPU
- Processamento Paralelo ≠ Proc. de Alto Desempenho (PAD)
  - PAD pode não ser paralelo, mesmo que seja "rápido"
    - Por exemplo, processamento com FPGAs

Introdução (cont.)

- Motivação para uso de Processamento *Paralelo*:
  - Largamente disponível com hardware atual
  - Geração corrente de processadores (pós-2004):
    - Velocidade de relógio estagnada
    - Aumento no número de núcleos (CPUs) por processador
    - Processadores *multi-core*:
      - Utilizados nos mercados de massa: laptops, celulares!
      - Utilizados como elemento básico de máquinas maiores

Introdução (cont.)

- Mercado de Massa: laptops

Introdução (cont.)

- Mercado de Massa: laptops

Notebook Ideapad 33 Intel Core I5-8250u 8GB 1TB HD 15.6" W10 Prata - Lenovo

Intel Quad-Core i5-8250U

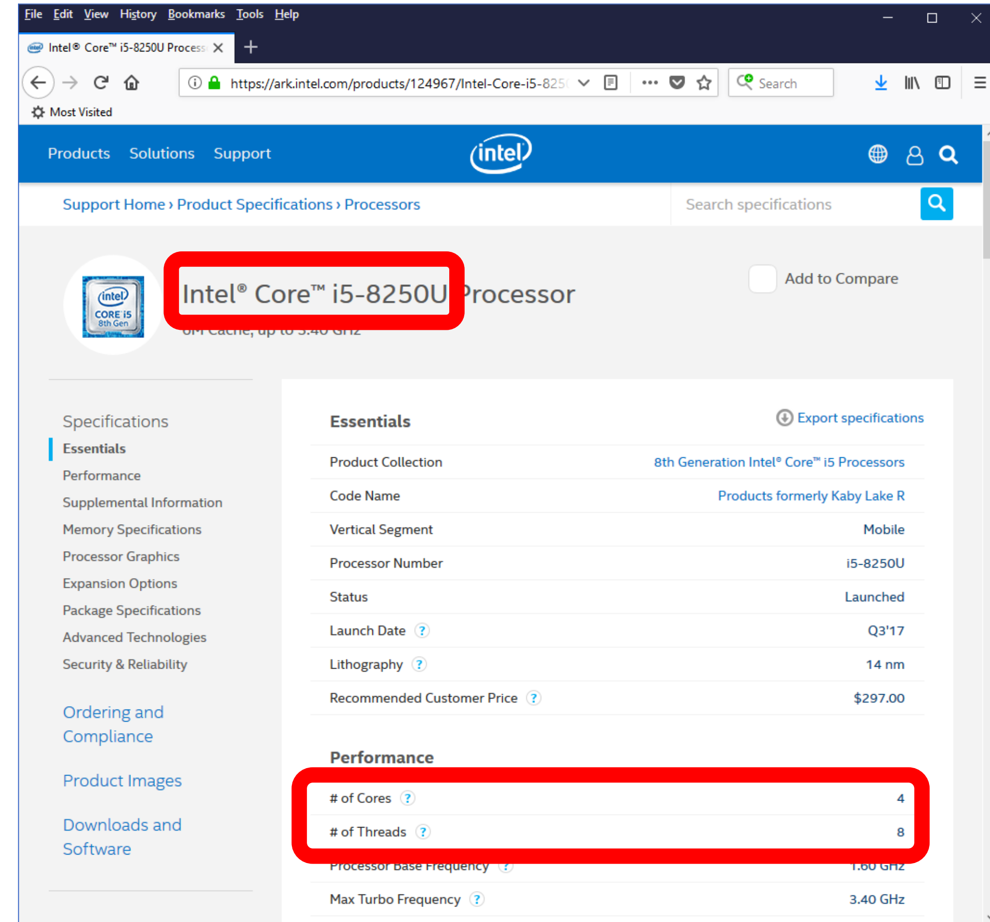
Introdução (cont.)

- Mercado de Massa: laptops
  - Comum: >1 núcleo (*core*)
  - Típico: >1 thread/core

  - Intel i5-8250U:
    - 4 cores
    - 8 threads
    - ➢ 2 threads/core

    Nomenclatura Intel:

    Hyper-threading



Introdução (cont.)

- Máquinas Maiores - Ex: Santos Dumont (Fabricante: Bull)

Local: LNCC
Petrópolis/RJ

Introdução (cont.)

- Sistema Santos Dumont:
    - 504 nós com dois procs. **Intel-Xeon 12-núcleos** (24 núcleos/nó)
    - 198 nós com dois procs. **Intel-Xeon 12-núcleos** + 2 GPUs Nvidia K40
    - 54 nós com dois procs**. Intel-Xeon 12-núcleos** + 2 Intel Xeon-Phi
    - 1 nó com 16 procs. **Intel-Xeon 15-núcleos**, 6 TB de memória
    - Desempenho de pico agregado acima de 1.1 Petaflops
    - Rede de interconexão Infiniband
    - 3 sistemas listados na Top500 de Nov/2015, 1 em Jun/2017, nenhum hoje
    - Acesso: alocações disponíveis para a comunidade acadêmica
        - http://sdumont.lncc.br/sdumont.php?pg=sdumont#

Introdução (cont.)

