Técnicas de Computação Paralela
Modelos de Programação

José Rogado
jrogado@ulusofona.pt

Universidade Lusófona

Programa da Cadeira

1. Introdução
   - Motivações para o Paralelismo
   - Evolução e Limitações das Arquitecturas
   - Características da Programação Paralela
   - Aplicações: Engenharia, Científicas e Comerciais
2. Arquitecturas para Computação Paralela
   - Noções de Controle e Comunicação
   - Modelos de Computadores Paralelos
   - Modelos de Programação Paralela
3. Desenho de Algoritmos Paralelos
   - Introdução
   - Técnicas de Decomposição e Aglomeração
   - Mapeamento e Balanceamento de Carga
   - Comunicação Inter Tarefas
   - Minimização de Interacções
   - Modelos de Algoritmos Paralelos

Leitura pelos alunos páginas 110 a 142

Programa da Cadeira

4. Modelos de Programação Paralela
   - Programação em Memória Partilhada: Modelo de Threads
   - Programação por Mensagens: MPI - Message Passing Interface
   - Exemplos de Algoritmos e Aplicações
   - Clusters: funcionamento e desempenho
   - Grid Computing: mecanismos e aplicações
5. Quantificação do Desempenho
   - Definição de Desempenho
   - Aproximações à Modelização do Desempenho
   - Modelos e Análise de escalabilidade
   - Desenho e Experimentação
   - Avaliação de Implementações
   - Problemas de Input/Output

Programação em Memória Partilhada

- Modelos de Programação em Memória Partilhada
- Programação com Threads
  - A API Posix para threads: pthreads
  - Exemplos de utilização
  - Primitivas de Sincronização
- Programação baseada em Directivas
  - Cilk
  - OpenMP

Modelos de Programação

- Quando se pretende implementar um algoritmo paralelo numa dada plataforma, é necessário definir o **modelo de programação**
- O modelo de programação fornece o formalismo necessário para controlar o paralelismo identificado numa aplicação
  - Suporte para a concorrência
  - Suporte para a sincronização
  - Suporte para a comunicação
- Num modelo de programação em memória partilhada, a comunicação é implicitamente assegurada
- Assim, o modelo de programação em memória partilhada foca essencialmente
  - Expressão da concorrência
  - Mecanismos de Sincronização
  - Minimização das interacções e overhead

Programação em Memória Partilhada

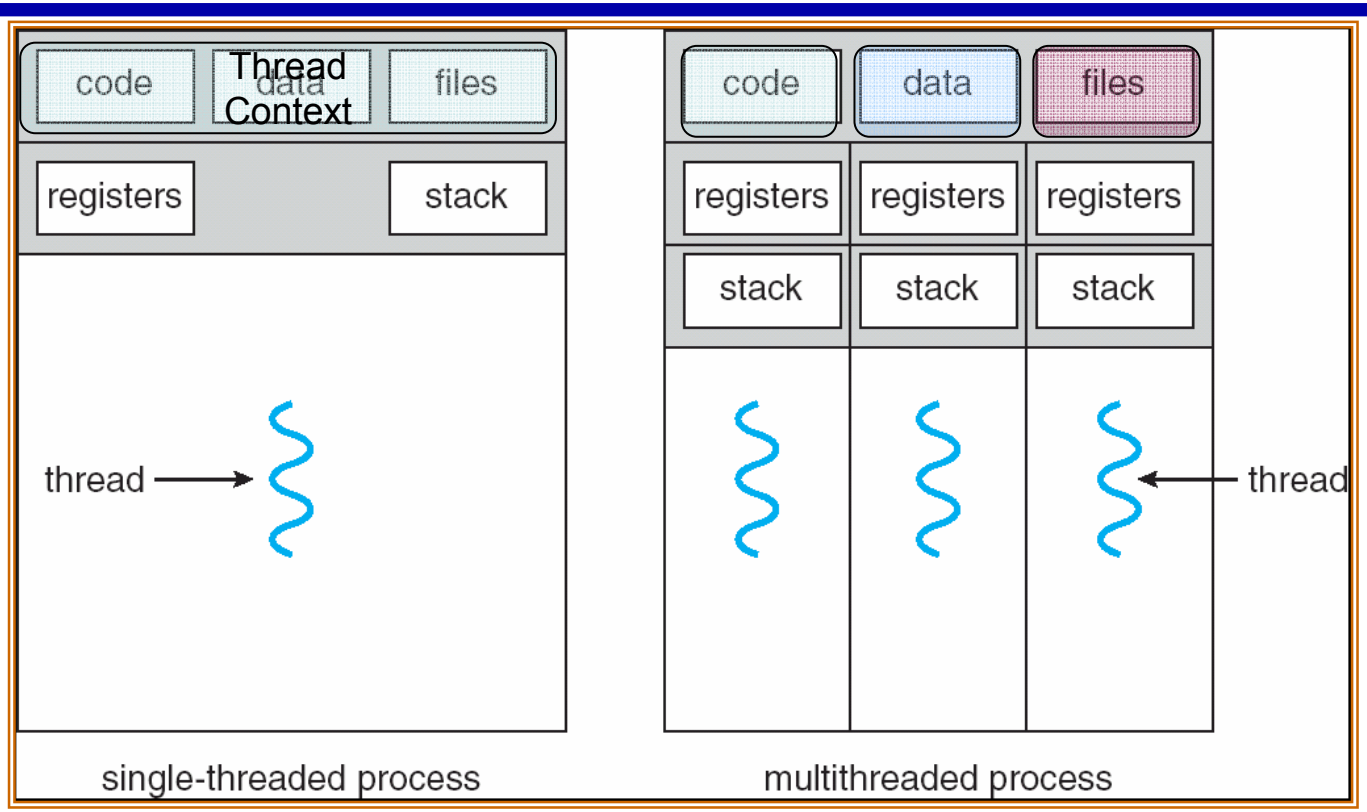
- Os modelos de programação em memória partilhada podem variar na forma de fornecer:
  - Modelo de concorrência
  - Partilha de dados
  - Suporte da sincronização
- Na maioria das plataformas o modelo de processos permite uma primeira aproximação ao modelo paralelo
- Geralmente os processos são entidades estanques
  - Não é possível partilhar dados em memória de forma simples
  - A sincronização entre processos é por vezes rudimentar
- Assim, a noção de processo ligeiro, ou thread, é mais adaptada às necessidades de programação paralela
  - Partilham o espaço de endereçamento (memória, código)
  - São facilmente criadas e destruídas
  - Permitem sincronização sofisticada
- É o modelo mais adequado à programação em memória partilhada

Diferentes Aproximações

Duas formas de realizar programação baseada em threads:

- Utilizando directamente a API fornecida por uma plataforma
  - Posix Threads
  - Solaris Threads
  - Windows Threads
  - Etc.
- Utilizando um modelo de mais alto nível em que as directivas de paralelismo são expressas numa extensão da linguagem de programação
  - Cilk
  - OpenMP
- A nível da primeira categoria, a plataforma mais utilizada é a das Posix Threads (pthreads)

Conceito de Thread



- Um processo tradicional tem um fluxo de execução único
- No modelo multithreaded, em cada processo podem existir vários fluxos de execução ou *threads*
- As várias *threads* de um processo partilham o espaço de endereçamento, mas têm contextos de execução distintos

Exemplo de Utilização de Threads

- O seguinte fragmento de programa série:

```
for (row = 0; row < n; row++)
   for (column = 0; column < n; column++)
     matC[row][col] = mult(matA, row, matB, col)
```

- Pode ser transformado no seguinte programa paralelo:

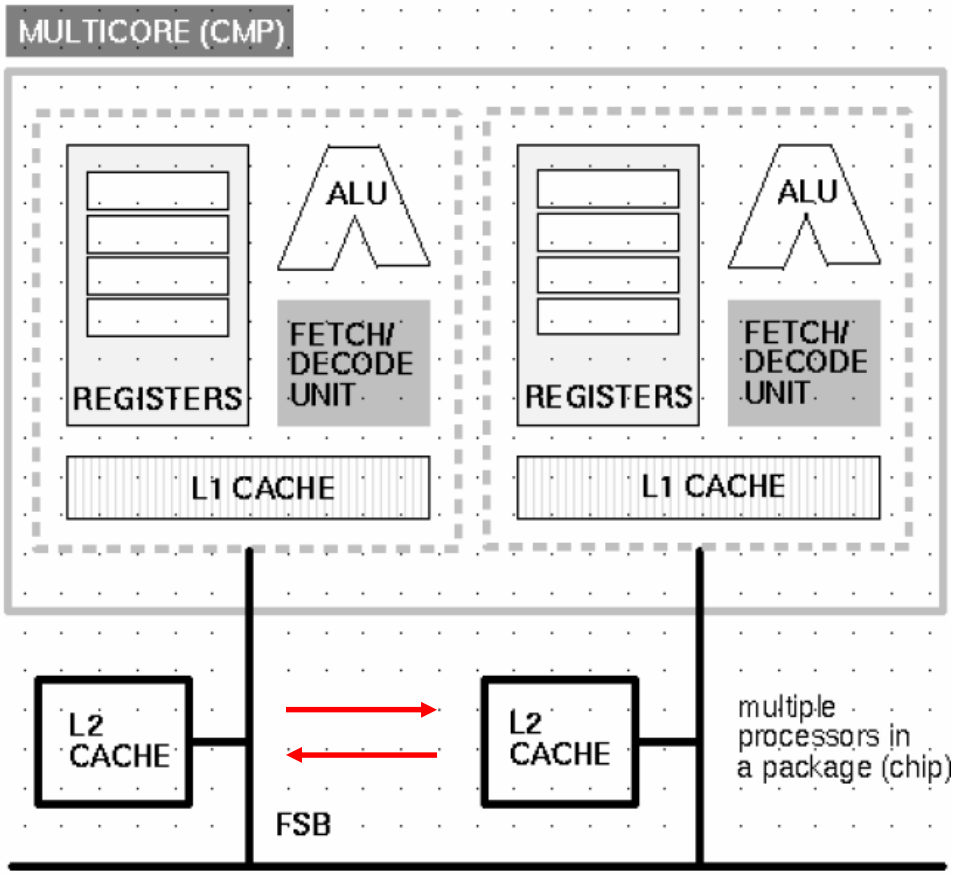
```
for (row = 0; row < n; row++)
   for (column = 0; column < n; column++)
     matC[row][col] = new_thread(mult(matA, row,
                                      matB, col))
```

