

Programação Paralela Avançada



Programação Paralela Avançada

Introdução Geral

Programação Paralela é crítica

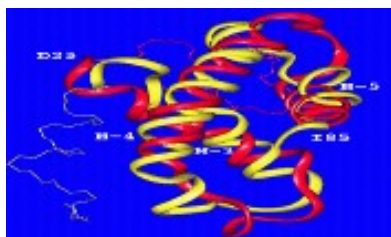


A Race for Flops...

Since 2001, Japan's supercomputers have left the U.S. in the dust

SUPERCOMPUTER	PEAK SPEED IN TERAFLIPS**	LOCATION
MOLECULAR DYNAMICS MACHINE*	78	Institute of Physical & Chemical Research (Riken)
GRAPE-6*	64	University of Tokyo
NEC EARTH SIMULATOR	41	Earth Simulator Center
HP ASCI Q	20.5	Los Alamos National Lab
CALIFORNIA DIGITAL CLUSTER*	20	Lawrence Livermore National Lab
APPLE CLUSTER	17.6	Virginia Tech
DELL CLUSTER	15	National Center for Supercomputing Applications
IBM ASCI WHITE	12	Lawrence Livermore National Lab
HP CLUSTER	11.6	Pacific Northwest National Lab
LINUX NETWORKX CLUSTER	11	Los Alamos National Lab

*Not ranked on the latest Top500 Supercomputer Sites. Riken's system was installed in 2001; University of Tokyo's in 2002. **Theoretical top speed in trillions of floating-point operations per second. Data: Riken, California Digital Corp., www.top500.org

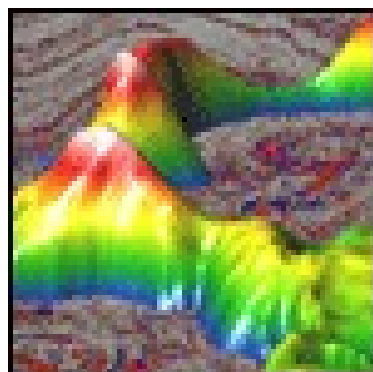
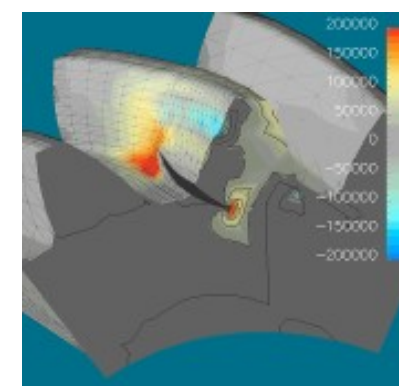
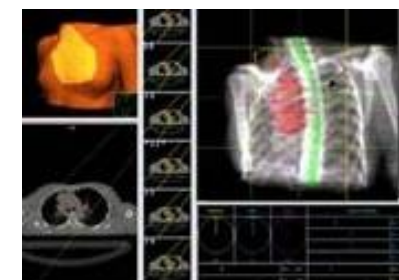


...And Tera is Just the Beginning

Power needed to complete most complex task in one week

TASK	PEAK SPEED
AUTOMOTIVE DEVELOPMENT	100 TERAFLIPS
HUMAN VISION SIMULATION	100 TERAFLIPS
AERODYNAMIC ANALYSES	1 PETAFLIPS
LASER OPTICS	10 PETAFLIPS
MOLECULAR DYNAMICS IN BIOLOGY	20 PETAFLIPS
AERODYNAMIC DESIGN	1 EXAFLIPS
COMPUTATIONAL COSMOLOGY	10 EXAFLIPS
TURBULENCE IN PHYSICS	100 EXAFLIPS
COMPUTATIONAL CHEMISTRY	1 ZETTAFLIP

(Note: A teraflops is a trillion floating-point operations per second. Peta and exa are 1,000 times faster and 1 million times faster. A zettaflops is 1 followed by 21 zeros calculations per second.) Data: NSF, Energy Dept., NASA



Programação Paralela é difícil



Em nível **conceitual** + em nível **técnico**.

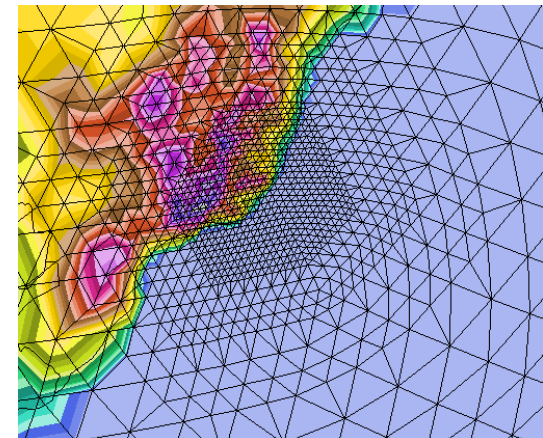
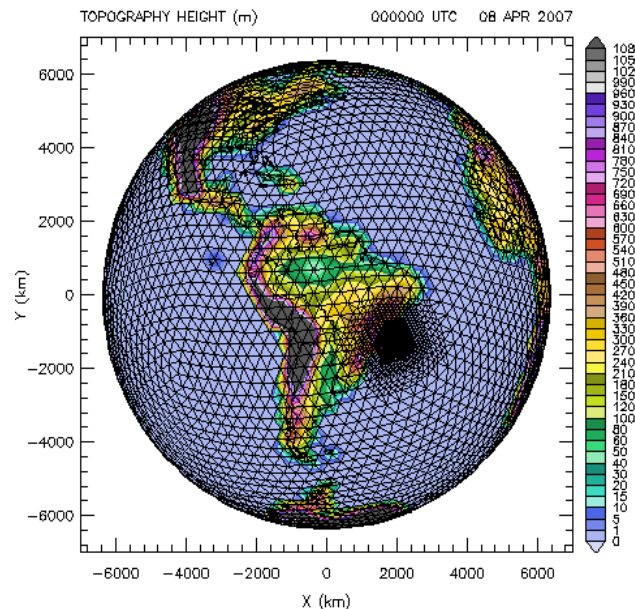
Não sub-estimar os usuários.

“Somos Especialistas e o Mundo precisa de Nós”... Será?

Vejam o caso do famoso “físico” que precisa “acelerar seu código”...

- ◆ Ou ele compra um PC 4-cores, usa OpenMP, e pronto.
- ◆ Ou ele tem R\$ 1 M, compra um Cluster+ TODO O SUPORTE... E ele pagará o necessário para que funcione.

MORAL: nos 2 casos, a paralelização “simples” JÁ terá sido feita.



Comparação com linguagem natural

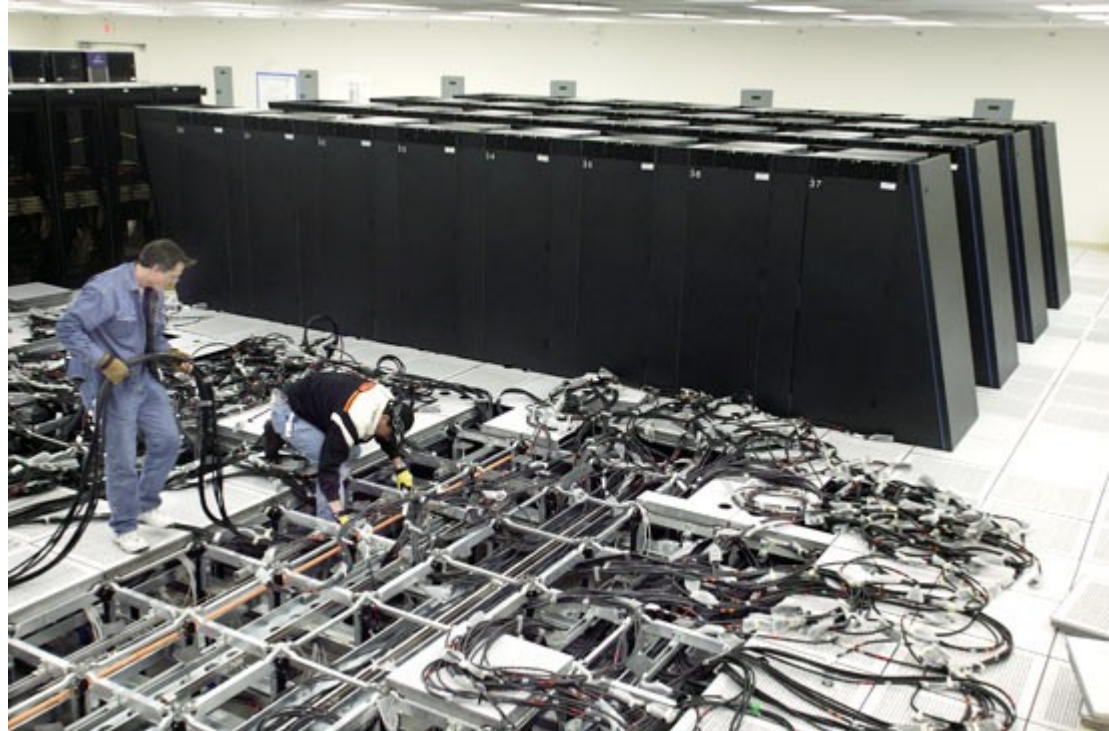
Aprender a “escrever programas paralelos” se compara com aprender uma linguagem estrangeira.

- ◆ Trabalha-se as técnicas básicas (sons, palavras simples...)
- ◆ Depois se trabalha o complicado (registros de língua...)
- ◆ Por fim, consegue-se atacar “conceitos” (poemas, literatura...)