- Programação Paralela:
  - Como escrever um programa que explore todas as CPUs disponíveis?
- Linguagens Utilizadas para Paralelismo:
  - Centenas (milhares ?) de novas linguagens propostas
    - Na maior parte, de interesse puramente acadêmico
  - Na prática: extensões de linguagens convencionais
    - Tipicamente, em aplicações científicas → Fortran, C, C++
    - Cada extensão tem seus prós e contras!
  - Alternativa recente: DSL – *Domain-Specific Language*
    - Define estruturas/operadores típicos de uma certa área
    - Geralmente são compiladas para uma linguagem convencional

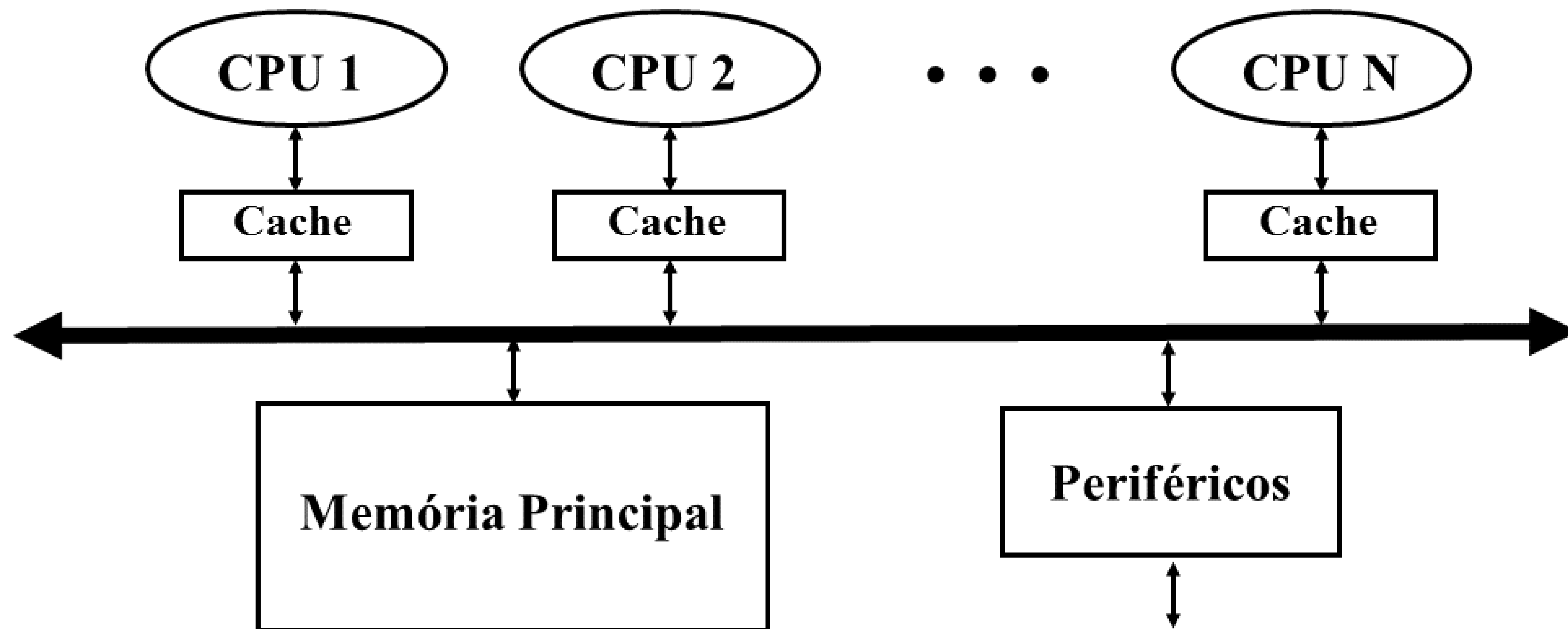
Programação Paralela para Aplicações Científicas

Tópicos:

1. Introdução
2. **Programação com Memória Compartilhada**
3. Programação com Memória Distribuída
4. Programação de Aceleradores
5. Conclusão & Referências

Prog. com Memória Compartilhada

- Ambiente de hardware típico



Prog. com Memória Compartilhada

- Objetivo: explorar todas as CPUs, minimizando o overhead de comunicação entre elas
- Paradigma preferencial: programação com *threads*
  - *thread* = fluxo de instruções, com registros e pilha próprios
  - Programa paralelo: um único processo
  - Alocação thread/CPU é feita pelo Sistema Operacional
  - Comunicação e sincronização via memória comum

Prog. com Memória Compartilhada

- Possíveis implementações de Threads:
  a) Vários threads em vários núcleos comuns
     - 1 thread/núcleo: overhead apenas de criação dos threads
  b) Vários threads em <u>um</u> núcleo comum
     - Núcleo executa um thread de cada vez
     - Ciclos de relógio são divididos pelos threads físicos
     - A cada chaveamento de thread: salvar/restaurar registros
  c) Vários threads em um núcleo com <u>suporte de hw multi-thread</u>
     - Um thread lógico alocado em cada thread físico
     - Ciclos de relógio são divididos pelos threads físicos
     - Não há overhead de chaveamento de threads – suporte de hw

Programação com Pthreads

- Pthreads: POSIX threads
  - Padrão POSIX, mais de 100 funções definidas
  - Criação de threads, encerramento, sincronização, etc.
  - Interface para programas em C apenas
  - Sintaxe de todas as funções: *pthread_<nome>(…)*
  - Arquivo a ser incluído: *pthread.h*
  - Compilação/link: *gcc –o prog prog.c –lpthread*
    - Invoca biblioteca *libpthread.a*
    - Disponível em quase todos os sistemas do tipo Unix

Exemplo com Pthreads

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
int global_var = 4 ;
void *func (void *value);
int main(argc,argv)
int argc; char *argv[];
{
     int i, count ;
     pthread_t *pts;
     count = global_var ;
     pts = (pthread_t *) malloc(count*sizeof(pthread_t));
     /* Cria outros threads */
     for (i=0; i<count; i++) {
         pthread_create(&pts[i], NULL, func, (void*) i);
     }
     /* Thread principal faz algum trabalho */
     printf("Este e' o thread principal\n");
```

variável global

Exemplo com Pthreads (cont.)