- A criação de uma thread pode ser considerada como a forma de invocar uma função sem esperar que ela termine

Princípios de Funcionamento

- Toda a memória lógica acessível a um processo é igualmente partilhada por todas as threads desse processo
  - É necessário particular atenção às modificações de variáveis globais que podem ser modificadas por qualquer das threads sem garantia de ordem
- A única zona de dados privada por defeito de cada *thread* é a sua pilha de execução
  - As variáveis locais à função executada na thread são privadas
  - O programador pode também criar zonas de memória que são alocadas a cada thread
- Os acessos repetidos e concorrentes a memória comum levantam problemas
  - Coerência: as instruções de linguagens de alto nível não são indivisíveis
  - Desempenho: as linhas de *cache* podem ser alternadamente passadas de um *cache* de um processador para outro
  - O acesso simultâneo de várias threads à memória pode saturar a largura de banda da malha de interconecção (*memory wall*)

Problema do *False-Sharing*



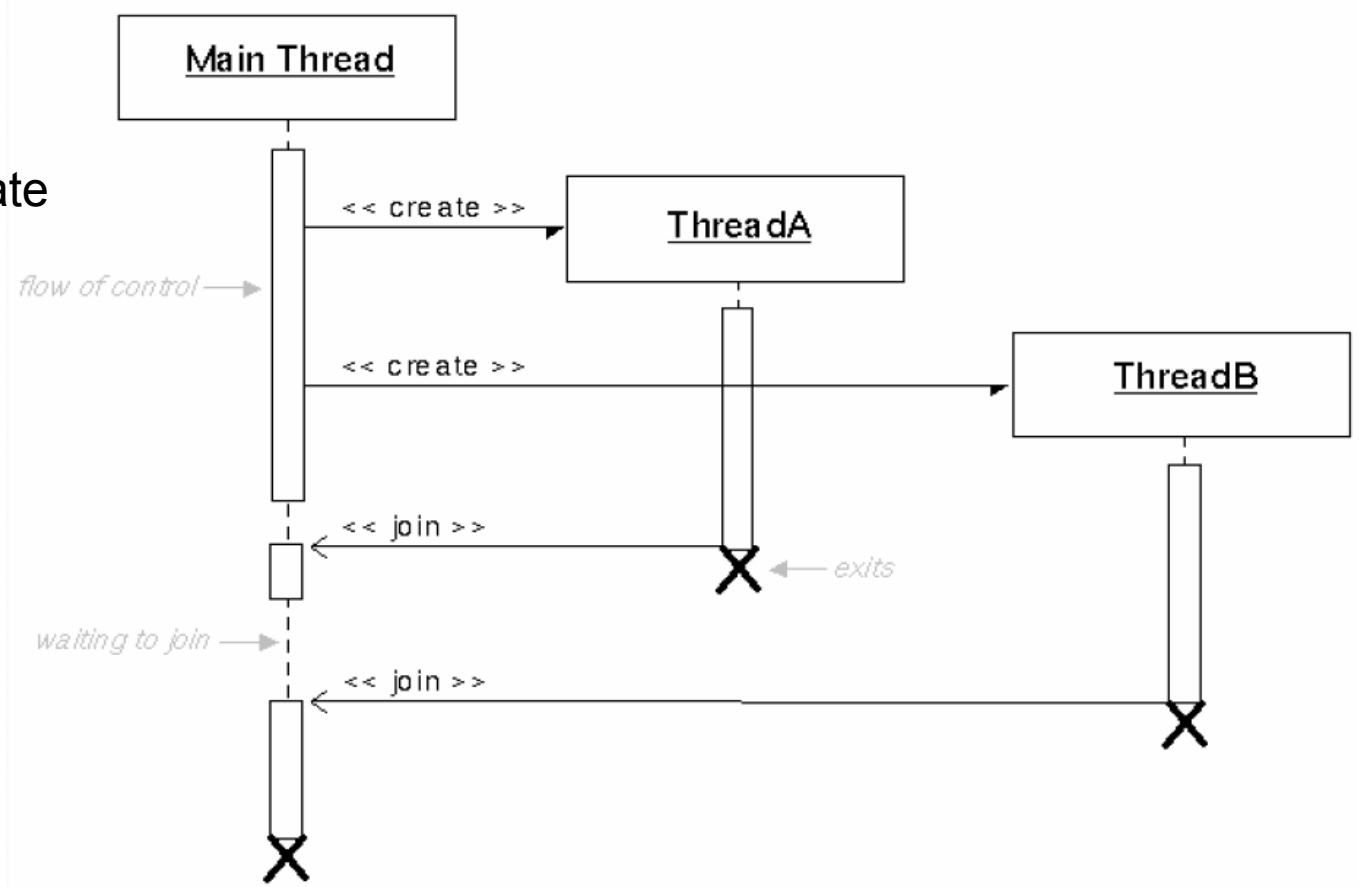
- O acesso em escrita a variáveis distintas mas pertencentes à mesma linha do cache, faz com que esta seja invalidada sucessivamente
- O valor actualizado está no cache do processador que modificou por último lugar

A API de threads Posix

- Conhecida vulgarmente como as Pthreads, a API Posix impôs-se como um standard para a programação em threads
  - Suportada pela maioria dos sistemas comerciais ou OpenSource
- Os conceitos que implementa são independentes das plataformas
  - Podem ser utilizados para a programação de outras APIs
- O desempenho e abrangência das funcionalidades podem variar segundo a implementação
  - No sistemas Linux, constituem a plataforma de threads nativa
- A API das pthreads é geralmente acessível em C e C++

Funções Básicas

- Duas funções básicas para definir fluxos concorrentes num programa
  - Criação
    - pthread_create
  - Terminação
    - pthread_join



Exemplo de criação e terminação

```
#include <pthread.h>
#include <stdlib.h>
#define MAX_THREADS 16
void *compute_pi (void *);
....
main() {
    ...
    pthread_t p_threads[MAX_THREADS];
    pthread_attr_t attr;
    pthread_attr_init (&attr);
    for (i=0; i< num_threads; i++) {
       hits[i] = i;
       pthread_create(&p_threads[i], &attr, compute_pi,
                                        (void *) &hits[i]);
    }
    for (i=0; i< num_threads; i++) {
       pthread_join(p_threads[i], NULL);
       total_hits += hits[i];
    }
    ...
}
```

Ver exemplo completo em:

http://netlab.ulusofona.pt/cp/praticas/pi.c

Outro Exemplo:

```c
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>

int file, mythread(int );

const char *msg[] = {
  "Thread #1 ",
  "Thread #2 "
};
#define MAXTHREADS 2

main(int argc, char *argv[])
{
   int ret, i;
   char buffer[64] = {"0"};
   pthread_t thread[MAXTHREADS];

   file = open (argv[1], O_RDWR);
   if (file < 0) {
     perror("file");
     exit(1);
   }
   for (i = 0; i < MAXTHREADS; i++) {
     ret = pthread_create(&thread[i], NULL, (void *)(&mythread), (void *)i);
     if (ret) {
       printf("ret %d\n", ret);
       perror("thread create");
       exit(1);
     }
   }
   printf("Main Thread\n");
   for (i = 0; i < MAXTHREADS; i++)
     pthread_join(thread[i], NULL);

   lseek(file, 0, SEEK_SET); // Reset the file I/O pointer !!

   ret = read(file, buffer, sizeof(buffer));
   if (ret < 0) {
     perror("read");
     exit(1);
   }
   printf("File Content:\n%s\n", buffer);
   printf("%s\n", buffer+strlen(buffer)+1);
}
```

Criação das threads

Espera

Código da thread

```c
mythread(int i)
{
  int ret;

  printf("I am %s\n", msg[i]);
  ret = write(file, msg[i], strlen(msg[i]));
  if (ret < 0)
     perror("write");
}
```

Sincronização em Exclusão Mútua

Necessidade:

- Quando múltiplas *threads* tentam aceder e modificar o mesmo endereço de memória, os resultados podem ser incoerentes se não forem tomadas precauções para garantir uma ordem determinística
- Considere-se o caso de 2 threads que acedem à mesma variável:

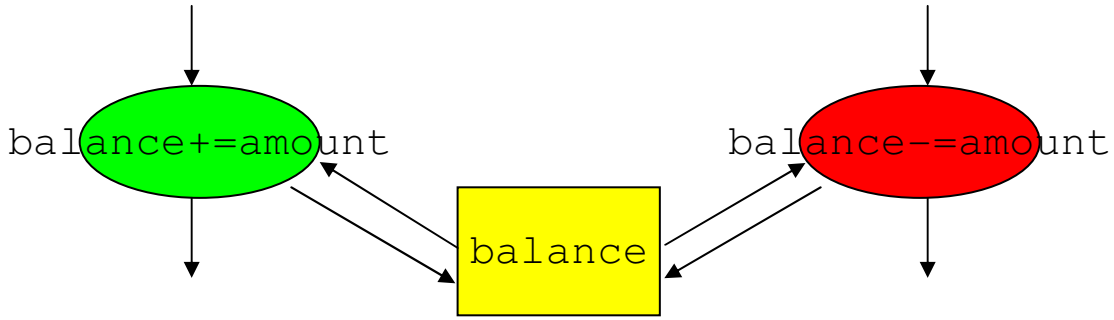
```
shared double balance;
```

$thread_1$:

```
. . .
balance = balance + amount;
```

$thread_2$:

```
. . .
balance = balance – amount;
```



- Se th_1 lê o valor da variável, é interrompida pelo scheduler do sistema e th_2 é executada, quando th_1 voltar a correr vai modificar o valor que th2 actualizou para um valor que não reflecte a modificação de th_2

Exclusão Mútua e Secções Críticas

- O código no exemplo anterior denomina-se uma secção crítica, e só pode ser executado por uma thread de cada vez
  - Tem de ser serializado
- As secções críticas constituem um problema recorrente de toda a programação paralela
  - Existem inúmeros metodologias para a sua resolução
- Nas pthreads, as protecção das secções críticas é realizada com recurso a *mutex locks*.
  - Objecto de sincronização que só pode ter dois estados
  - Locked e unlocked
- Num dado instante, **só uma thread** é capaz de mudar o estado do mutex, tornando-se assim proprietária exclusiva do mesmo
  - Assim, se o mutex estiver associado à secção crítica, é garantido que só uma thread a poderá executar

Protocolo de Sincronização

- Sempre que existe uma secção crítica no código executado por várias threads, estas devem seguir o protocolo de acesso e saída:

Teste e aquisição do mutex numa operação atómica

```
while (mutex <=> locked)
    thread sleep;
// mutex locked
Início secção crítica

Fim secção crítica
mutex_unlock;
// mutex unlocked
```

Esta operação é indivísivel !