Objetivo destes cursos

- ◆ Técnicas básicas (MPI e OpenMP)
 - ◆ ver o cursos do Profs. Rebonnatto.
- ◆ Nós vamos apresentar **técnicas avançadas**
 - ◆ Relacioná-las com conceitos de programação paralela.
 - ◆ Relacioná-las com aplicações.



4 partes sobre MPI

1) Comunicações não-bloqueantes

Conceito: efetuar comunicações durante os cálculos

2) Comunicadores

Conceito: “namespaces” para comunicações globais

3) Tipos de dados do usuário

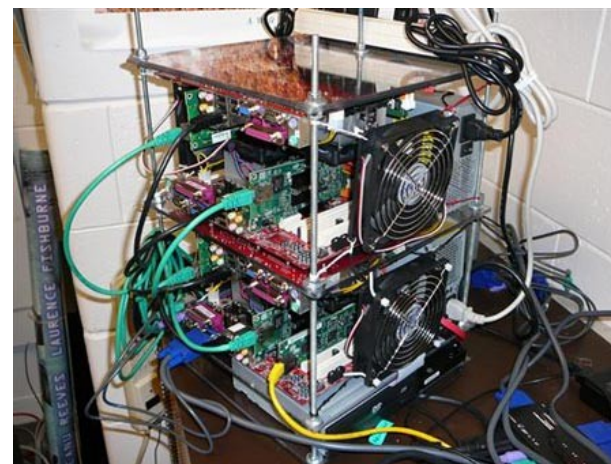
Conceito: abstração dos dados

4) Spawn

Conceito: paralelismo de tarefas

OpenMP

- ◆ Revisão de OMP básico
 - ◆ A diretiva `schedule()` - Conceito: balanceamento de carga e localidade.
- ◆ OMP avançado (OpenMP 3.0)
 - ◆ Conceito: paralelismo de tarefas.
- ◆ OMP + MPI
 - ◆ Conceito: programação paralela híbrida – Demonstrações



Comunicações não-bloqueantes



Conceito

O conceito atrás da técnica é a cobertura (overlap) comunicações / cálculos

Vejam Correios vs. Telefone:

- ◆ O telefonema necessita sincronização dos dois lados (Send/Recv normal)
- ◆ Enquanto uma carta está encaminhada, pode-se fazer outras coisas.



Send/Recv normal com MPI

```
void main() {  
    int p, r, tag = 103;  
    MPI_Status stat;  
    double val ;  
    MPI_Init(&argc, &argv) ;  
    MPI_Comm_rank(MPI_COMM_WORLD, &r);  
    if (r==0) {  
        printf("Processor 0 sends a message to 1\n");  
        val = 3.14 ;  
        MPI_Send(&val, 1, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD); }  
    else {  
        printf("Processor 1 receives a message from 0\n");  
        MPI_Recv(&val, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD, &stat);  
        printf("I received the value: %.2lf\n", val); }  
    MPI_Finalize() ;  
}
```

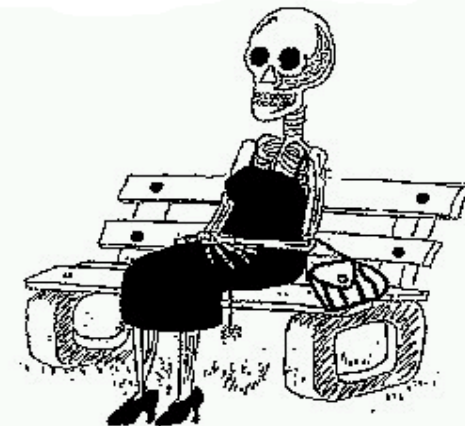
Send vs Isend, Recv vs. IRecv

- ◆ `MPI_Recv(&x,...)` é bloqueante.
 - ◆ Quando se executa a próxima instrução, `x` já tem o valor correto
- ◆ `MPI_Send(&x,...)` é não-bloqueante
 - ◆ Porém, `x` é bufferizado, e por isso a próxima instrução pode alterar `x` sem problema.
 - ◆ Obs.: `Send` é não-bloqueante... Até estourar o buffer interno!
- ◆ Variações não-bloqueantes: `MPI_Irecv()` e `MPI_Isend()`
 - ◆ Mesmos argumentos como `Recv/Send`, com um extra de tipo `MPI_Request`.
 - ◆ O `MPI_Request` possibilita testar a finalização de comunicações não-bloqueantes.
- ◆ Existe também versões explicitamente bufferizadas do `Send/Recv`: `MPI_Bsend()`.

Testar e esperar por com. não bloqueantes

- ◆ **MPI_Test**(MPI_Request* req, int* flag, MPI_Status* stat)
 - ◆ Seta 'flag' a 0 ou 1, em função do termino da com não-bloqueante descrita por 'req'.
 - ◆ Deve-se testar 'flag' depois...
 - ◆ 'status' tem a mesma semântica como no Recv.
 - ◆ Informa remetente e tag da mensagem.
- ◆ **MPI_Wait**(MPI_Request* req, MPI_Status* stat)
 - ◆ Bloqueia até a comunicação não-bloqueante tenha terminado.
 - ◆ MPI_Irecv seguido de MPI_Wait == MPI_Recv.

Waiting...



Testar e esperar por com. não bloqueantes

- ◆ Prática (muito) comum: sobreposição computação / comunicação
 - ◆ Dispara uma comunicação não-bloqueante (e.g recv),
 - ◆ (em um laço) executa todo o que se pode fazer sem ter recebido os dados, testando a recepção.
 - ◆ Se o laço acabar, bloqueia com Wait.
- ◆ Serve quando se pode executar um laço que “leva tempo” enquanto a comunicação inicial “anda”.

Irecv, Test e Wait em laços

```
MPI_Request req; MPI_Status status;

int flag;

MPI_Irecv(&dado, /* vários parâmetros */, &req);

while ( ! okay) { /* poderia ser qualquer outro tipo de laço */
    MPI_Test(req, &chegou, &status);
    if (chegou) { /* faça o que deve fazer com 'dado' */ }
    /* Faça qualquer outra coisa até 'okay' passar a valer 'true' */
}

/* Agora se precisa de 'dados' para valer
MPI_Wait(req, &status); /* Bloqueia aqui ! */
/* faça o que deve fazer com 'dado' */
```

Desafio: como otimizar um mestre/escravo?

- ◆ Idéia 1: o mestre manda uma tarefa para um escravo e **espera seu retorno**.
 - ◆ É uma forma original de escrever com MPI um programa sequencial...
- ◆ Idéia 2: o mestre manda uma tarefa para um escravo e espera o **retorno de qualquer um** deles.
 - ◆ Já melhora muito...
 - ◆ Usa o 'status' de MPI_Recv.
- ◆ Idéia 3: o mestre manda uma tarefa para cada escravo. **Enquanto um retorno está vindo**, ele processa tarefas também.
 - ◆ Aí sim se obtém 100% do paralelismo!
 - ◆ Obs: os escravos podem fazer a mesma coisa!

Time for a break

```
MPI_Bcast("Intervalo - MERENDA!");
```



```
MPI_Wait("todo o mundo voltar");
```

Comunicadores



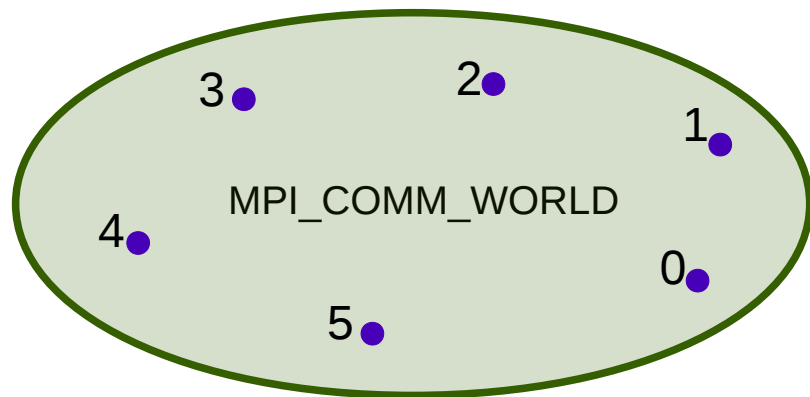
Conceito: “namespace”

- ◆ Precisa-se:
 - ◆ Evitar conflitos entre os “nomes” (de processos ou de mensagens),
 - ◆ Particionar as comunicações entre os processos,
 - ◆ Possibilitar comunicações coletivas entre sub-conjuntos de processos.
- ◆ Exemplo: e-mails: pode-se mandar mails de pessoa-a-pessoa, ou entre grupo de pessoas, e pessoas podem pertencer a mais de um grupo.
- ◆ Como fazer isso no MPI?
 - ◆ Como efetuar um Broadcast entre apenas parte dos processos que executam?