```
        pthread_create(&pts[i], NULL, func, (void*) i);
    }
    /* Thread principal faz algum trabalho */
    printf("Este e' o thread principal\n");
    /* Termina outros threads */
    for (i=0; i<count; i++) {
        pthread_join(pts[i], NULL);
    }
    free(pts);
}
void *func (void *value) {
    int local_var = global_var;
    /* thread realiza trabalho individual */
    printf("==> Este e' o thread %d de um total de %d\n", value, local_var);
    return NULL;
}
```

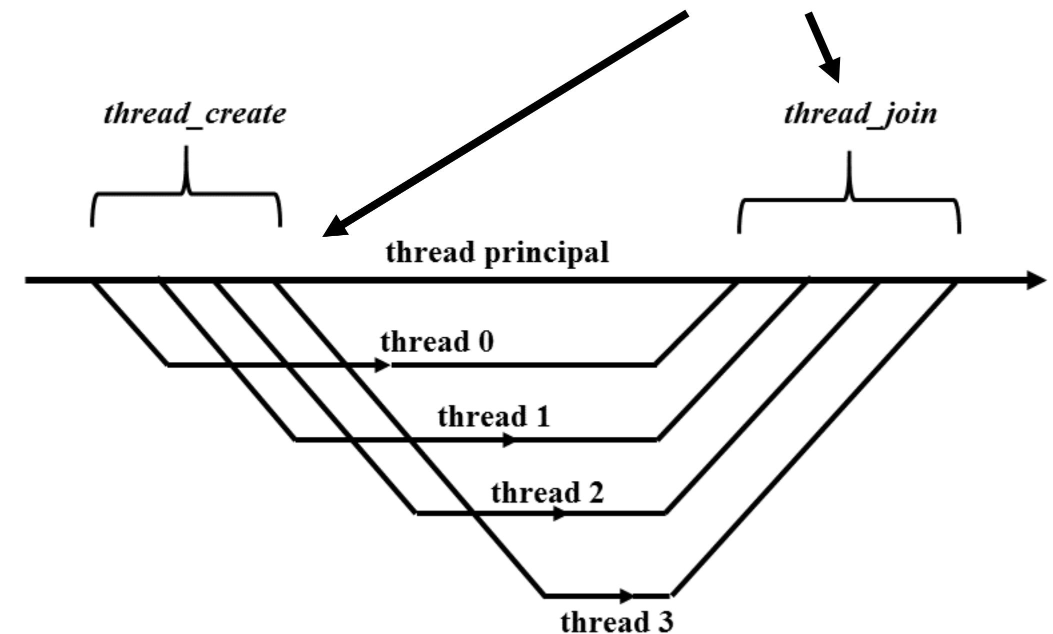
variável privativa do thread

Exemplo com Pthreads (cont.)

- Execução:

==> Este e' o thread 0 de um total de 4
==> Este e' o thread 1 de um total de 4
==> Este e' o thread 2 de um total de 4
==> Este e' o thread 3 de um total de 4
Este e' o thread principal

Modelo de execução fork/join:



Programação com OpenMP

- Principais motivações:
  - Passar o gerenciamento de threads para o compilador
  - Simplificar a programação e migração de programas seriais
- OpenMP:
  - Padrão definido pelo fórum OpenMP, década de 90
  - Baseado em dois componentes:
    1. Diretivas inseridas no código-fonte
    2. Biblioteca com funções de apoio
  - Disponível para programas em C/C++ e Fortran

Programação com OpenMP (cont.)

- Principais características:
  - Compiladores podem ignorar as diretivas
  - Número de threads pode ser definido com variável ambiental

    OMP_NUM_THREADS
  - Cada thread tem um identificador: 0, 1, 2,…
    - thread principal sempre existe, e tem identificador = 0
    - identificador pode ser recuperado com *omp_get_thread_num()*
  - É possível requerer que variáveis globais sejam privatizadas
    - implementado através de anotação extra na diretiva

Programação com OpenMP (cont.)

- Escopo das diretivas
  - *Bloco básico* imediatamente posterior
    - Conjunto de instruções com entrada única e saída única
  - Encarado como *região paralela*
    - Cada instância da região é executada por um thread
  - Se região é um laço: iterações são distribuídas pelos threads
    - Distribuição em bloco: {i=0,1},{i=2,3},{i=4,5},{i=6,7}
    - Distribuição cíclica: {i=0,4},{i=1,5},{i=2,6},{i=3,7}
    - Outras formas de distribuição são possíveis

Exemplo com OpenMP

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int global_var = 4 ;
void func ();
#define N 8
int main(argc,argv)
int argc; char *argv[];
{
    int i, count ;
    count = global_var ;
    /* Invoca a execucao dos varios threads */
#pragma omp parallel for num_threads(count)
    for (i=0; i<N; i++)  func(i) ;
    /* Thread principal imprime mensagem */
    printf("Este e' o thread principal\n");
}
void func (int arg) {
    int local_var;
    /* thread realiza trabalho individual */
    local_var = omp_get_thread_num();
    printf("--> Este e' o thread-omp %d de um total de %d, arg=%d\n",
        local_var, global_var, arg);
}
```

diretiva inserida no código

função da biblioteca OpenMP

OpenMP - Resumo

- Principais vantagens
  - Simplificação da programação em relação a Pthreads
  - Permite paralelização “incremental” de programas antigos
  - Padrão maduro – mais de duas décadas de amadurecimento
  - Quase todos os compiladores atuais suportam o padrão
  - Fórum ainda ativo – novas versões em discussão
    - V. 5.0: publicada
- Mais informações: www.openmp.org