Secção Crítica

resto do código

Primitivas de Sincronização

- As pthreads fornecem múltiplas primitivas de sincronização, mas a maioria dos casos pode ser resolvido com recurso a 3 funções:
  - **`pthread_mutex_init`**: inicialização de um mutex
  - **`pthread_mutex_lock`**: aquisição do mutex
  - **`pthread_mutex_unlock`**: libertação do mutex
- Assim o código do exemplo anterior pode ser escrito:

```
shared double balance;
pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL);
```

$thread_1$:

```
  . . .
pthread_mutex_lock(&lock);
balance = balance + amount;
pthread_mutex_unlock(&lock);
  . . .
```

$thread_2$:

```
. . .
pthread_mutex_lock(&lock);
balance = balance – amount;
pthread_mutex_unlock(&lock);
. . .
```

https://computing.llnl.gov/tutorials/pthreads/#Mutexes

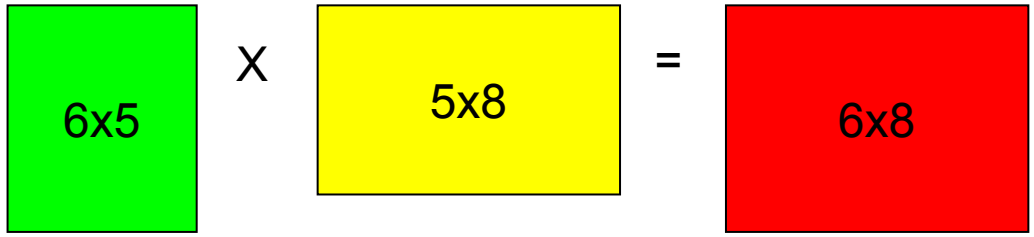
Semântica de Invocação

- Invocação de **`pthread_mutex_lock`** por uma thread:
  - Se o mutex já estiver bloquado por outra thread, a thread que invoca é suspensa até o mutex ser libertado
  - Se o mutex estiver livre, o mutex fica na posse da thread que invoca
- Uma thread que obtém um mutex deve SEMPRE libertá-lo quando sai da secção crítica
  - A não observância desta regra pode levar à situação de bloqueio fatal (*deadlock*) do conjunto de threads
- A invocação de **`pthread_mutex_lock`** por uma thread que já tenha obtido o mutex leva ao bloqueio fatal da mesma
- Existem outras formas de inicializar o mutex que permitem evitar estas situações
  - Mutex recursivo permite múltiplos locks pela mesma thread
  - Mutex com teste de erro, retorna erro se já estiver tomado pela thread que o tenta bloquear
  - **`pthread_mutex_try_lock`** que não suspende a thread no caso do mutex estar bloqueado

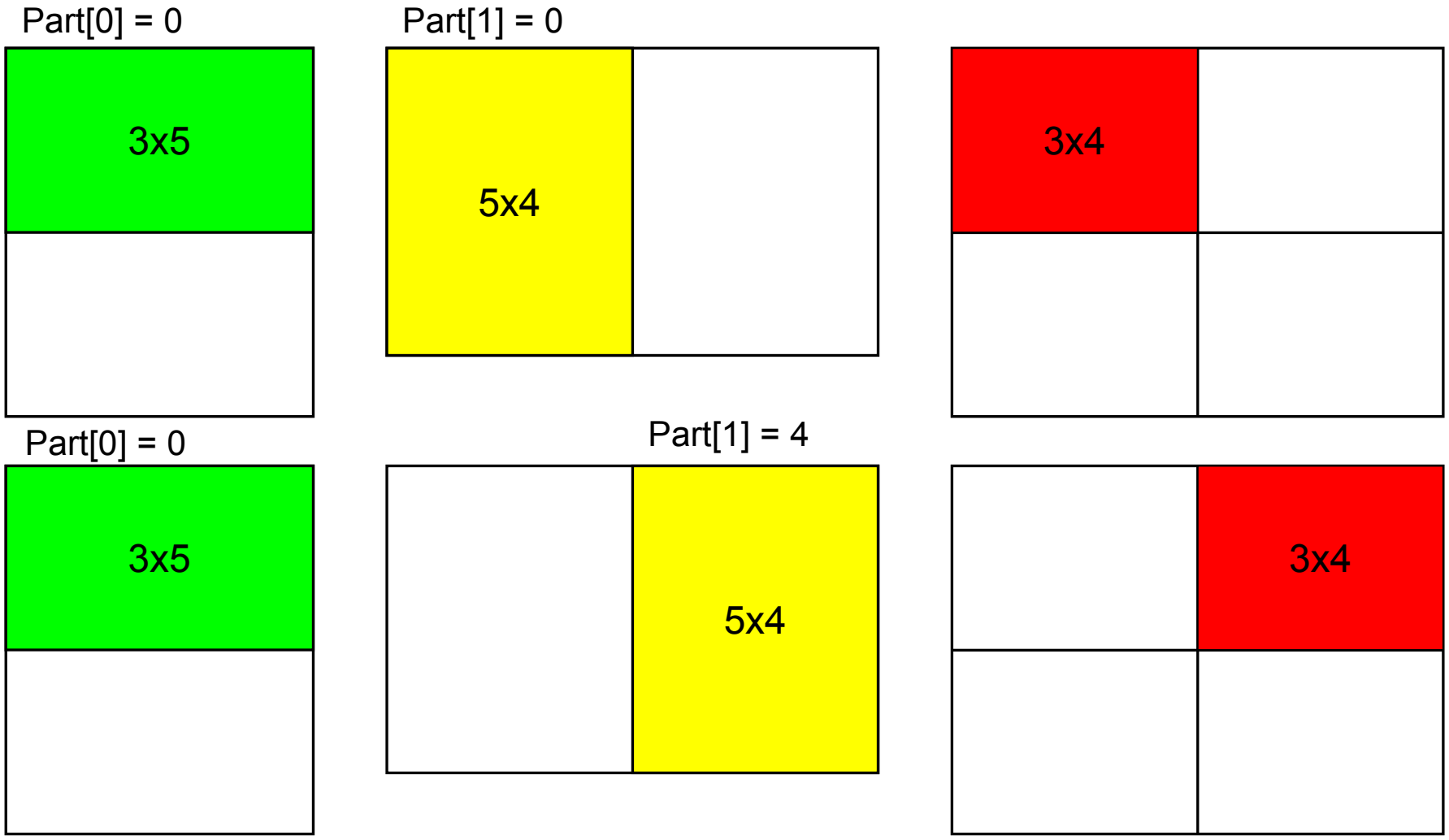
Exemplo de Paralelização

Objectivo:

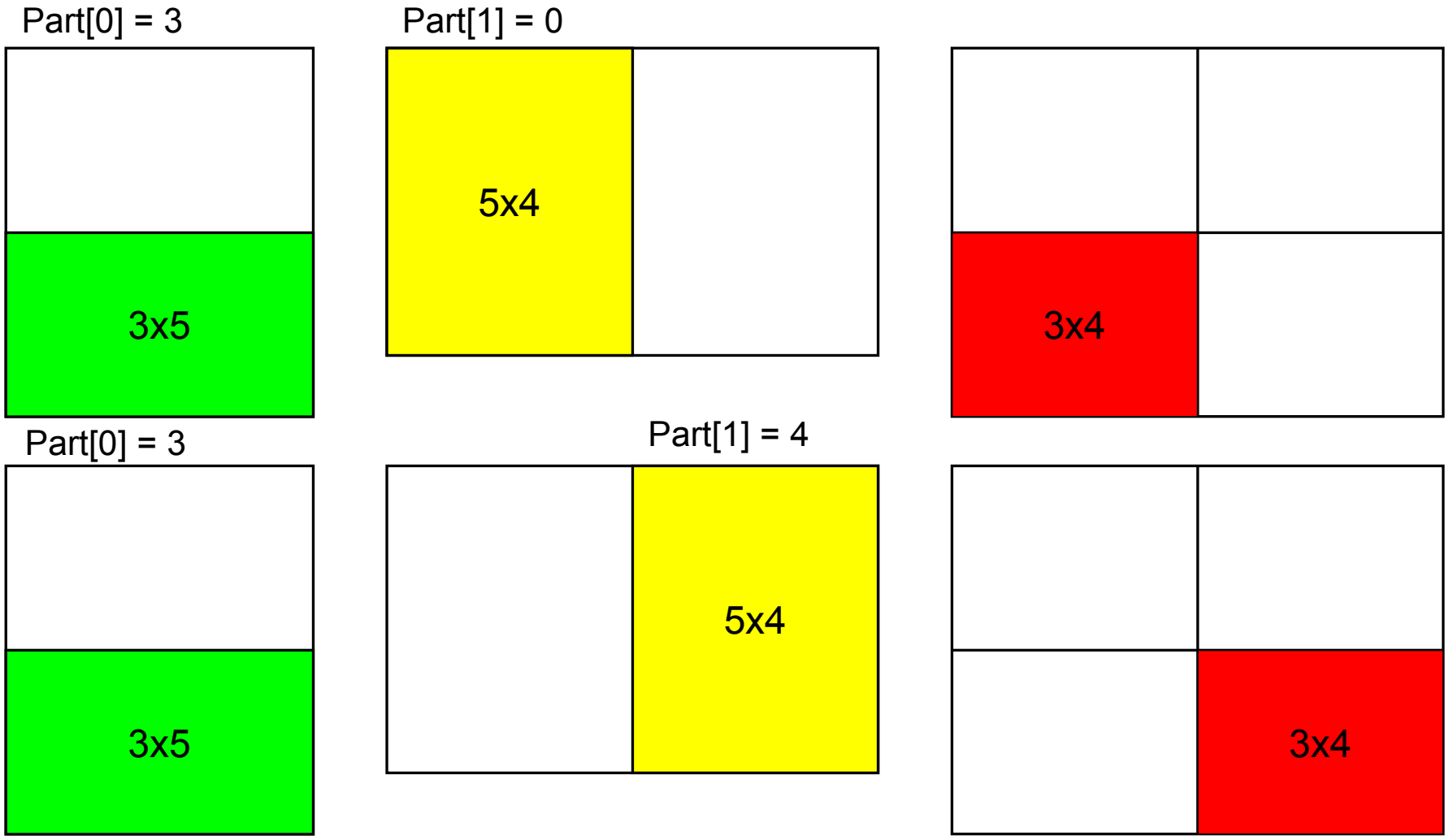
- Implementar a multiplicação de matrizes com recurso a paralelização, utilizando um método de decomposição adequado
  - 1º exemplo: decomposição dos dados de saída
- A matriz resultado é decomposta em 4 sub-matrizes com dimensões iguais a metade das dimensões finais
  - Restrição: as dimensões da matriz resultado devem ser pares
- As matrizes factores são decompostas de forma concordante com o algoritmo do produto
- Exemplo:



Partição das Matrizes (i)

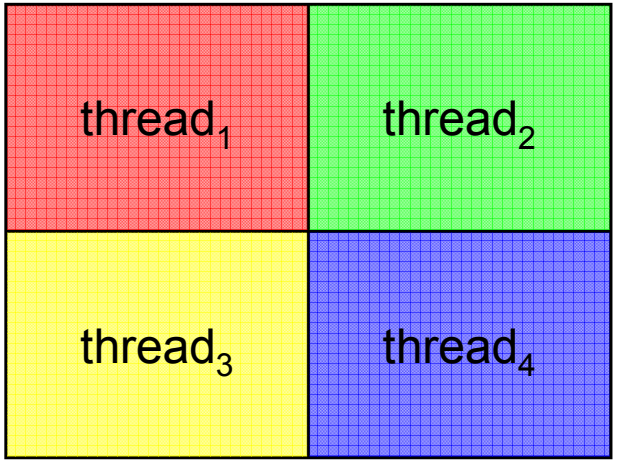


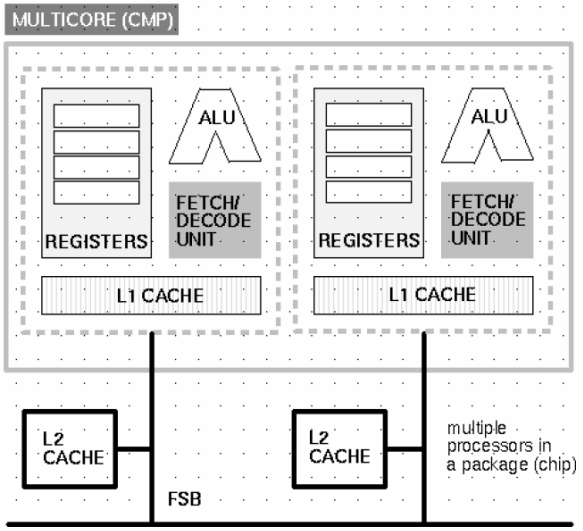
Partição das Matrizes (ii)



Mapeamento das Tarefas

- O cálculo da Matriz produto é realizado por 4 tarefas paralelas
- Cada uma calcula o produto das duas submatrizes que correspondem a uma partição da matriz resultado
- É feito um mapeamento de cada tarefa para uma thread que vai calcular uma partição da matriz resultado
- As threads lêem as partições respectivas das matrizes factor e escrevem numa partição distinta da matriz produto
  - Não existem secções críticas
- A partilha de dados é feita a partir da memória principal
  - As dimensões das matrizes podem não caber nos caches dos vários processadores





Análise do Código

Programação Baseada em Directivas

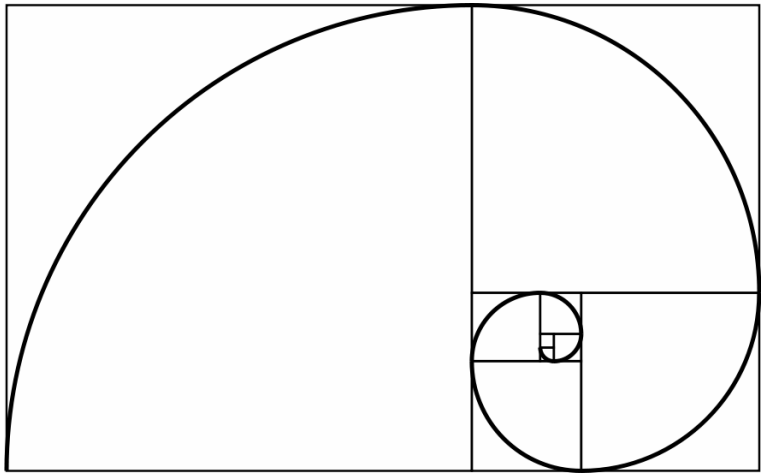
- O modelo de programação das *Pthreads* é eficaz e detalhado, mas o nível de abstracção é pouco elevado
- O programador tem de gerir os todos os aspectos ligados implementação podendo perder de vista os aspectos do paralelismo
- Nesse sentido, a **Programação Baseada em Directivas** permite abstrair toda a complexidade da implementação das threads
  - O programador concentra-se na expressão do paralelismo
  - O paralelismo é declarado com base em *directivas* que são transformadas em fluxos de execução paralela por um pré-processador
- Dois exemplos entre muitos:
  - **Cilk**: plataforma desenvolvida no Supercomputing Technologies Group do MIT (Prof. Charles Leiserson)
  - **OpenMP**: standard desenvolvido por um conjunto de entidades ligadas à computação paralela e com diversas implementações

Introdução ao Cilk

- O principal objectivo do Cilk é estender a linguagem C com algumas (poucas) directivas ***simples***
  - Permite a programação de algoritmos paralelos por não especialistas de programação paralela
  - O runtime Cilk encarrega-se de tirar o máximo partido da plataforma de execução
- Um programa Cilk pode ser criado a partir da versão série do algoritmo, com a inserção de directivas de paralelismo
  - A estrutura do programa série é integralmente respeitada e o programa continua a funcionar mesmo se forem retiradas as directivas
  - A versão série do programa é designada por "*C elision*"

Exemplo: cálculo da série de Fibonacci

- Introduzidos pelo italiano Leonardo de Pisa - *Fibonacci* (séc XIII)
- Conhecidos desde a antiguidade na Índia, representam diversas formas da natureza e do pensamento filosófico
- O cálculo da série de Fibonacci é um exemplo clássico de recursividade



$$F(n) = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F(n-1) + F(n-2) & \text{if } n > 1. \end{cases}$$

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}	F_{18}	F_{19}	F_{20}
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584	4181	6765

Série de Fibonacci em Cilk

Source: Multithreaded Programming in Cilk, Charles Leiserson

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

C elision

Cilk code

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

- O Cilk é uma extensão exacta do C
- A versão série de um programa em Cilk é um programa em C correcto
- O Cilk não introduz nenhum tipo novo de dados nem de operador, apenas as directivas de paralelismo

Directivas Cilk Básicas

Source: Multithreaded Programming in Cilk, Charles Leiserson

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

Identifica uma função como uma ***rotina Cilk***, capaz de ser executada em paralelo

Indica que a rotina Cilk ***invocada*** pode ser executada em paralelo com a que a ***invoca***

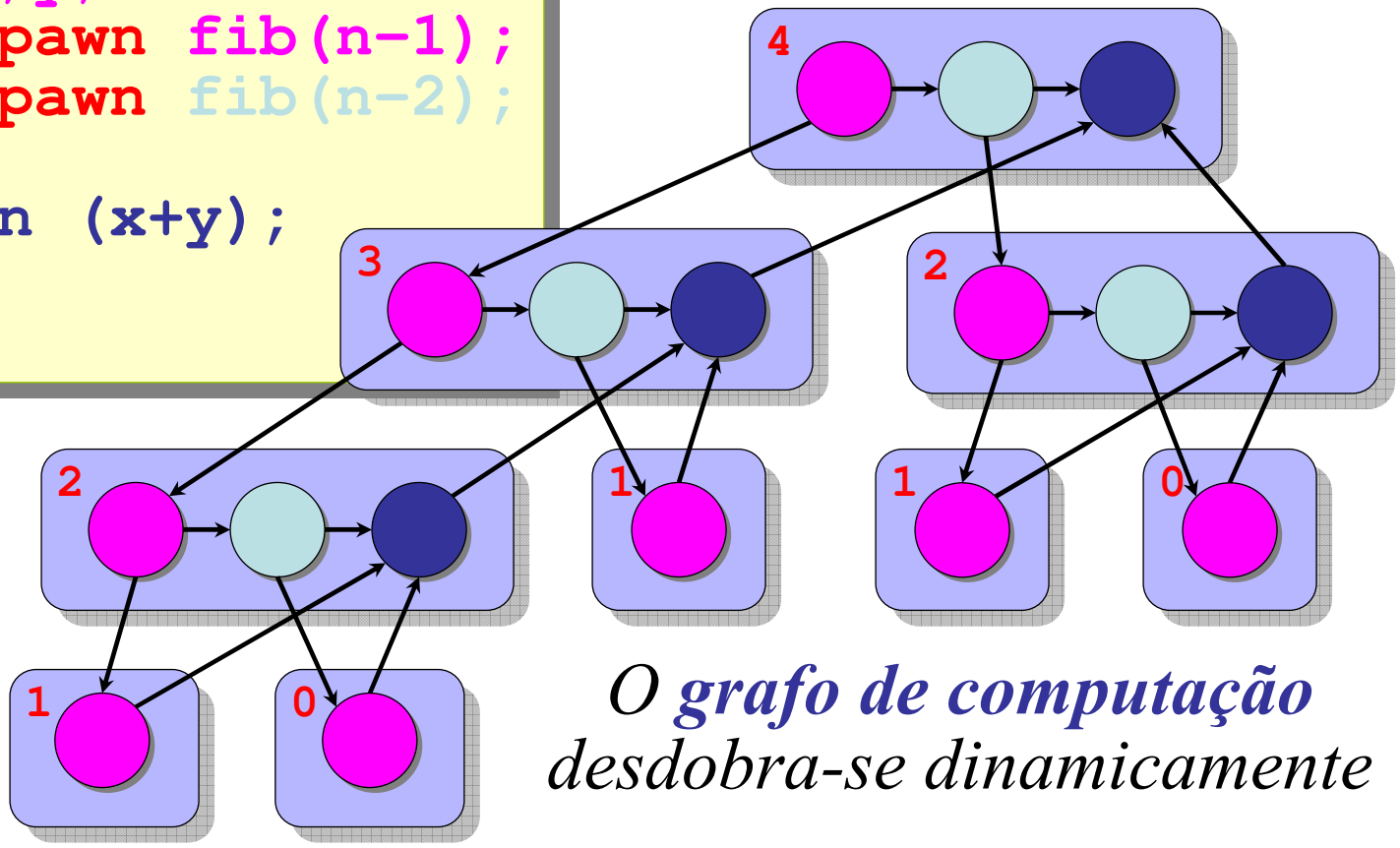
O programa não passa deste ponto sem que ***todas*** as rotinas Cilk invocadas tenham terminado