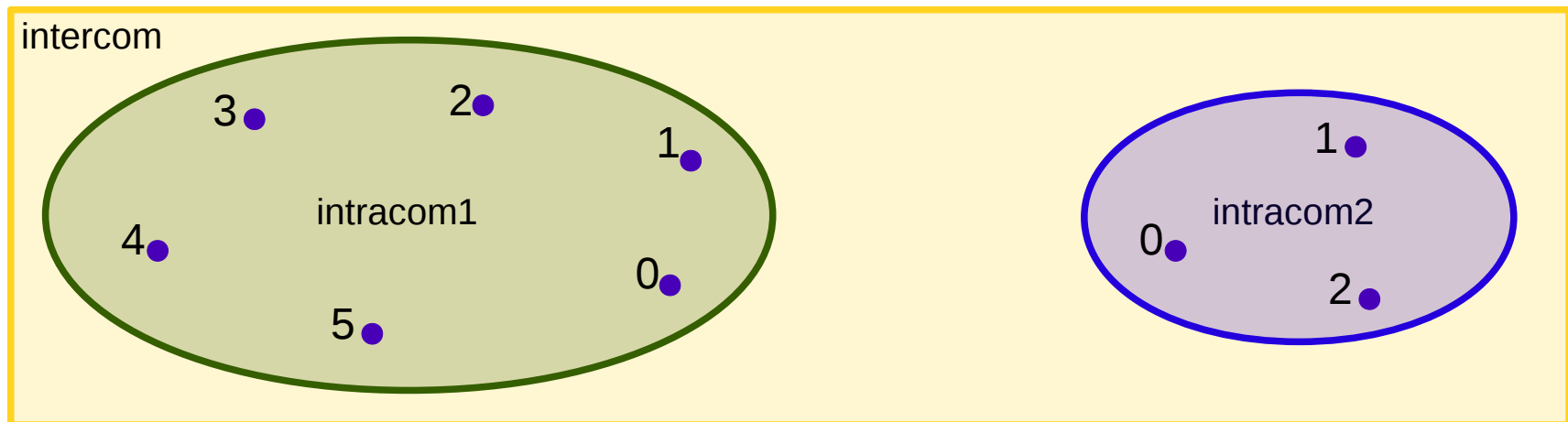
Processos MPI - grupos e comunicadores

- ◆ Cada processo tem um par (rank, grupo de processos)
- ◆ **Comunicador** é uma estrutura que define um grupo e um contexto
- ◆ Contexto = universo de comunicação
- ◆ MPI_Init → `MPI_COMM_WORLD`
- ◆ Como faço para facilitar a programação de aplicações onde nem todos os processos precisam interagir?



Intracomunicadores e Intercomunicadores

- ♦ **Intracomunicador** – mensagens dentro de um mesmo grupo local
 - ♦ MPI_COMM_WORLD é um intracomunicador
- ♦ **Intercomunicador** – mensagens entre um grupo local e remoto

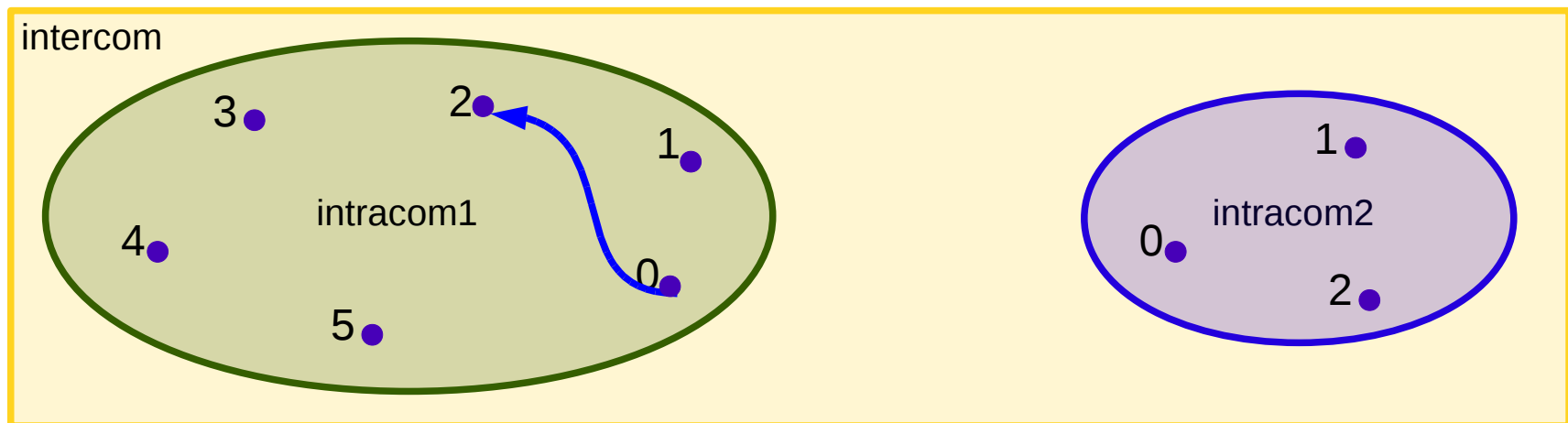


Intracomunicadores e Intercomunicadores

◆ Comunicação local

```
MPI_Send(&i, 1, MPI_INT, 2, tag, intracom1);
```

```
MPI_Recv(&i, 1, MPI_INT, 0, tag, intracom1, &st);
```



Intracomunicadores e Intercomunicadores

◆ Comunicação local

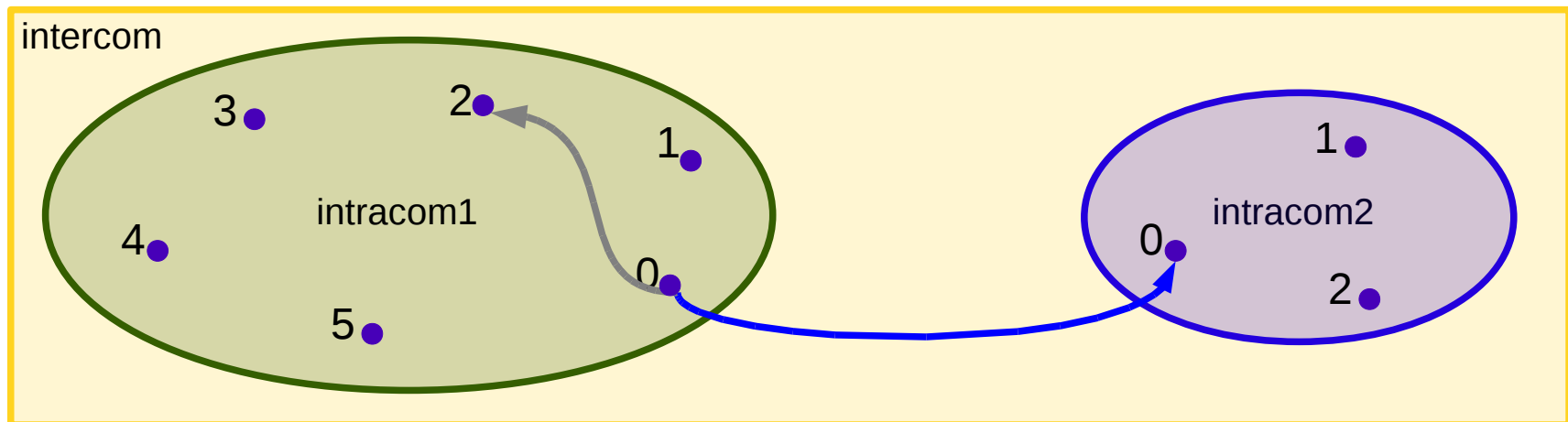
```
MPI_Send(&i, 1, MPI_INT, 2, tag, intracom1);
```

```
MPI_Recv(&i, 1, MPI_INT, 0, tag, intracom1, &st);
```

◆ Comunicação entre grupo local e remoto

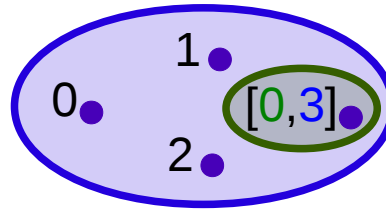
```
MPI_Send(&j, 1, MPI_INT, 0, tag, intercom);
```

```
MPI_Recv(&j, 1, MPI_INT, 0, tag, intercom, &st);
```



Manipulação de comunicadores

- ◆ `MPI_COMM_SELF`



- ◆ Uso do `MPI_Comm_split`: particionamento de comunicadores

- ◆ **Demonstração:** código fonte + execução

- ◆ Manipulação de comunicadores

- ◆ `MPI_Comm_split`, `MPI_Intercomm_create`, `MPI_Intercomm_merge`

- ◆ **Demonstração:** código fonte + execução

Tipos de Dados do Usuário



Conceito: Estrutura de Dados

Aplicativos Locais Sistema 28 °C Dom 14 Mar, 18:38

passo fundo,rs - Google Maps - Mozilla Firefox

Arquivo Editar Exibir Histórico Favoritos Ferramentas Ajuda

http://maps.google.com.br/ livermore

Home Google Agenda Gmail Moodle WIE 2010 orç Webmail - Uni Minha Unisinos RBIE CEIE PRAIA Google Tradutor Cambridge Dictiona...

Carregando... Google Agenda Resultados da Pesquisa de i... 10ª Escola Regional de Alto ... passo fundo,rs - Google Maps

Web Imagens Vídeos Mapas Notícias Orkut Gmail mais ▼ nico.maillard@gmail.com Meu perfil Novidade! Minha conta Ajuda Sair

Google maps Brasil passo fundo,rs Pesquisar no Mapa Mostrar opções de pesquisa

Localize empresas, endereços e locais de seu interesse.

Imprimir Enviar Link

Mais... Mapa Satélite Terreno

Imagem paralelizar com MPI o GoogleMaps
necessita mandar (via rede) mapas, fotos, etc...
Como mandar tais dados com o MPI?

1 km 2000 pés

passo fundo,rs - Goog...

©2010 Google - Dados cartográficos ©2010 MapLink/TeleAtlas - Termos de Uso

Area de Trabalho 1 Area de Trabalho 2 agenda
Desk 3 www email

Datatypes do MPI

- ◆ Os MPI Datatypes básicos para as comunicações são MPI_INT, MPI_FLOAT, MPI_CHAR, MPI_DOUBLE...
 - ◆ Mais qualquer Linguagem de Programação sequencial provê tipos abstratos compostos (structs, union), ponteiros, etc...
- ◆ Como **comunicar tipos de dados abstratos**?

Pode-se usar o “método naïve”:

- 1) memcpy() toda a estrutura de dados num buffer de Bytes,
- 2) recupera seu tamanho e
- 3) manda-o como MPI_CHAR.