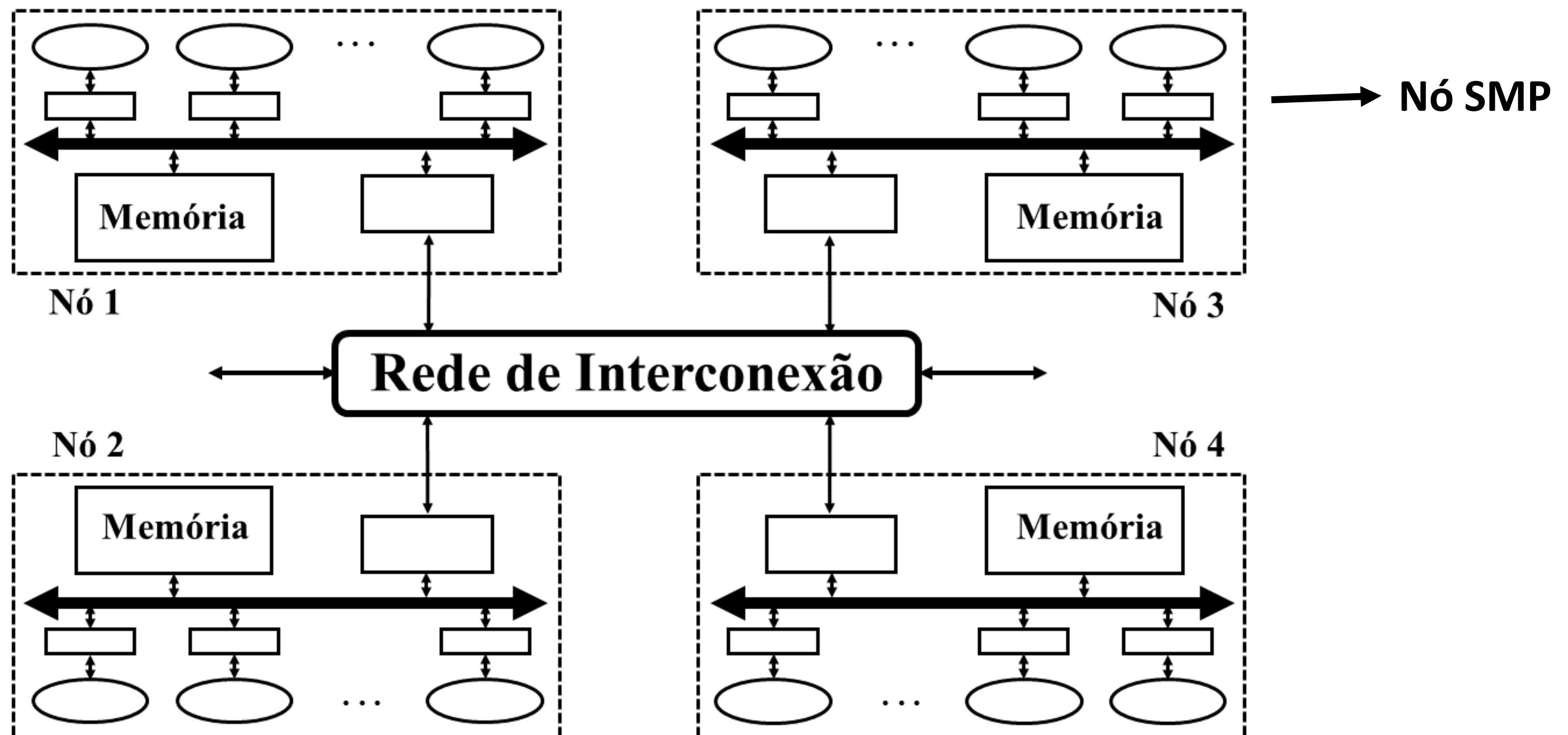
Programação Paralela para Aplicações Científicas

Tópicos:

1. Introdução
2. Programação com Memória Compartilhada
3. **Programação com Memória Distribuída**
4. Programação de Aceleradores
5. Conclusão & Referências

Programação com Memória Distribuída

- Ambiente de hardware típico



Programação com Memória Distribuída

- Objetivo: explorar todas as CPUs, com overhead de comunicação entre elas tolerável, para executar um programa
- Paradigma preferencial: troca de mensagens entre CPUs
  - Um ou mais processos por nó
  - Mensagem entre processos
  - Participação necessária do Sistema Operacional

Troca de Mensagens - MPI

- Situação original:
  - Cada fabricante fornecia sua biblioteca de troca de mensagens
  - Mudança de plataforma sempre exigia ajustes no código
- Padronização: Message-Passing Interface (MPI)
  - Fórum MPI criado na década de 90: indústria, labs, universidades
    - V. 3.1 publicada, V. 4.0 em ativa discussão (www.mpi-forum.org)
  - Especificação para Fortran, C e C++ (C++ não é mais suportada hoje)
  - Implementação modelo: MPICH (Argonne National Lab - EUA)
    - Código totalmente aberto, com documentação, etc (www.mpich.org)
    - Licenciada por diversos fabricantes
  - Forte adesão de toda a comunidade

MPI Básico

- Seis funções *canônicas*:
  - MPI_Init(): inicialização da biblioteca
  - MPI_Finalize(): encerramento da biblioteca
  - MPI_Comm_size(): retorna número de processos
  - MPI_Comm_rank(): retorna identificador (rank)
  - MPI_Send(): envia mensagem
  - MPI_Recv(): recebe mensagem