Criação de Dinâmica de Threads

Source: Multithreaded Programming in Cilk, Charles Leiserson

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

Exemplo: `fib(4)`



Independente do Processador

O grafo de computação desdobra-se dinamicamente

Paralelização da Adição de Vectores

Source: Multithreaded Programming in Cilk, Charles Leiserson

C

```
void vadd (real *A, real *B, int n){
  int i; for (i=0; i<n; i++) A[i]+=B[i];
}
```

C

```
void vadd (real *A, real *B, int n){
  if (n<=BASE) {
    int i; for (i=0; i<n; i++) A[i]+=B[i];
  } else {
    vadd (A, B, n/2);
    vadd (A+n/2, B+n/2, n-n/2);
  }
}
```

Estratégia de Paralelização:

1. Converter ciclos em recursividade.

Paralelização da Adição de Vectores

Source: Multithreaded Programming in Cilk, Charles Leiserson

C

```
void vadd (real *A, real *B, int n){
  int i; for (i=0; i<n; i++) A[i]+=B[i];
}
```

Cilk

```
cilk void vadd (real *A, real *B, int n){
  if (n<=BASE) {
    int i; for (i=0; i<n; i++) A[i]+=B[i];
  } else {
    spawn vadd (A, B, n/2);
    spawn vadd (A+n/2, B+n/2, n-n/2);
  } sync;
}
```

Estratégia de Paralelização:

1. Converter ciclos em recursividade.
2. Inserir directivas Cilk

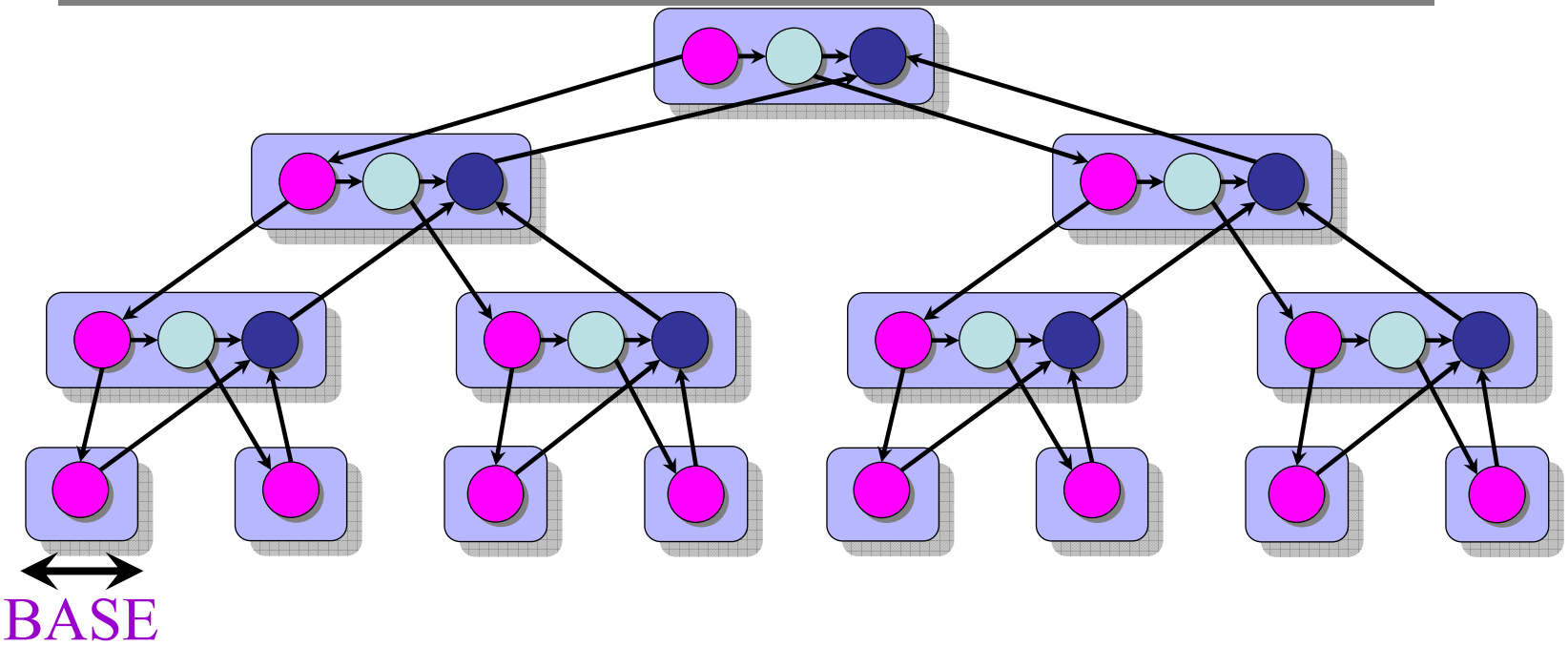
Efeito adicional:
D&C é geralmente bom para os caches!

Grau de Recursividade

Source: Multithreaded Programming in Cilk, Charles Leiserson

```
cilk void vadd (real *A, real *B, int n){
  if (n <= BASE) {
    int i; for (i=0; i<n; i++) A[i]+= B[i];
  } else {
    spawn vadd (A, B, n/2);
    spawn vadd (A+n/2, B+n/2, n-n/2)
    sync;
  }
}
```

A escolha do valor de BASE determina o grau de recursividade



Multiplicação de Matrizes Quadradas

$$\begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \mathbf{x} \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}$$

C A B

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Para simplificar supomos que $n = 2^k$.

Utilização de Recursividade

Metodologia *"Divide and conquer":*

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \mathbf{x} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$= \begin{bmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{bmatrix} + \begin{bmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{bmatrix}$$

8 multiplicações de (n/2) x (n/2)
1 adição de 2 matrizes n x n.

Multiplicação

```
cilk void Mult(*C, *A, *B, n) {
  float *T = Cilk_alloca(n*n*sizeof(float));
  < partition matrices >
  if (n = BASE)
      C11 = A11.B11
  else {
     spawn Mult(C11,A11,B11,n/2);
     spawn Mult(C12,A11,B12,n/2);
     spawn Mult(C22,A21,B12,n/2);
     spawn Mult(C21,A21,B11,n/2);
     spawn Mult(T11,A12,B21,n/2);
     spawn Mult(T12,A12,B22,n/2);
     spawn Mult(T22,A22,B22,n/2);
     spawn Mult(T21,A22,B21,n/2);
     sync;
     spawn Add(C,T,n);
     sync;
  }
}
```

Alocação de memória temporária na pilha

$$C = A \cdot B$$

Adição

```
cilk void Mult(*C, *A, *B, n) {
  float *T = Cilk_alloca(n*n*sizeof(float));
  <partition matrices>
  if (n = BASE)
      C11 = A11.B11
  else {
     spawn Mult(C11,A11,B11,n/2);
     spawn Mult(C12,A11,B12,n/2);
     spawn Mult(C22,A21,B
     spawn Mult(C21,A21,B
     spawn Mult(T11,A12,B
     spawn Mult(T12,A12,B
     spawn Mult(T22,A22,B
     spawn Mult(T21,A22,B
     sync;
     spawn Add(C,T,n);
     sync;
  }
}
```

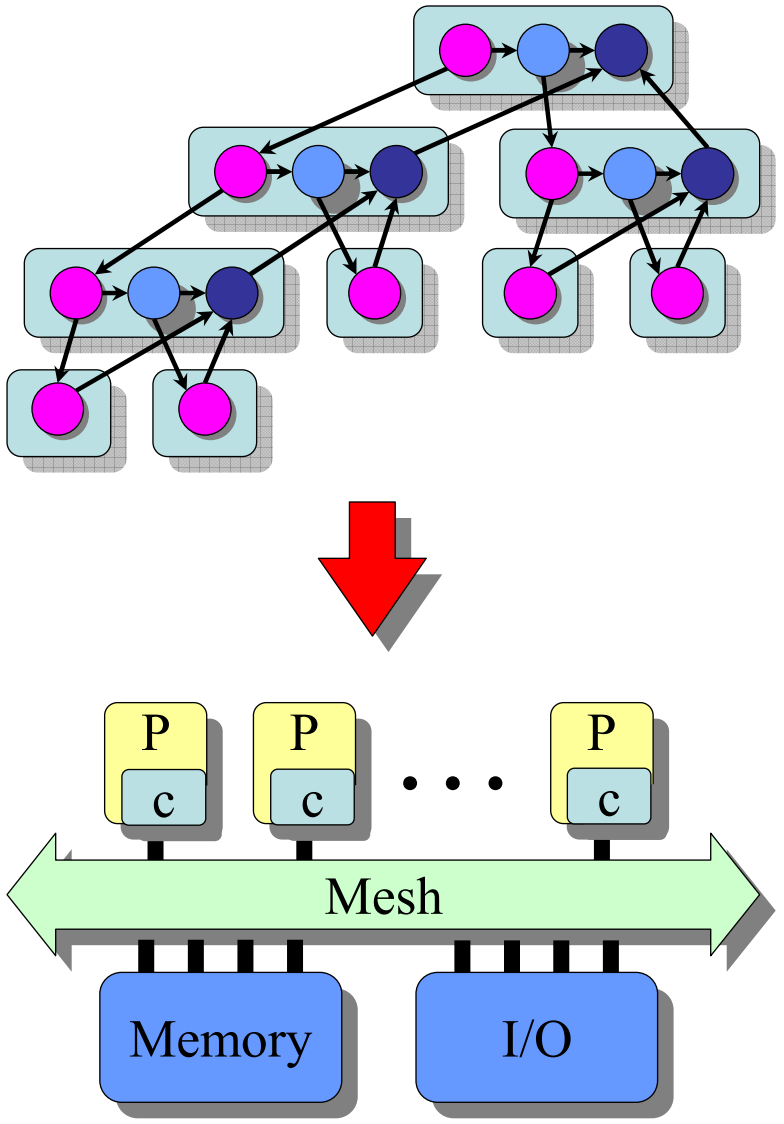
$C = A \,.\, B$

```
cilk void Add(*C, *T, n) {
<partition matrices>
 if (n == BASE)
  C11 = A11 + T11
 else {
  spawn Add(C11,T11,n/2);
  spawn Add(C12,T12,n/2);
  spawn Add(C21,T21,n/2);
  spawn Add(C22,T22,n/2);
  sync;
 }
}
```