Pouco elegante, e nem um pouco portátil.

- ◆ Obs: no MPI, receber uma mensagem implica em **conhecer** seu tamanho.

1a solução mais elegante: MPI_Pack()

- ◆ Usa o MPI para empacotar os dados:
 - ◆ **MPI_Pack**(void* inbuf, int count, MPI_Datatype dtype, void* outbuf, int outcount, int* position, MPI_Comm comm)
 - ◆ Empacota os dados em 'inbuf' (= 'count' itens de tipo básico 'dtype'), retorna 'outbuf' e atualiza 'position' para outras chamadas a MPI_Pack().
- ◆ Manda o buffer com tipo básico **MPI_PACK**:
 - ◆ MPI_Send(outbuf, MPI_Pack_size(outbuf), MPI_PACK, . . .)
- ◆ Usa **MPI_Unpack**(. . .) depois do MPI_Recv para desempacotar o buffer.

2a solução mais elegante

- ◆ “Descreve” ao MPI como serializar sua estrutura de dados.
- ◆ Efetua um “commit” para criar um novo MPI Data_type personalizado,
- ◆ Usa o Data_type como se fosse um Datatype MPI básico, nas mensagens.
- ◆ Exemplo: mandar uma struct de 2 campos:
 - ◆ Usa-se `MPI_Type_create_struct`(2, VetorTam, VetorDesloc, VetorTipo, MPI_Datatype* meu_tipo),
 - ◆ VetorTam: vetor de 2 entradas para indicar o número de entradas em cada campo,
 - ◆ VetorDesloc: vetor de 2 entradas para indicar o endereço inicial de cada campo,
 - ◆ VetorTipo: vetor de 2 entradas para indicar o tipo (básico) de cada campo,
 - ◆ meu_tipo: novo Datatype retornado.
 - ◆ Mais `MPI_type_commit()`.

Datatypes: vantagens e inconvenientes

Pros:

- ◆ Evita a duplicação de cópias dos dados para mandá-los,
- ◆ Aumenta a portabilidade do código,
- ◆ Funciona bem para tipos compostos.



Contras:

- ◆ Trabalhoso!!!
 - ◆ Comparar com mecanismos de serialização automática de C# ou Java...
- ◆ Não resolve o caso de ponteiros.

Fim do 1o. encontro



Nicolas Maillard:
nicolas@inf.ufrgs.br

Márcia C. Cera:
marcia.cera@inf.ufrgs.br

Resumo da Obra



Lembrando as técnicas já apresentadas

- ◆ Comunicações não bloqueantes
 - ◆ Sobreposição comunicações/cálculos
- ◆ Comunicadores
 - ◆ Definindo “namespaces” para comunicação
- ◆ Tipos de dados do usuário
 - ◆ Abstração dos dados (estruturas de dados)
- ◆ Aplicações reais **mesclam** essas técnicas para prover um **melhor desempenho** e utilização das arquiteturas paralelas

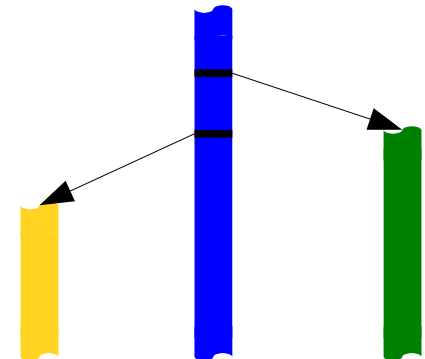


Criação Dinâmica de Processos



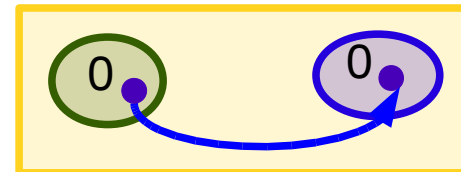
Processos dinâmicos

- ◆ Em tempo de execução, um processo pode lançar a criação de outro
 - ◆ Modelo de execução dinâmico
 - ◆ Processos podem ser criados conforme a demanda da aplicação
- ◆ Norma MPI-2: Modelo SPMD → MPMD
- ◆ Possibilita a implementação de novas classes de aplicações
 - ◆ **Maleáveis**: capazes de se adaptar a dinamicidade de recursos
 - ◆ **Evolutivas**: capazes de se adaptar a irregularidade de sua carga de trabalho
 - ◆ **Paralelismo de Tarefas** (*Task Parallelism*)



Processos dinâmicos

- ◆ Cada processo possui seu **intracomunicador**
- ◆ O processo que dispara a criação é chamado de **pai**
- ◆ O processo(s) criado(s) é(são) o(s) **filho(s)**
- ◆ Comunicação entre pai e filhos acontece através de um **intercomunicador**
 - ◆ Do lado do pai:
 - ◆ `MPI_Comm_spawn`
 - ◆ Do lado do(s) filho(s):
 - ◆ `MPI_Comm_get_parent`



Criando processos dinâmicos

```
int MPI_Comm_spawn(
```

```
    char *command,
```

--> nome do binário do novo processo

```
    char *argv[],
```

--> argumentos para o novo processo

```
    int maxprocs,
```

--> número de processos a serem criados

```
    MPI_Info info,
```

--> informações para runtime system

```
    int root,
```

--> identificador do processo pai

```
    MPI_Comm comm,
```

--> intracomunicador do processo pai

```
    MPI_Comm *intercomm,
```

--> intercomunicador: pai e filho

```
    int array_of_errcodes[]
```

--> vetor de erros

```
)
```

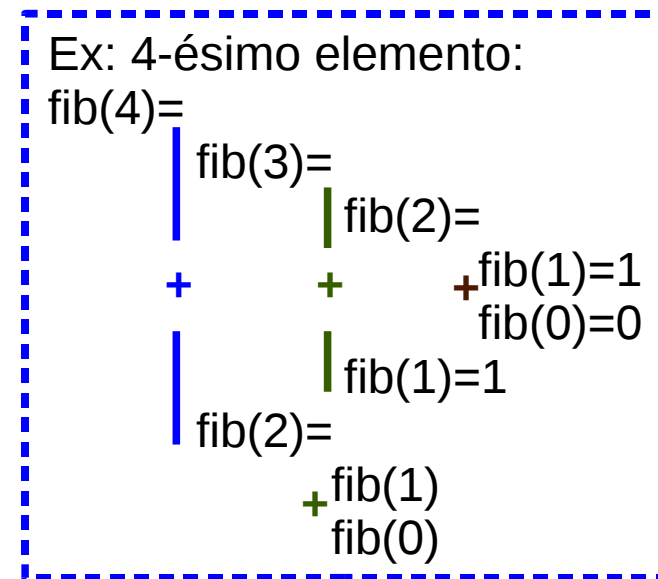
- ◆ **Demonstração:** pai-filho com troca de mensagens

Exemplificando

- ◆ Cálculo do *i*-ésimo elemento da sequência de Fibonacci
 - ◆ (0, 1, 1, 2, 3, 5, 8, 13, 21,...)
 - ◆ Um elemento é a soma de seus 2 antecessores

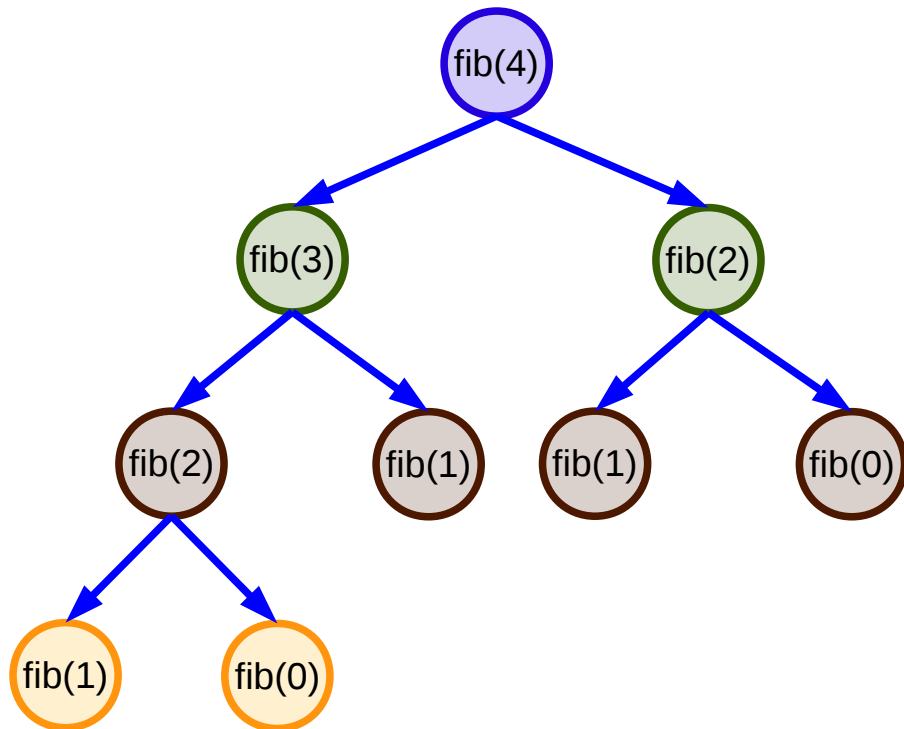
- ◆ Algoritmo Sequencial:

```
int fib (int n){  
    if (n < 2) return n;  
    else{  
        int x, y;  
        x = fib (n - 1);  
        y = fib (n - 2);  
        return (x + y);  
    }  
}
```