É possível escrever ~99% dos programas MPI com isto!
(porém com pouca eficiência em alguns casos)

Exemplo com MPI

```
#include <mpi.h>
#define SIZE 100

int main(argc,argv)
int argc; char *argv[];
{
    int i,numprocs,myid,
    char *buf;
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    buf = (char *)malloc(SIZE);

    if (myid==1) {
        for (i=0; i<SIZE; i++) buf[i] = (i%64);
        MPI_Send(buf,SIZE,MPI_CHAR,0,0,MPI_COMM_WORLD);
        MPI_Recv(buf,SIZE,MPI_CHAR,0,0,MPI_COMM_WORLD,&status);
    } else if (myid==0) {
        MPI_Recv(buf,SIZE,MPI_CHAR,1,0,MPI_COMM_WORLD,&status);
        MPI_Send(buf,SIZE,MPI_CHAR,1,0,MPI_COMM_WORLD);
    }
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();
}
```

Programa:
rank 1 → rank 0
rank 0 → rank 1

- inicializações (MPI_Init, MPI_Comm_size, MPI_Comm_rank)
- executado pelo rank 1 (for, MPI_Send, MPI_Recv)
- executado pelo rank 0 (MPI_Recv, MPI_Send)
- espera por todos os ranks (MPI_Barrier)

Evolução de MPI

- MPI 1: 1994 (1.0, 1.1, 1.2, 1.3)
  - Funções ponto-a-ponto, coletivas, datatypes, topologia
- MPI-2: 1997 (2.0, 2.1, 2.2)
  - I/O paralelo, comunic. unilateral, múltiplos threads, etc.
- MPI-3: 2012 (3.0, 3.1)
  - Coletivas sem bloqueio, oper. mem. compartilhada, etc.
- MPI-4: em discussão
- Info: https://www.mpi-forum.org/

Programação Híbrida MPI+OpenMP

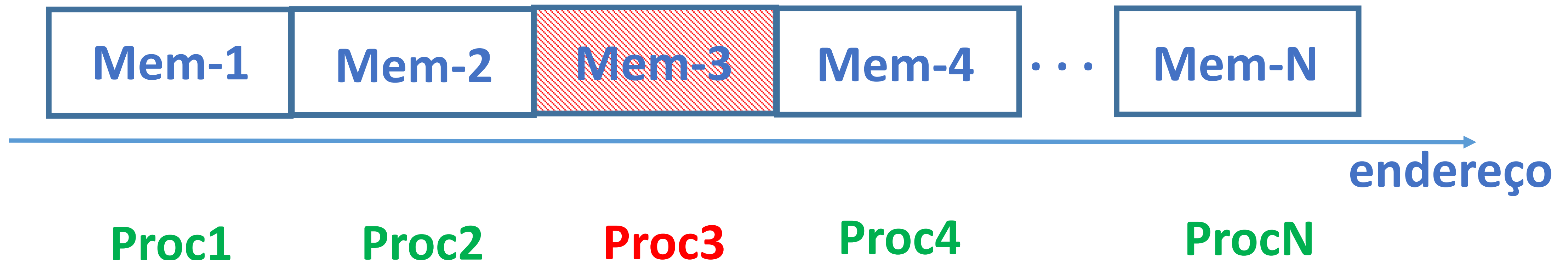
- Idéia:
  - Usar ranks MPI em diferentes nós
  - Usar threads OpenMP nos processadores de um nó
- Uso Típico:
  - #ranks/nó × #threads/rank = #CPUs/nó (ou seja: 1 thread/cpu)
- Vantagens
  - Minimização do número de processos MPI
  - Aproveitamento de todas as CPUs em um nó
- Possíveis problemas
  - Pode não ser possível chamar funções MPI em todos os threads: depende da implementação da biblioteca MPI
  - Programação é mais complexa – dois níveis de paralelismo

Programação com Memória Global Distribuída

- Objetivos:
  - Elevar o nível de abstração da programação em multicomputadores
  - Evitar ter que lidar com troca de mensagens no programa
- Abordagem: linguagens PGAS
  - *Partitioned Global Address Space*
  - Oferecer a cada processador a *ilusão* de memória global
- Implementações
  - CoArray Fortran, Unified Parallel C (UPC), Chapel, X10, Fortress

Linguagens PGAS

- Princípios:
  - Todas as memórias fazem parte do espaço de endereços
  - Para cada processador, uma faixa de endereços é local (rápida), as outras faixas são remotas (lentas)
  - ProcK: Mem-K=local (rápida), Mem-j(j≠K)=remotas (lentas)



Exemplo com CoArray Fortran

Programa Fortran2008

```
program reduce
        integer :: my_rank, num_procs, i, result
        integer      :: coarray[*]

        my_rank   = this_image()
        num_procs = num_images()
        if ( (my_rank/2)*2 .EQ. my_rank ) then
                coarray[my_rank] = my_rank;
        else
                coarray[my_rank] = my_rank * (-1);
        end if
        result = 0

        sync all
        if (my_rank .EQ. 1) then
                do i=1, num_procs
                        result = result + coarray[i]
                end do
        end if
        sync all

        if (my_rank .EQ. 1) print *,"CAF_size: ",num_procs,"  result: ",result
        sync all
end program reduce
```

co-array

inicializações

barreira (sincroniza todos os processadores)

Proc 1 faz todo o trabalho

Programação Paralela para Aplicações Científicas

Tópicos:

1. Introdução
2. Programação com Memória Compartilhada
3. Programação com Memória Distribuída
4. **Programação de Aceleradores**
5. Conclusão & Referências

Aceleradores

- Objetivo
  - Complementar CPU de propósito geral com maior capacidade de cálculos numéricos
- Longo histórico
  - Co-processadores aritméticos, DSPs, etc.
- Esquema predominante
  - CPU é usada para programação, coordenação da execução
  - Acelerador opera sobre partes críticas da tarefa
- Tipos de aceleradores: vários!