$C = C + T$

Mapeamento

- Cilk permite ao programador exprimir o *potencial* paralelismo de uma aplicação
- O *scheduler* do Cilk mapeia dinamicamente as threads nos processadores da plataforma
- Se bem que o *scheduler* seja considerado eficaz, o desempenho está sempre dependente do sistema operativo e da correcção do programa

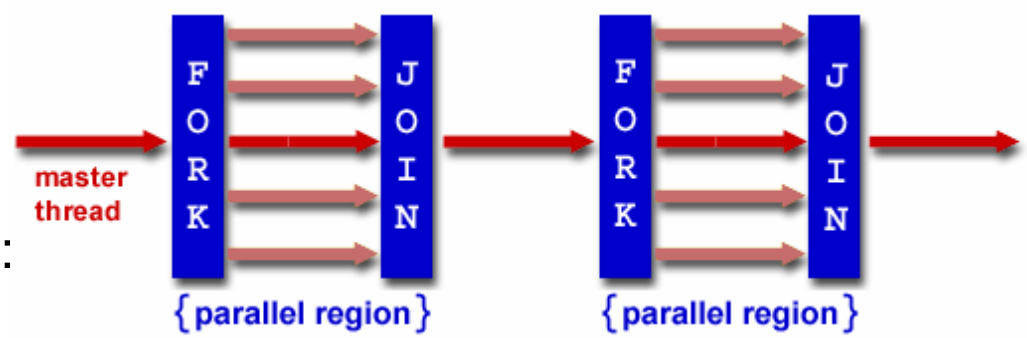


OpenMP

- Se bem que o Cilk seja uma plataforma didáctica e intuitiva, não foi adoptada de forma generalizada pelos programadores de aplicações paralelas
  - Demasiado focada nos algoritmos recursivos de tipo "*Divide and Conquer*"
- Na realidade, a plataforma mais utilizada em sistemas de memória partilhada é o OpenMP
  - Open Multiprogramming Platform (http://openmp.org)
- Plataforma suportada por inúmeros fabricantes de distribuidores de software
  - Hewlett-Packard Company
  - Intel Corporation
  - International Business Machines (IBM)
  - Silicon Graphics, Inc.
  - Sun Microsystems, Inc.
  - National Center for Supercomputing Applications, etc…
- A especificação da API é um standard que começou a ser definido a partir de 1997, contando com inúmeras versões e optimizações
  - Suporte para Fortran e C/C++

Modelo de Programação OpenMP

- O OpenMP é composto por vários componentes
  - Directivas para o compilador
  - Biblioteca de rotinas de *runtime*
  - Variáveis de ambiente
- As directivas fornecem suporte para:
  - Concorrência
  - Sincronização
  - Gestão de dados
- Evitando recurso explícito a:
  - Mutexes
  - Variáveis de condição
  - Definição de âmbito dos dados (*scope*)
  - Inicialização



Directivas OpenMP

- As directivas em C e C++ são baseadas na directiva `#pragma` do compilador que permite explicitar o paralelismo e controlá-lo
- Uma directiva consiste num tipo seguido de cláusulas

```
#pragma omp directive [clause list]
```

- Um programa corre sequencialmente até encontrar uma directiva de tipo `parallel` que instancia um grupo de novas threads

```
#pragma omp parallel [clause list]
/* structured block */
```

- A thread que encontra a directiva `parallel` transforma-se na coordenadora (master) do grupo de threads criado, e é-lhe atribuída a identidade 0 dentro desse grupo

Cláusulas OpenMP

- As cláusulas associadas à directiva `parallel` podem especificar condições, grau de paralelismo e propriedades dos dados
- Paralelização Condicional
  - Criação de paralelismo depende do resultado da expressão

    `if (scalar expression)`
- Grau de paralelismo
  - Especificação do número de threads a criar

    `num_threads(integer expression)`
- Propriedades dos dados
  - Variáveis locais a cada thread

    `private (variable list)`
  - Variáveis locais a cada thread inicializadas previamente

    `firstprivate (variable list)`
  - Variáveis partilhadas pelas threads criadas nesse ponto

    `shared (variable list)`

Exemplo de Programa OpenMP

- Exemplo de programa com directivas e a sua eventual tradução para pthreads realizada pelo compilador OpenMP

```
int a, b;
main() {
  // serial segment
  #pragma omp parallel num_threads (8) private (a) shared (b)
  {
    // parallel segment
  }
  // rest of serial segment
}
```

Sample OpenMP program

```
int a, b;
main() {
  // serial segment

  for (i = 0; i < 8; i++)
      pthread_create (......., internal_thread_fn_name, ...);

  for (i = 0; i < 8; i++)
      pthread_join (.......);

  // rest of serial segment

}

void *internal_thread_fn_name (void *packaged_argument) [
    int a;
  // parallel segment
}
```

Code inserted by the OpenMP compiler

Corresponding Pthreads translation

Exemplo de Directiva `parallel`

```
#pragma omp parallel if (is_parallel == 1)
 num_threads(8) \
 private (a) shared (b) firstprivate(c) {
 /* structured block */
 }
```

- Se o valor de `is_parallel` for verdadeiro, são inicializadas 8 threads
- Cada thread recebe cópias da das variáveis `a` e `c` e todas partilham a variável `b`
- A variável `c` é inicializada com o valor corrente antes da criação das threads
- O estado por defeito das variáveis pode ser determinado através da cláusulas

  `default(shared) ou default(none)`

Cláusula de Redução

- A cláusula especifica como são combinadas as múltiplas cópias locais de uma variável numa única cópia quando as threads terminam

```
reduction (operator: variable list)
```

- Indica como são combinadas as múltiplas cópias locais de uma variável numa única cópia quando as threads terminam
- As variáveis constando da lista são implicitamente especificadas como privadas
- O operador pode ser:

```
+, *, –, &, |, ^, &&, e ||
```

- Exemplo:

```
#pragma omp parallel reduction(+: sum) num_threads(8) {
    /* compute local sums here */
}
/*sum here contains sum of all local instances of sums */
```

Exemplo do Cálculo de PI

```
/*****************************************************
An OpenMP version of a threaded program to compute PI.
*****************************************************/
#pragma omp parallel default(private) shared (npoints) \
   reduction(+: sum) num_threads(8)
{
   num_threads = omp_get_num_threads();
   sample_points_per_thread = npoints / num_threads;
   sum = 0;
   for (i = 0; i < sample_points_per_thread; i++) {
      rand_x =(double)(rand_r(&seed))/(double) RAND_MAX;
      rand_y =(double)(rand_r(&seed))/(double) RAND_MAX;
      if (((rand_x - 0.5) * (rand_x - 0.5) +
         (rand_y - 0.5) * (rand_y - 0.5)) < 0.25)
         sum ++;
   }
}
```

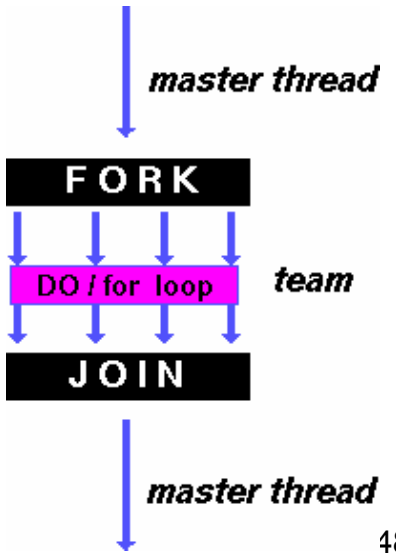
Especificação Explícita de Concorrência

- A directiva `parallel` pode ser utilizada em conjunto com outras para explicitar divisão específica de ciclos por threads
- A directiva `for` associada a um ciclo permite distribuir automaticamente o espaço de iteração pelas *threads* declaradas na directiva `parallel` anterior

```
#pragma omp for [clause list]
   /* for loop */
```

- Exemplo: cada uma das threads irá realizar ¼ das iterações `maxi`

```
#pragma omp parallel num_threads(4) {
  #pragma omp for
  for(i=1; i<=maxi; i++){
     a[i] = b[i];
  }
}
```



Exemplo de utilização da Cláusula `for`

```
/*****************************************************
An OpenMP version of a threaded program to compute PI.
*****************************************************/
#pragma omp parallel default(private) shared (npoints) \
   reduction(+: sum) num_threads(8)
{
   sum = 0;
   #pragma omp for
   for (i = 0; i < npoints; i++) {
      rand_x =(double)(rand_r(&seed))/(double) RAND_MAX;
      rand_y =(double)(rand_r(&seed))/(double) RAND_MAX;
      if (((rand_x - 0.5) * (rand_x - 0.5) +
         (rand_y - 0.5) * (rand_y - 0.5)) < 0.25)
         sum ++;
   }
}
```