- ◆ Demonstração:
Fibonacci com processos dinâmicos

Exemplificando

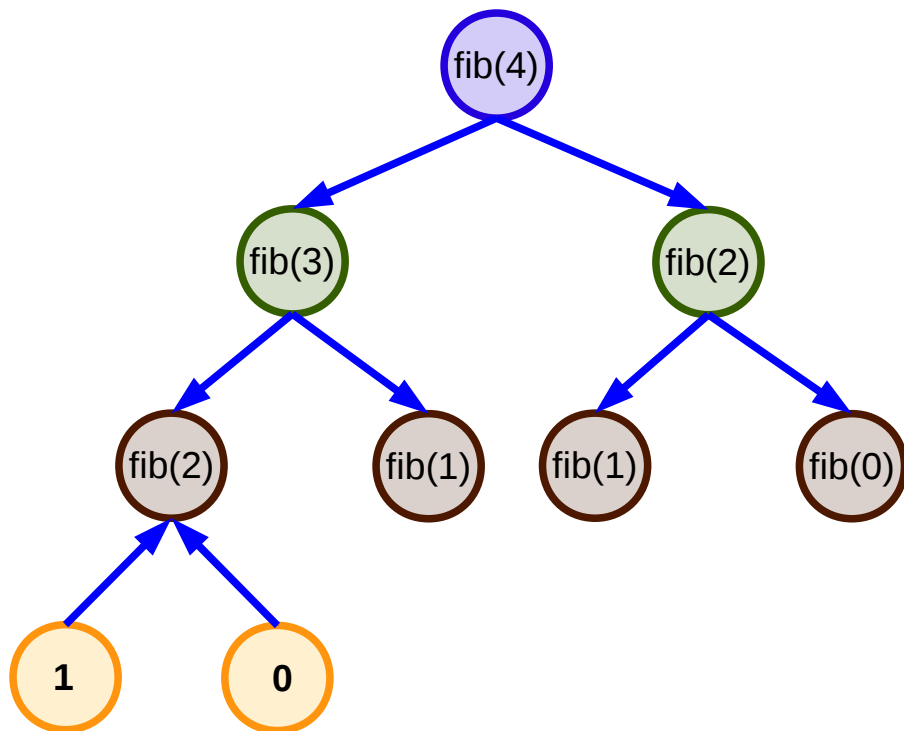


Ex: 4-ésimo elemento:

$$\begin{array}{l} \text{fib}(4) = \text{fib}(3) + \text{fib}(2) \\ \text{fib}(3) = \text{fib}(2) + \text{fib}(1) \\ \text{fib}(2) = \text{fib}(1) + \text{fib}(0) \end{array}$$

fib(1)=1
fib(0)=0

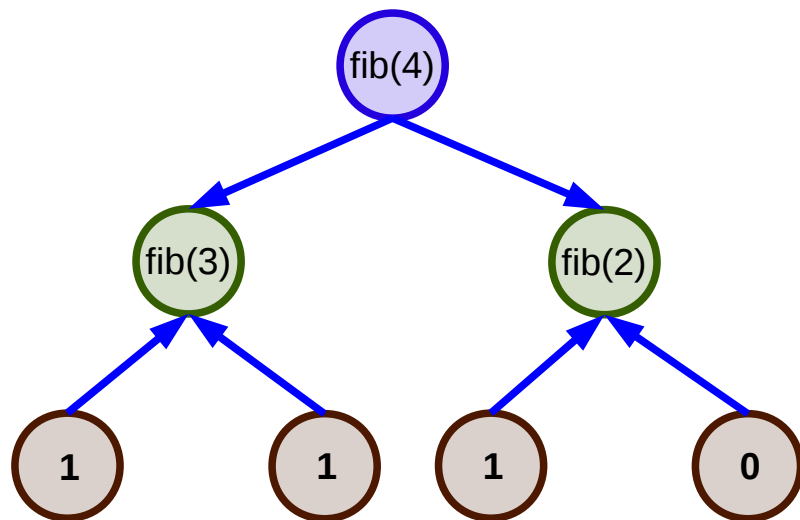
Exemplificando



Ex: 4-ésimo elemento:

$$\begin{array}{l} \text{fib}(4)= \\ \quad \left| \begin{array}{l} \text{fib}(3)= \\ + \\ \text{fib}(2)= \end{array} \right. \left| \begin{array}{l} \text{fib}(2)= \\ + \\ \text{fib}(1)=1 \\ \text{fib}(0)=0 \end{array} \right. \\ \quad \left| \begin{array}{l} \text{fib}(2)= \\ + \\ \text{fib}(1)=1 \\ \text{fib}(0)=0 \end{array} \right. \end{array}$$

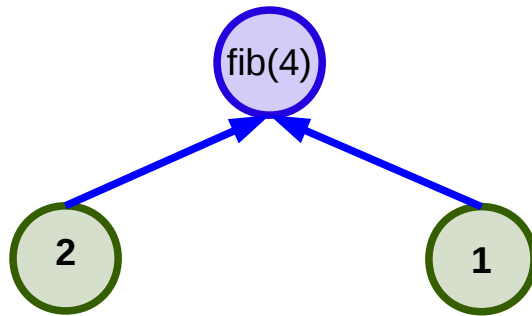
Exemplificando



Ex: 4-ésimo elemento:

$$\begin{array}{l} \text{fib}(4)= \\ \quad \text{fib}(3)= \\ \quad + \quad \text{fib}(2)= \\ \quad \quad \quad \text{fib}(1)=1 \\ \quad \quad \quad \text{fib}(0)=0 \\ \quad \text{fib}(2)= \\ \quad \quad \quad \text{fib}(1)=1 \\ \quad \quad \quad + \text{fib}(0) \end{array}$$

Exemplificando



Ex: 4-ésimo elemento:

$$\begin{aligned} \text{fib}(4) &= \text{fib}(3) + \text{fib}(2) \\ &= \text{fib}(2) + \text{fib}(1) + \text{fib}(1) + \text{fib}(0) \\ &= \text{fib}(1) + \text{fib}(0) + \text{fib}(1) + \text{fib}(0) \\ &= 1 + 0 + 1 + 0 = 2 \end{aligned}$$

Exemplificando

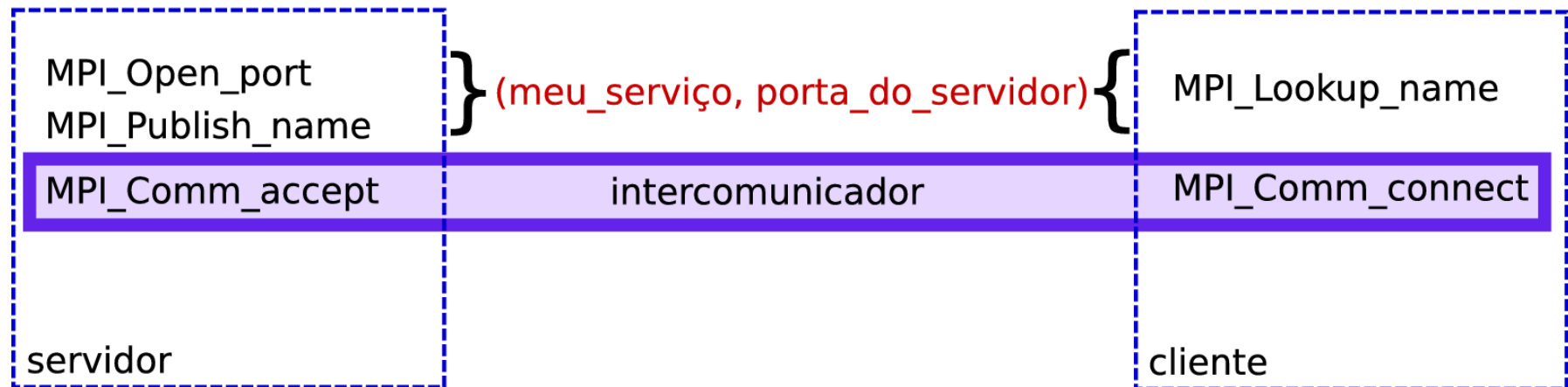
3

Ex: 4-ésimo elemento:

$$\begin{array}{l} \text{fib}(4)= \\ \quad \left| \begin{array}{l} \text{fib}(3)= \\ + \\ \text{fib}(2)= \\ + \\ \text{fib}(1)=1 \\ + \\ \text{fib}(0)=0 \end{array} \right. \\ \quad \left| \begin{array}{l} \text{fib}(2)= \\ + \\ \text{fib}(1)=1 \\ + \\ \text{fib}(0)=0 \end{array} \right. \end{array}$$

Estabelecendo Comunicação

- Comunicações seguindo a relação Cliente/Servidor



Concluindo apresentação do MPI

- ◆ MPI é o **padrão** para programação com troca de mensagens
- ◆ É largamente utilizado no desenvolvimento de aplicações de **Alto Desempenho**
- ◆ O padrão prevê **diversas técnicas avançadas** objetivando ganhos de desempenho, dentre elas nós apresentamos nesse curso:
 - ◆ Comunicações não-bloqueantes
 - ◆ Manipulação de Comunicadores
 - ◆ Definição de Tipos de Dados
 - ◆ Criação dinâmica de processos
- ◆ Desenvolver programas MPI eficientes não é uma tarefa trivial;
- ◆ Ela está diretamente relacionada ao **conhecimento e domínio** de técnicas e conceitos do MPI

Memória Compartilhada - OpenMP



Noções Básicas de OpenMP

- ◆ **OpenMP** – Open Multi-processing
- ◆ Consórcio de fabricantes: Intel, IBM, Sun, ...
 - ◆ Suportada em compiladores: Intel, gcc, ...
- ◆ É um padrão que define
 - ◆ A forma como os cálculos serão distribuídos entre
 - ◆ A forma como os dados serão replicados ou compartilhados
 - ◆ As sincronizações
- ◆ Programação baseada em pragmas e rotinas providas pela biblioteca OpenMP
 - ◆ Número de threads:
 - ◆ `OMP_NUM_THREADS` ou `omp_set_num_threads(int n)`
 - ◆ **Demonstração:** Olá mundo e como estruturar um programa OpenMP