Acelerador Típico: GPU

- *Graphics Processing Unit*
  - Inicialmente criada para processar exclusivamente gráficos
  - Mercados de massa: placas de video de PCs, consoles de jogos
    - Preço accessível, devido à economia de escala
    - Difícil programação, tarefa para especialistas
    - Primeiros modelos: ponto-flutuante em precisão simples
- GPGPU: *General Purpose* GPUs
  - Possibilidade de uso em aplicações de PAD
  - Ponto-flutuante em precisão dupla
  - Esforço para facilitar as técnicas de programação

GPU - Exemplo

- GPU – Exemplo: Nvidia K40
  - Mais de 7.1 bilhões de transistores
  - Relógio: 745 MHz
  - 15 “processadores” SMX, 192 núcleos/processador
    - 2880 núcleos no total
  - Threads são agrupados em conjuntos de 32 (*warp*)
  - Desempenho: mais que 1 Tflops efetivos, em precisão dupla
    - 80% do desempenho de pico
  - 6 canais de acesso à memória de 64 bits cada
  - Hierarquia de memória
    - cache L1: em cada processador SMX
    - cache L2: compartilhada
    - memória principal (externa) DRAM, com ECC opcional

GPU - Exemplo (cont.)

- Nvidia K40 – Diagrama de Blocos

PCI Express 3.0 Host Interface

GigaThread Engine

SMX

L2 Cache

Memory Controller

Processador SMX (15)

Cache L2

Cache L1

Control.Memória (6)

fonte: Nvidia

Programação de Aceleradores

- GPU: duas alternativas concretas
  a. Linguagens – OpenCL, CUDA (Nvidia), HIP (AMD)
  b. Diretivas – OpenACC, OpenMP
- CUDA: extensão de C, com duas modificações
  1. Declaração de funções (*kernels*) para a GPU:
     ___global___ mykernel
  2. Invocação de kernels: *mykernel<<<val1,val2>>>(args)*

Exemplo com CUDA

```
#include <stdio.h>
#include <cuda.h>
__global__ void meukernel(int * p) {
        int val = ( *p) * blockIdx.x + threadIdx.x ;
        printf("Dentro do Kernel, valor=%2d coords=%d,%d\n",
                val, blockIdx.x, threadIdx.x);
}
main() {
        int x=9, *x_gpu;
        printf("No inicio do Programa Principal\n");
        int size=sizeof(int);
        cudaMalloc((void **)&x_gpu,size);
        cudaMemcpy(x_gpu, &x, size, cudaMemcpyHostToDevice);
        meukernel<<<4,2>>>(x_gpu);
        cudaFree(x_gpu);
        printf("No final do Programa Principal\n");
}
```

roda na GPU

roda na CPU

4 blocos, 2 threads/bloco

Exemplo com CUDA (cont.)

- Compilação/link:
  - *nvcc –o prog prog.cu*
- Execução:

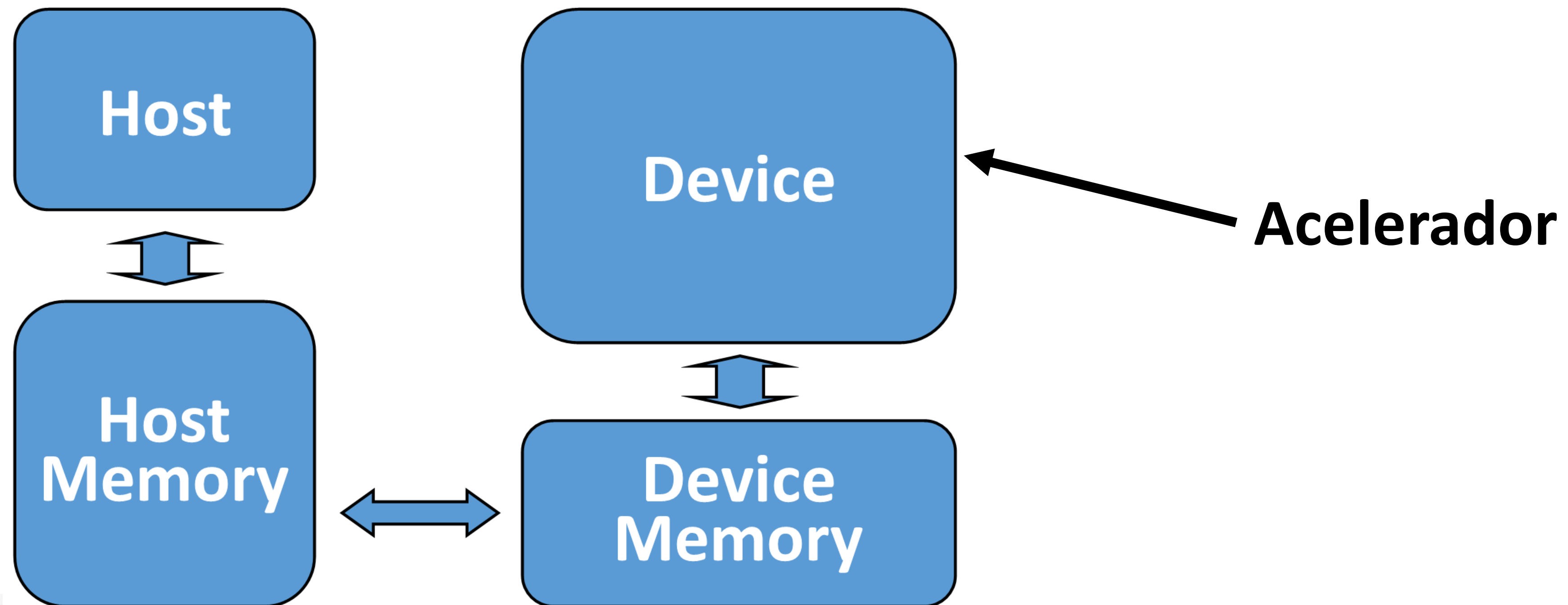
```
No inicio do Programa Principal
Dentro do Kernel, valor= 9 coords=1,0
Dentro do Kernel, valor=10 coords=1,1
Dentro do Kernel, valor=18 coords=2,0
Dentro do Kernel, valor=19 coords=2,1
Dentro do Kernel, valor=27 coords=3,0
Dentro do Kernel, valor=28 coords=3,1
Dentro do Kernel, valor= 0 coords=0,0
Dentro do Kernel, valor= 1 coords=0,1
No final do Programa Principal
```