Mais funcionalidades…

- As funcionalidades apresentadas são suficientes para implementar grande parte dos problemas abordados.
- Para saber mais, existem inúmeros *tutorials*
  - *https://computing.llnl.gov/tutorials/openMP*: para referência, com muitos exemplos
  - *http://ci-tutor.ncsa.uiuc.edu/index.php*: conjunto de cursos da NCSA muito detalhados e didácticos sobre várias plataformas
    - OpenMP
    - MPI
- Exercício:
  - Transpor para OpenMP o exemplo de cálculo de PI disponível em
    - *http://netlab.ulusofona.pt/cp/praticas/pi.c*
  - Para compilar o programa
    - cc -o prog prog.c -fopenmp

Programação por Mensagens

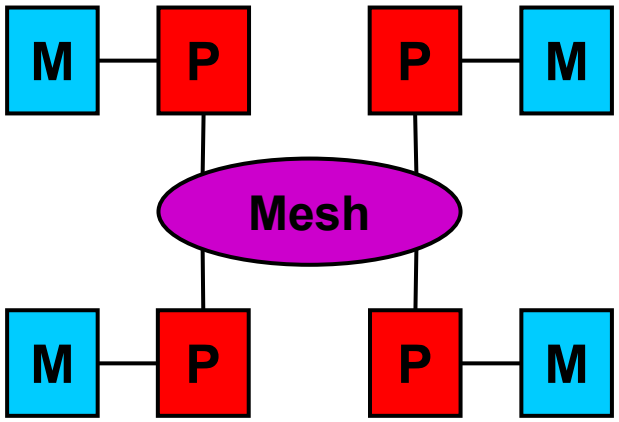
- Princípios de Programação por Mensagens (PpM)
- As operações básicas: send e receive
- Variantes das operações básicas
- MPI: Message Passing Interface
- Exemplos e aplicações

Princípios da Programação por Mensagens

- A programação paralela baseada em mensagens é utilizada em plataformas em que cada nó de processamento dispõe da sua memória própria e não existe memória partilhada
  - Existência de uma partição física do espaço memória
- Cada conjunto de dados de processamento pertence a uma única partição de memória que é gerida por um processo
  - Os dados têm portanto de ser explicitamente particionados e colocados no local adequado
- A coordenação do processamento e distribuição dos dados é feita por interacção entre os processos através da troca de ***mensagens***
- As interacções requerem uma colaboração explícita entre os processos intervenientes
  - Processo que envia e o processo que recebe
- Estes constrangimentos fazem com que os custos de comunicação sejam claramente perceptíveis pelo programador

Modelo de Execução

- A programação por mensagens utiliza um modelo de programação em que os processos de execução de executam de forma assíncrona ou com um sincronismo fraco
- As tarefas são executadas de forma independente com pontos de sincronização impostos pelas dependências existentes nos algoritmos
- A execução obedece ao modelo ***Single Program Multiple Data*** (SPMD)



Primitivas Básicas

- A PpM é baseada em duas operações fundamentais

  `send e receive`

- A definição dessas funções básicas pode ser a seguinte

```
send(void *sendbuf, int nelems, int dest)
receive(void *recvbuf, int nelems, int source)
```

- Considerando o excerto de código:

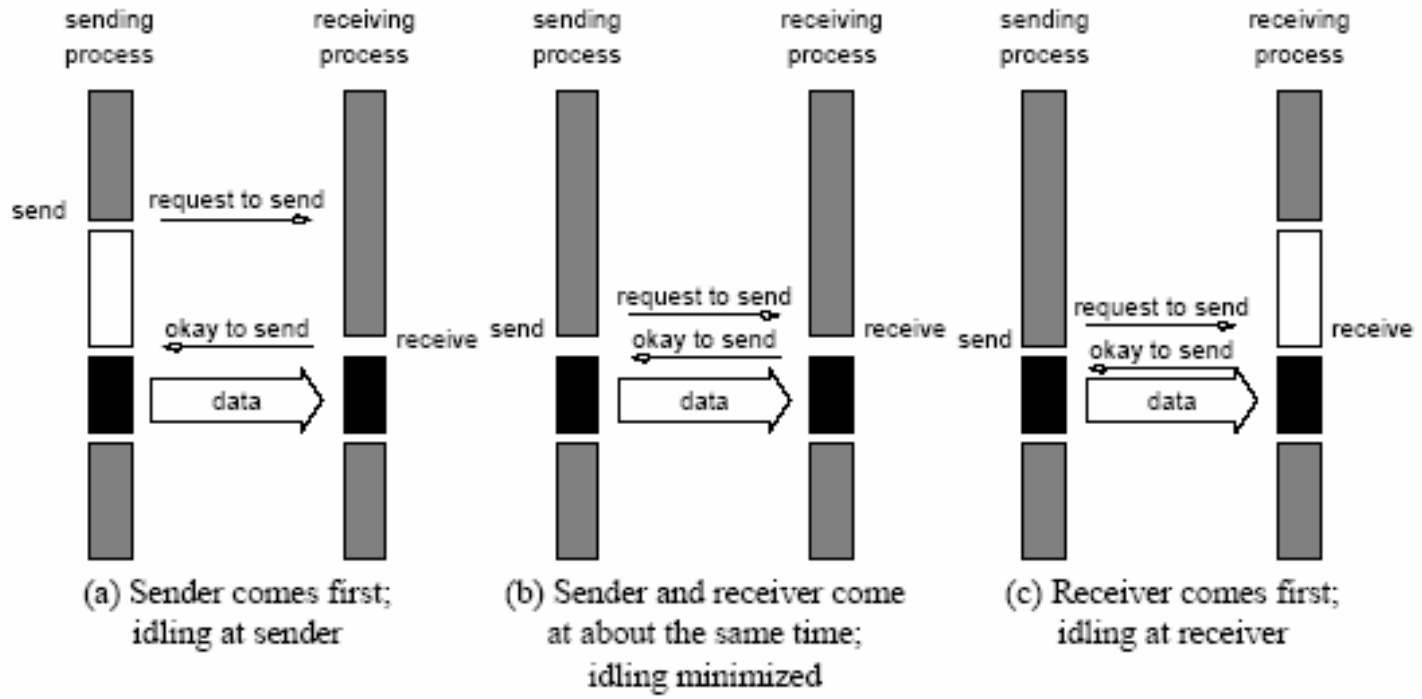
```
P0                      P1
a = 100;                receive(&a, 1, 0)
send(&a, 1, 1);         printf("%d\n", a);
a = 0;
```

- A semântica da operação `send` requer que o valor recebido pelo processo P1 seja 100 e não 0
- Isto condiciona a implementação do protocolo `send/receive` entre os dois processos

Operações Bloqueantes

- Um método simples para obter a semântica desejada é que `send` só retornar depois de ser garantida a recepção
- No caso do `send` bloqueante sem armazenamento temporário, a operação não retorna sem que o `receive` correspondente tenha sido completado pelo processo receptor
- Esta aproximação tem dois inconvenientes
  - O processo que emite não faz nada sem que o send retorne
  - No caso de não haver um processo para receber a mensagem, pode surgir um *deadlock*

Sincronização Emissor/Receptor

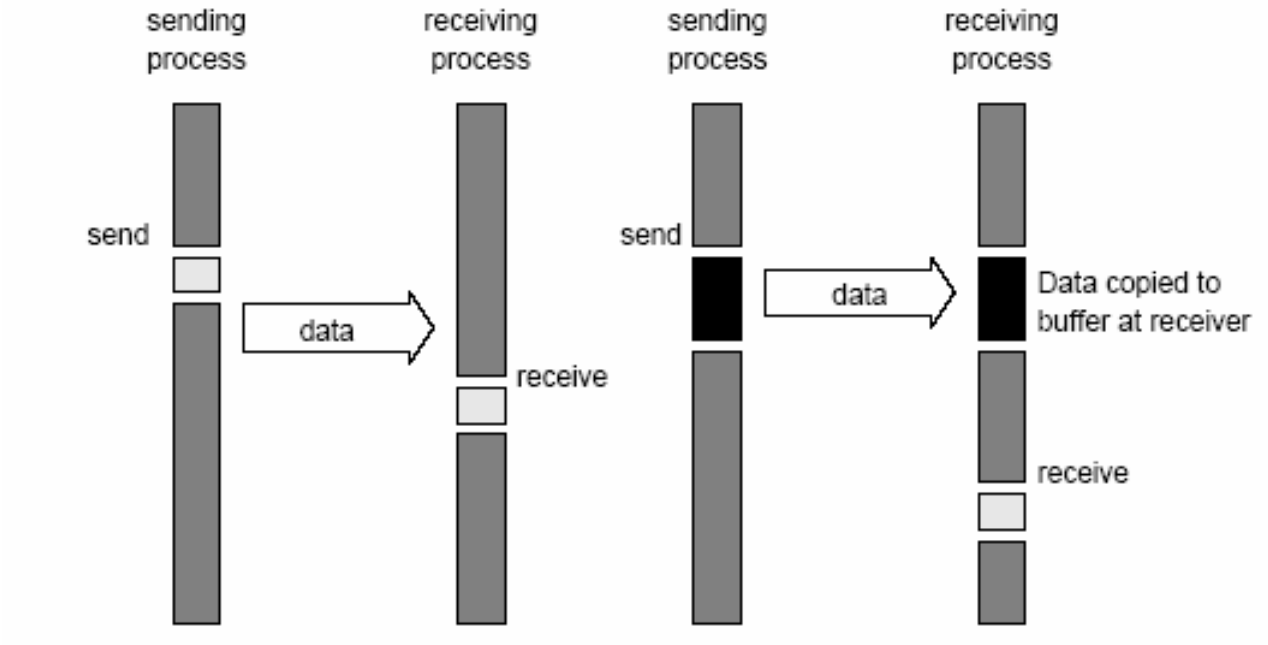


- A sincronização pode levar a tempos de espera consideráveis se os processos não atingem o ponto de comunicação em simultâneo

Utilização de Armazenamento Temporário

- Uma segunda variante do `send` pode copiar os dados para um *buffer* temporário e retornar assim que a cópia esteja realizada
  - A memória cujo conteúdo foi copiado pode ser modificada logo de seguida mantendo a semântica da operação
- Os dados também devem ser armazenados temporariamente na recepção
- O armazenamento temporário (*buffering*) evita tempos de espera inactivos à custa de um maior número de cópias e de maior utilização de memória

Independência entre Emissor e Receptor



- A sincronização em caso de utilização de buffering pode ser mais eficaz se o hardware permitir operações de transferência sem intervenção do CPU
  - DMA - *Direct Memory Access*

Problema do *buffer* Limitado

- No caso de se utilizar *buffering*, o espaço temporário disponível pode ter impactos importantes no desempenho

- Exemplo:

```
P0                                  P1
  for (i = 0; i < 10000; i++){      for (i = 0; i < 10000; i++){
     produce_data(&a);                 receive(&a, 10, 0);
     send(&a, 10, 1);                  consume_data(&a);
  }                                 }
```

- O que acontece se o receptor é mais lento que o emissor?