Noções Básicas de OpenMP

- ◆ Distribuição dos cálculos e dos dados
 - ◆ Cláusula `shared(x, y, ...)` : acesso compartilhado entre as threads
 - ◆ Cláusula `private(x, y, ...)`: cria réplicas das variáveis para cada fluxo – variáveis locais de um bloco parallel são automaticamente privadas
- ◆ **Demonstrações:** entendendo variáveis compartilhadas e privadas

Noções Básicas de OpenMP

◆ Distribuição dos cálculos

◆ Sections

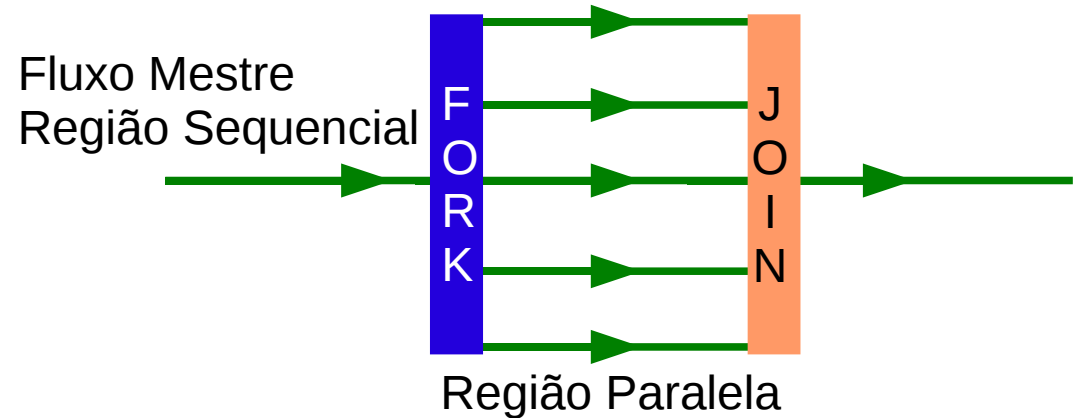
- ◆ Exemplo: Considere um programa que calcula operações sobre as N entradas de um vetor dados[10000]

```
#pragma omp sections
{
  #pragma omp section
  {    Compute(0, N, dados)  }
  #pragma omp section
  {    Compute(1, N, dados)  }
  #pragma omp section
  {    Compute(2, N, dados)  }
  #pragma omp section
  {    Compute(3, N, dados)  }
}
```

```
Compute(int index, int N, int dados)
{
  Int start, end, i;
  Start = index * N/4 + 1;
  End = (index + 1) * N/4;
  // efetua cálculos sobre suas N/4
  // entradas do vetor dados
}
```

Paralelismo de Laços

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



◆ **Demonstração:** exemplo de paralelismo de laços

◆ **Restrições de acesso:**

```
int fator = 0.5
#pragma omp parallel for private( fator )
for (i = 0; i < N; i++){
    c[i] = fator * a[i];
}
```

```
#pragma omp parallel for private( j )
for (i = 0; i < N; i++){
    for(j = 0; j < N; j++){
        c[i] = a[i] + b[j];
    }
}
```