Padrão OpenACC

- Foco: compromisso desempenho/portabilidade
- Objetivos Principais
  - Oferecer o máximo de portabilidade entre arquiteturas
  - Garantir desempenho minimamente aceitável
  - Permitir otimização gradativa de códigos existentes/legados
- Método: diretivas inseridas pelo programador
  - Fortran: comentários - !$acc <diretiva>
  - C/C++: pragmas - #pragma acc <diretiva>
- Padrão OpenAcc: https://www.openacc.org/
  - V. 1.0 (2011), V.2.0 (2013), V.2.5 (2015), V.2.6 (2017), V.2.7 (2018)

Modelo Abstrato de OpenACC

- Host: CPU principal; Device: Acelerador
  - Host/Device podem ser os mesmos, memórias também!
  - Processamento baseado em *offloading* de código e/ou dados



OpenACC - Exemplo

- Código Original

```
for (i=0; i<N; i++)
{
  y[i] = 0.0;
  x[i] = (float)(i+1);
}
```

Loop-1

```
for (i=0; i<N; i++)
{
  y[i] = 2.0f * x[i] + y[i];
}
```

Loop-2

- Código comOpenACC

```
#pragma acc parallel loop
for (i=0; i<N; i++)
{
  y[i] = 0.0;
  x[i] = (float)(i+1);
}
#pragma acc parallel loop
for (i=0; i<N; i++)
{
  y[i] = 2.0f * x[i] + y[i];
}
```

OpenACC – Exemplo (cont.)

- Compilação Trivial:

  Mover X e Y da mem.Host p/ mem.Device

  Executar Loop-1 no Device (acelerado)

  Mover X e Y de volta p/ mem. Host

  desnecessários ?

  Mover X e Y da mem.Host p/ mem.Device

  Executar Loop-2 no Device (acelerado)

  Mover Y de volta p/ mem.Host

- Possível complicação: Loop-1 e Loop-2 podem estar em arquivos separados!
- Solução: Utilizar diretivas de controle sobre a movimentação de dados
  - Dispensáveis para resultados corretos; essenciais para bom desempenho!

OpenMP para Aceleradores

- Suporte de OpenMP para Aceleradores
  - Introduzido em OpenMP-4.0, atualizado em OpenMP-4.5
  - Diretivas para computação heterogênea em OpenMP:
    - Execução de trechos de programas
    - Gerenciamento de dados
- Modelo de Execução: similar a OpenACC
  - Host (CPU) e Device (Acelerador)
  - Código é enviado (*offloaded*) do Host para o Device, com os dados necessários sendo transferidos

OpenMP para Aceleradores

- Exemplo:

```
#pragma omp target map(a,b,c,d)
{
        for (i=0; i<N; i++) { a[i] = b[i] * c + d; }
}
```

  - *target* : identifica código a ser executado no acelerador
  - *map* : indica variáveis a serem mapeadas entre memórias
    - É possível determinar sentido do mapeamento de cada variável (entrada, saída, entrada/saída=default) → Evita cópias inúteis

Programação Paralela para Aplicações Científicas

Tópicos:

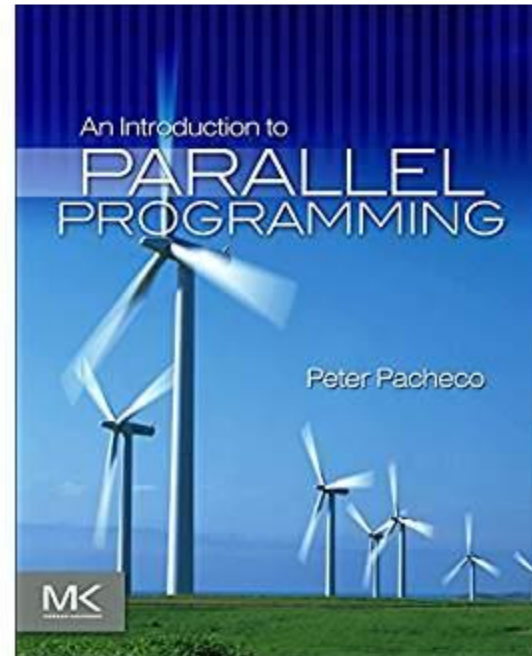
1. Introdução
2. Programação com Memória Compartilhada
3. Programação com Memória Distribuída
4. Programação de Aceleradores
5. **Conclusão & Referências**

Conclusão

- Processamento Paralelo
  - Largamente empregado, de laptops a supercomputadores
  - Presente em quase todos os processadores atuais
- Programação paralela em aplicações científicas
  - Vários paradigmas, dependendo da plataforma
  - Tipicamente, extensão de linguagens convencionais
  - Mem. Compartilhada: Pthreads, OpenMP
  - Mem. Distribuída: MPI, MPI+OpenMP, PGAS
  - Aceleradores: Cuda, HIP, OpenACC, OpenMP