Problema de *Deadlock*

- Podem acontecer situações de deadlock com buffering, uma vez que estas são bloqueantes

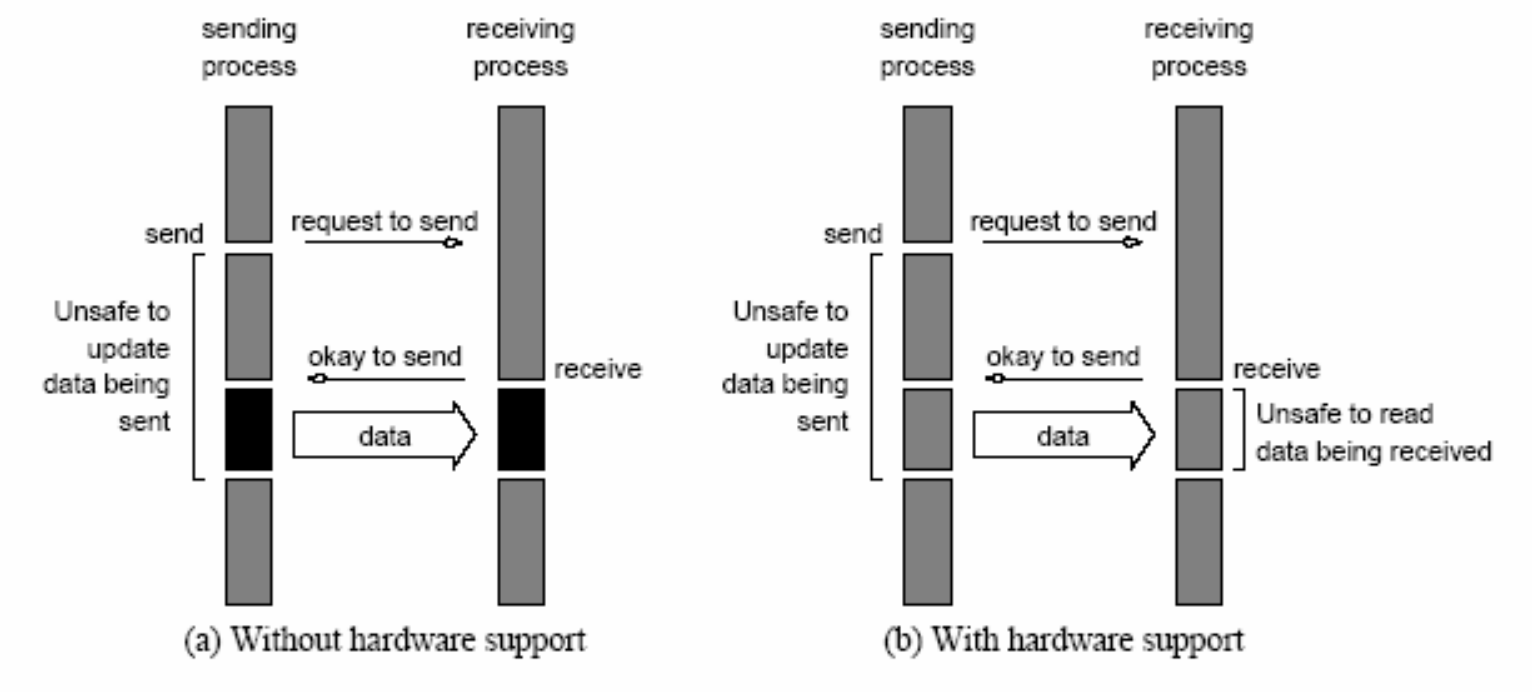
- Como se sai desta sequência?

```
P0                          P1
receive(&a, 1, 1);          receive(&a, 1, 0);
send(&b, 1, 1);             send(&b, 1, 0);
```

Operações Não Bloqueantes

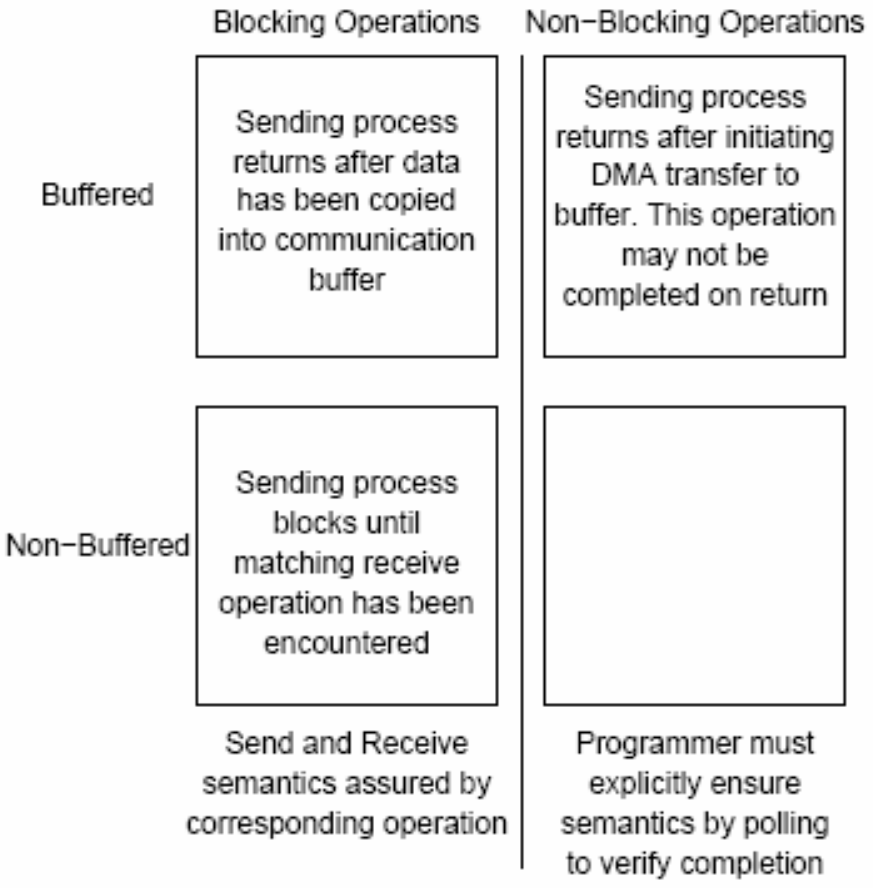
- Uma outra forma de implementar as operações send e receive é iniciar o processo de transmissão de dados mas não esperar pela conclusão e retornar imediatamente
  - Não é semanticamente seguro aceder aos dados depois da invocação das operações
- As operações não bloqueantes têm de ser acompanhadas por operações que permitam testar o estado da transferência
  - Compete ao programador garantir a correcção da operação através do teste da sua conclusão
- Quando usadas correctamente, estas primitivas permitem realizar operações de transferência em paralelo com computação
  - *Overlapping Operations*
- Usualmente as bibliotecas de mensagens fornecem versões bloqueantes e não bloqueantes das primitivas de transferência

Sincronização em Modo não Bloqueante



- A sincronização em modo não bloqueante implica a validação da operação pela parte do programador
- O hardware de DMA permite desacoplar ainda mais as transferências da computação

Resumo das Variantes Possíveis



O MPI: Message Passing Interface

- O MPI é um exemplo de plataforma que implementa as várias variantes do envio de mensagens
  - E não só….!
- O MPI é um standard que define uma API para mensagens programação paralela em C e Fortran, de forma independente da plataforma
  - O standard define a sintaxe e a semântica de uma biblioteca de rotinas
- Existem implementações gratuitas (ou não) de MPI para todas as versões de computadores paralelos com arquitectura distribuída
  - A comunicações podem ser optimizadas em função da malha de interligação dos processadores
- O MPI contém as definições de cerca de 300 rotinas
  - Na realidade, é possível escrever programas paralelos completamente funcionais utilizando apenas 6 rotinas !

MPI: Rotinas Básicas

O conjunto mínimo de rotinas MPI

`MPI_Init`	Inicializa o runtime MPI.
`MPI_Finalize`	Finaliza o runtime MPI.
`MPI_Comm_size`	Determina o número de processos.
`MPI_Comm_rank`	Determines a ordem do processo corrente.
`MPI_Send`	Envia uma mensagem.
`MPI_Recv`	Recebe uma mensagem.

Tutoriais MPI

Para uma introdução ao MPI, seguir um dos seguintes tutoriais:

- LLNC - Lawrence Livermore National Laboratory
  - Informação directa e resumida
  - https://computing.llnl.gov/tutorials/mpi
- NCSA - National Center for Supercomputing Applications
  - Apresentação completa, com muitos exemplos e testes (necessita registo)
  - http://ci-tutor.ncsa.uiuc.edu/browse.php
    - Introduction to MPI
- Objectivo
  - Escrever um pequeno programa básico em MPI e testá-lo na instalação do laboratório

Instalação do MPI

- A instalação pode ser feita em qualquer conjunto de (pelo menos) quatro máquinas de uma mesma rede
  - Exemplo: uma bancada do laboratório
- A versão aconselhada é o MPICH2 desenvolvida pelo Argonne National Laboratory
  - http://www.mcs.anl.gov/research/projects/mpich2/index.php
- A instalação do MPI pode ser feita em cerca de 30 a 60 minutos com a ajuda de alguns documentos essenciais:
  - Guia de Instalação
    - http://www.mcs.anl.gov/research/projects/mpich2/documentation/files/mpich2-doc-install.pdf
  - Configuração do SSH para MPI
    - http://source.ggy.bris.ac.uk/wiki/Configure_ssh_for_MPI
  - Adicionalmente, o guia de utilizador também pode ser útil
    - http://www.mcs.anl.gov/research/projects/mpich2/documentation/files/mpich2-doc-user.pdf
- Se possível convém configurar um DNS para que os nomes dos nós sejam conhecidos uns dos outros