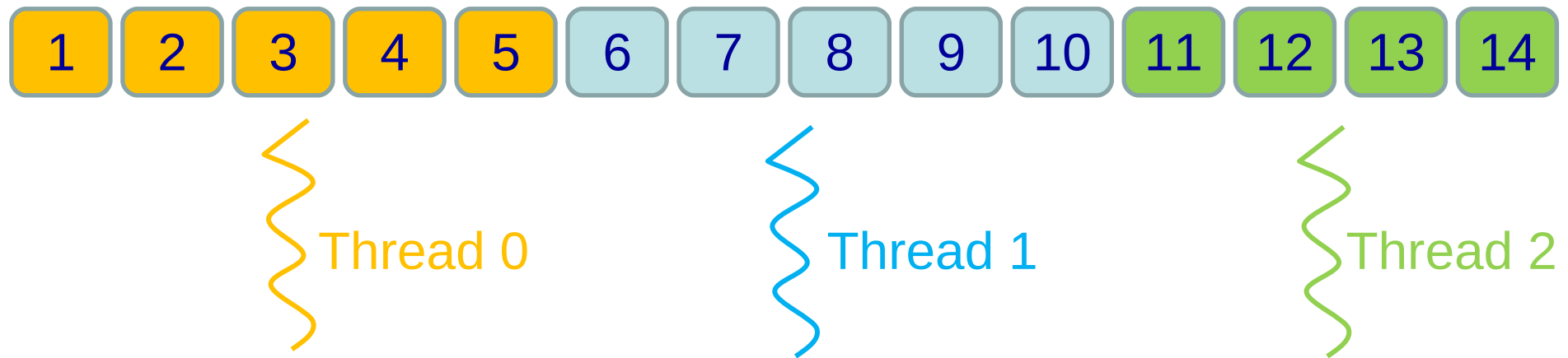
*firstprivate, lastprivate

Paralelismo de Laços

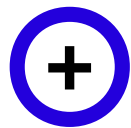


Por exemplo:
N = 14 iterações
P = 3 threads

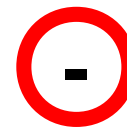
Paralelismo de Laços



Divisão das iterações em blocos

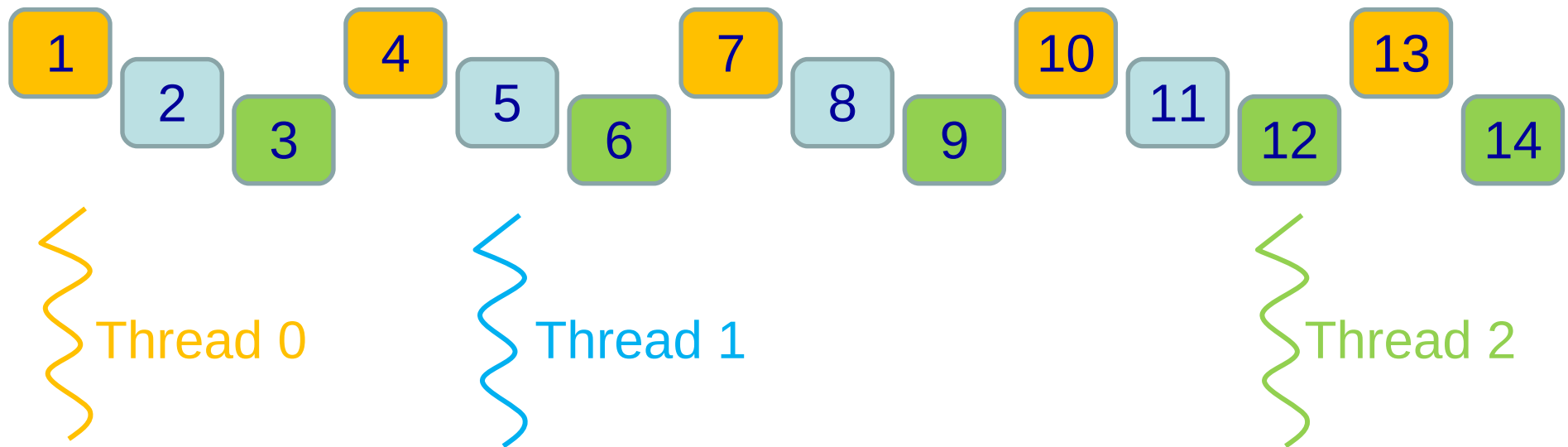


localidade

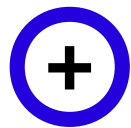


Iterações com carga irregular

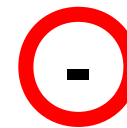
Paralelismo de Laços



Alocação **cíclica** – Round Robin

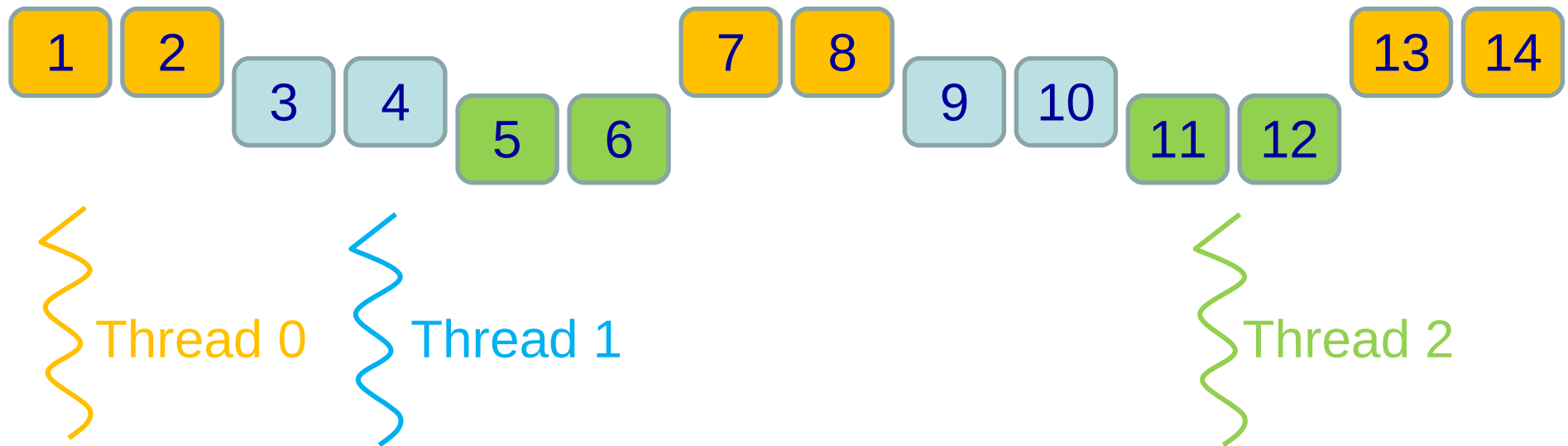


Balanceamento de carga

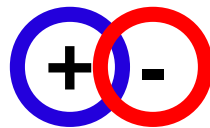


Perdeu-se a localidade

Paralelismo de Laços

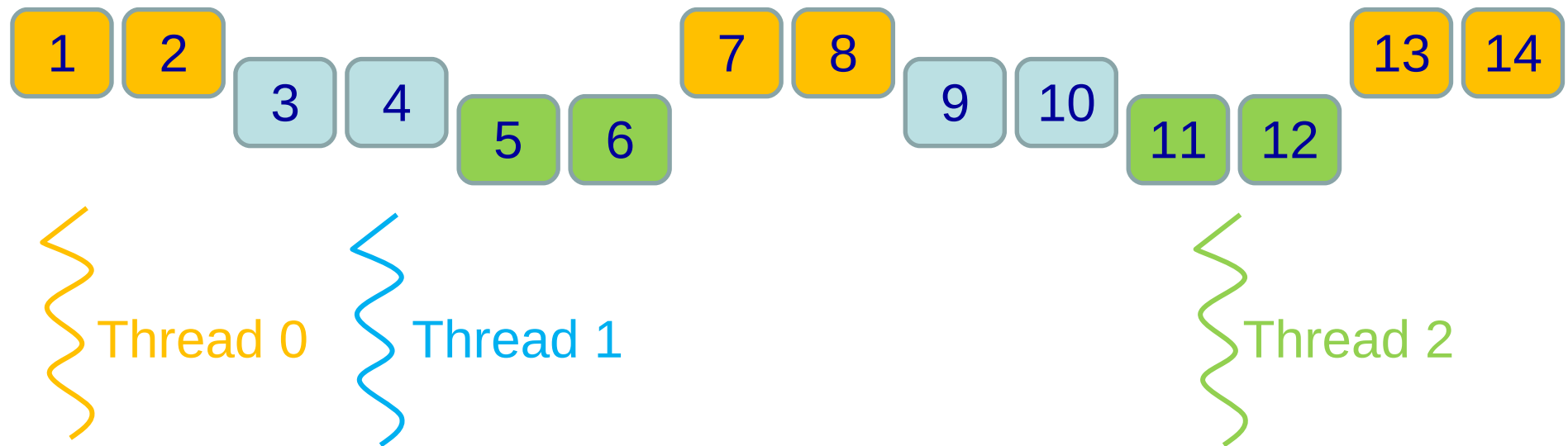


Alocação por **blocos cíclicos**



Meio termo entre Balanceamento de carga e Localidade

Paralelismo de Laços



Alocação por **blocos cíclicos**



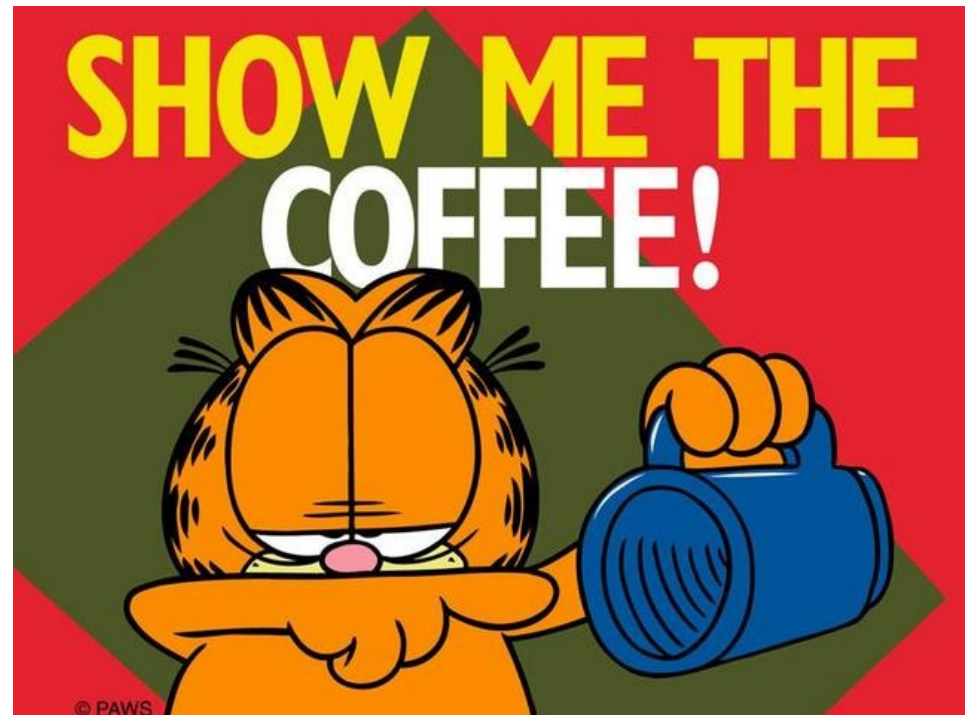
Tamanho dos Blocos

Paralelismo de Laços

- ◆ Distribuição das iterações do laço - `schedule(tipo[ , chunk])`:
 - ◆ `static`: chunk com valor estático pré-determinado
 - ◆ `dynamic`: chunk depende da quantidade restante de iterações
 - ◆ `guided`: mescla dos anteriores
- ◆ A forma como a distribuição das iterações serão distribuídas influencia diretamente na **granularidade do trabalho** das threads
 - ◆ Iterações com tempos diferentes
 - ◆ Bloco maior == melhor localidade espacial;
 - ◆ Bloco menor == melhor balanceamento de carga.

Intervalo

- ◆ Nos vemos às **16 horas** para o encerramento deste minicurso:
 - ◆ **OpenMP Avançado**
 - ◆ Paralelização de um laço while
 - ◆ Paralelismo de tarefas no OpenMP 3.0
 - ◆ Mesclando **OpenMP e MPI** visando clusters multi-core
 - ◆ Considerações finais



OMP Avançado



Paralelizar um while ()

```
int main() {  
    int tid, i;  
  
    {  
        tid = omp_get_thread_num();  
        i = next_indice();  
        while (i != -1) {  
            calculo(tid,i);  
            i = next_indice();  
        }  
    }  
}
```

```
int indice=0;  
int next_indice() {  
    int res;  
    {  
        if (indice == N)  
            res = -1;  
        else  
            res = ++indice;  
    }  
    return(res);  
}
```

Como paralelizar este laço while com OpenMP?

Paralelizar um while ()

```
int main() {  
    int tid, i;  
    #pragma omp parallel private(tid,i)  
    {  
        tid = omp_get_thread_num();  
        i = next_indice();  
        while (i != -1) {  
            calculo(tid,i);  
            i = next_indice();  
        }  
    }  
}
```

```
int indice=0;  
int next_indice() {  
    int res;  
    {  
        if (indice == N)  
            res = -1;  
        else  
            res = ++indice;  
    }  
    return(res);  
}
```

Como paralelizar este laço while com OpenMP?

Paralelizar um while ()

```
int main() {  
    int tid, i;  
    #pragma omp parallel private(tid,i)  
    {  
        tid = omp_get_thread_num();  
        i = next_indice();  
        while (i != -1) {  
            calculo(tid,i);  
            i = next_indice();  
        }  
    }  
}
```

```
int indice=0;  
int next_indice() {  
    int res;  
    {  
        if (indice == N)  
            res = -1;  
        else  
            res = ++indice;  
    }  
    return(res);  
}
```

PROBLEMA de acesso concorrente a next_indice()

Paralelizar um while ()

```
int main() {  
    int tid, i;  
    #pragma omp parallel private(tid,i)  
    {  
        tid = omp_get_thread_num();  
        i = next_indice();  
        while (i != -1) {  
            calculo(tid,i);  
            i = next_indice();  
        }  
    }  
}
```

```
int indice=0;  
int next_indice() {  
    int res;  
    #pragma omp critical  
    {  
        if (indice == N)  
            res = -1;  
        else  
            res = ++indice;  
    }  
    return(res);  
}
```

PROBLEMA resolvido.

OpenMP 3.0

- ◆ OpenMP 3 introduz uma nova forma de paralelização: o paralelismo de tarefas
 - ◆ ver Spawn com MPI.
 - ◆ Adequado para algoritmos recursivos.
- ◆ Uma tarefa OpenMP é definida pelo pragma “#pragma omp task”
- ◆ Pode-se sincronizar as tarefas com um “#pragma omp taskwait”

```
int fib ( int n )  
{  
  int x,y;  
  if ( n < 2 ) return n;  
  
  x = fib(n-1);  
  
  y = fib(n-2);  
  
  return x+y;  
}
```

```
int fib ( int n )  
{  
  int x,y;  
  if ( n < 2 ) return n;  
  #pragma omp task shared(x)  
  x = fib(n-1);  
  #pragma omp task shared(y)  
  y = fib(n-2);  
  #pragma omp taskwait  
  return x+y;  
}
```

De novo o while(), com OpenMP3

- ◆ Construção típica do while() com uma pilha...
- ◆ No caso, usa-se uma pilha para varrer uma árvore binária em profundidade.

```
stack_t pilha = init_pilha();
node_t node, root = init_tree();

push(pilha, root);
while (! is_empty(pilha) ) {
    node = pop(pilha);
    if (is_leaf(node))
        compute(node);
    else {
        push(pilha, node->right);
        push(pilha, node->left);
    }
}
```

De novo o while(), com OpenMP3

- ◆ Construção típica do while() com uma pilha...
- ◆ No caso, usa-se uma pilha para varrer uma árvore binária em profundidade.

```
stack_t pilha = init_pilha();
node_t node, root = init_tree();

push(pilha, root);
while (! is_empty(pilha) ) {
    node = pop(pilha);
    if (is_leaf(node))
        #pragma omp task
        compute(node);
    else {
        push(pilha, node->right);
        push(pilha, node->left);
    }
}
```

De novo o while(), com OpenMP3

- ◆ Construção típica do while() com uma pilha...
- ◆ No caso, usa-se uma pilha para varrer uma árvore binária em profundidade.

```
stack_t pilha = init_pilha();
node_t node, root = init_tree();

push(pilha, root);
while (! is_empty(pilha) ) {
    node = pop(pilha);
    if (is_leaf(node))
        #pragma omp task
        compute(node);
    else {
        push(pilha, node->right);
        push(pilha, node->left);
    }
}
```

Cuidado!!!