Referências

- Programação paralela em aplicações científicas
  - Peter Pacheco, *An Introduction to Parallel Programming*



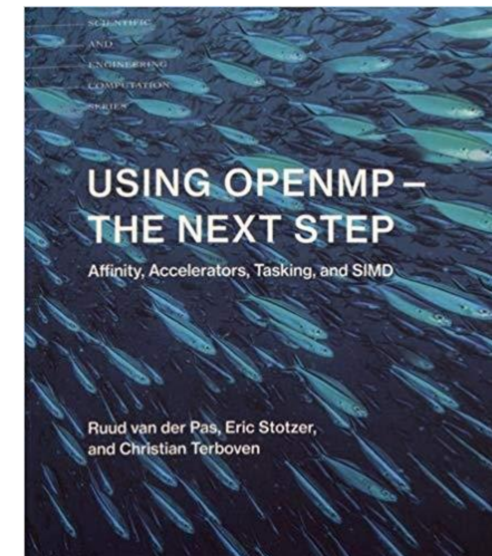
Morgan Kaufmann
2011

Referências

- Prog. com Memória Compartilhada: OpenMP

- B.Chapman et al, *Using OpenMP* MIT Press, 2007

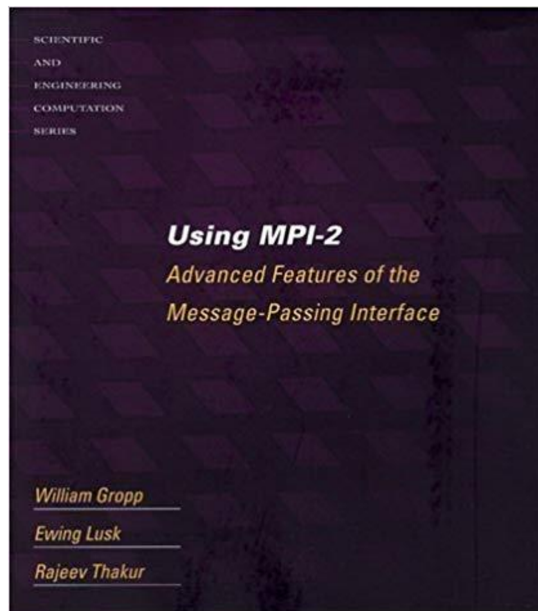
- Van der Pas & Stotzer, *Using OpenMP - The Next Step* MIT Press, 2017



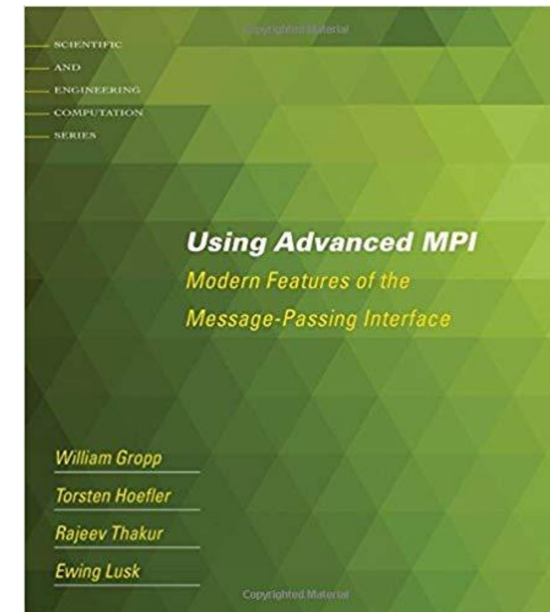
Referências

- Prog. com Memória Distribuída: MPI

- Gropp et al, *Using MPI-2* MIT Press, 2000



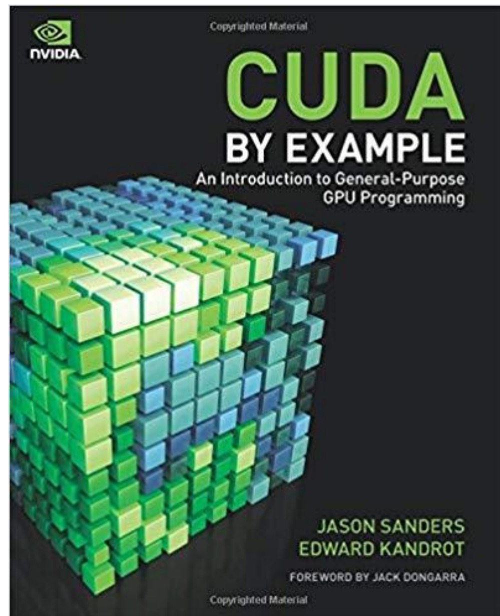
- Gropp et al, *Using Advanced MPI* MIT Press, 2014



Referências

- Prog. de Aceleradores: CUDA, OpenACC, OpenMP

  - Sanders & Kandrot, *CUDA by Example* - NVIDIA, 2011



  - Rob Farber, *Parallel Programming with OpenACC* Morgan Kaufmann, 2016

Cursos na Região

- INPE - CAP
  - CAP-372: Proc. Alto Desempenho (Dr. Stephan Stephany)
  - CAP-396: Progr. Sist. Massivamente Paralelos (Dr. Celso Mendes)
- ITA- Div.Computação
  - CE-265: Processamento Paralelo (Dr. Jairo Panetta)
- Unifesp – PPG-CC
  - Processamento de Alto Desempenho (Dr. Álvaro Fazenda)