- Funciona se “compute” não introduz efeito colateral.
- Não se garante mais a ordem de cálculos sobre os nós.

OpenMP + MPI



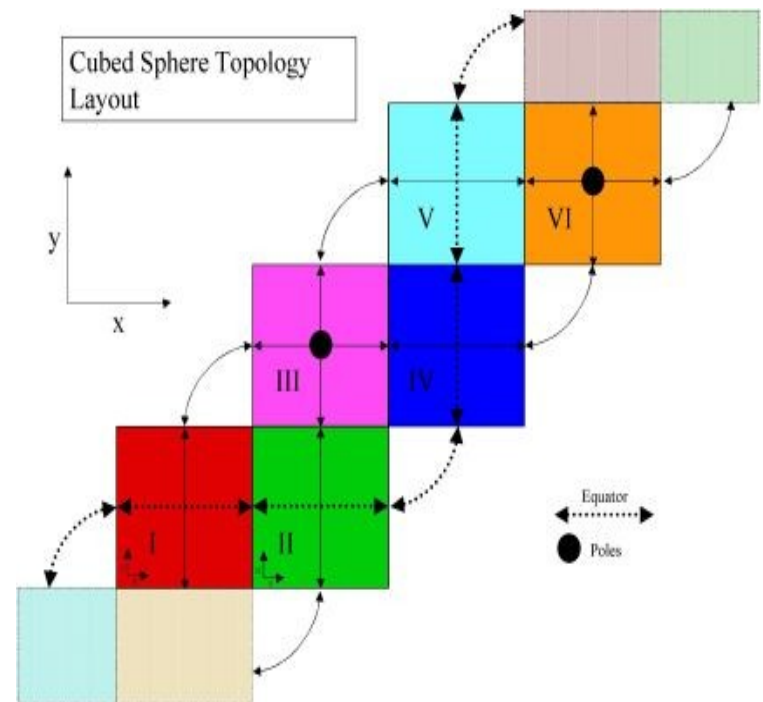
Programação Híbrida

- ◆ Fazer o que para um cluster de multicores?
 - ◆ Usar o MPI e disparar um processo por core?
 - ◆ Opção interessante – MPI como API de programação paralela?
 - ◆ Usar um OpenMP que emule uma memória compartilhada? (see Intel).
 - ◆ Fator 10x a pagar...
 - ◆ Usar MPI junto com OpenMP?
 - ◆ É técnico...



Um exemplo: decomposição de domínios

- ◆ Simula-se um fenômeno físico num domínio espacial (2D, 3D).
- ◆ A paralelização implica no “recorte” em subdomínios.
- ◆ A simulação itera (no tempo) a resolução de sistemas de equações para cada domínio,
- ◆ Há troca de informações a respeito das bordas.



Exemplo: HOMB

- ◆ Resolve uma equação de Laplace.
- ◆ Domínio 2D
- ◆ Uso de:
 - ◆ Paralelismo MPI entre os domínios (com Irecv/Isend...)
 - ◆ Paralelismo OpenMP no cálculo interno a cada domínio.
 - ◆ O cálculo envolve 2 laços aninhados.
 - ◆ Observar o uso de “critical” para calcular a norma.

Conclusão Geral



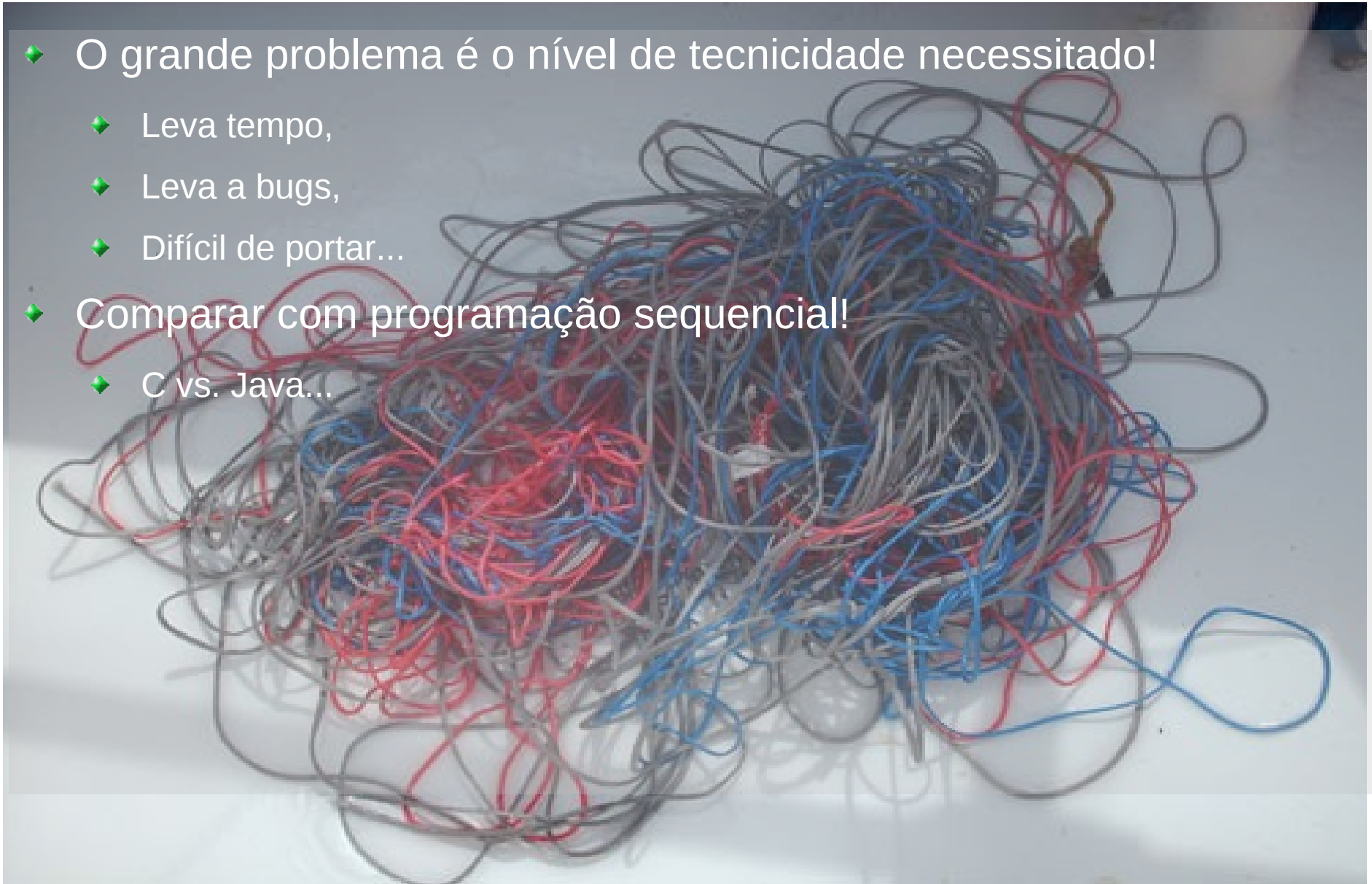
O que vimos?

- ◆ Message Passing Interface – uso avançado.
 - ◆ Programas reais usam dados complexos, precisam de dinamicidade / balanceamento de carga, executam com 10K processos, etc.
- ◆ OpenMP – do paralelismo de laços a paralelismo de tarefas.
 - ◆ Programação recursiva, granularidade fina.
- ◆ Usar os dois juntos para arquiteturas híbridas
 - ◆ Shared + Distributed memory.



What's wrong?

- ◆ O grande problema é o nível de tecnicidade necessitado!
 - ◆ Leva tempo,
 - ◆ Leva a bugs,
 - ◆ Difícil de portar...
- ◆ Comparar com programação sequencial!
 - ◆ C vs. Java...



Precisa-se de avanços em programação paralela

Precisa-se de abstrações maiores para:

- ◆ Que o programador não deva pensar na arquitetura,
- ◆ Que o programador possa definir facilmente o grau máximo de paralelismo,
- ◆ Que o middleware de execução mapeia facilmente as (pequenas) tarefas nas unidades de execução providas pelo hardware.
- ◆ Existem ferramentas recentes, ou de pesquisa, indo neste rumo
 - ◆ Intel TBB, por exemplo.
- ◆ Para memória distribuída, ainda tem pouca coisa.
- ◆ O aumento de cores, e a necessidade de Rede-On-Chip, vai levar a dever atacar este problema nos anos que vêm.

Márcia Cristina Cera, Nicolas Maillard
{mccera, nicolas}@inf.ufrgs.br

Grupo de Processamento Paralelo e Distribuído
UFRGS

