# Arquitetura de Computadores

*Suporte às Aulas Laboratoriais*

Paulo Menezes



Para os alunos do
Mestrado em Eng. Electrotécnica e Computadores
Dept. Eng. Electrotécnica e Computadores
Universidade de Coimbra

# Conteúdo

# Prefácio

Conhecer a arquitectura dos computadores actuais é de extrema importância para o Engenheiro Electrotécnico, dado que estes dispositivos estão presentes em qualquer ramo desta engenharia, quer sob a forma do computador propriamente dito, como embebido em robôs, sistemas de controlo de energia, dispositivos de automação, ou telecomunicações.

Ao analisar os sistemas embebidos vemos que o suporte computacional realiza actualmente operações em tempo real que no passado eram efectuados por circuitos analógicos ou digitais, uma vez que permite o realização, teste e correcção dos algoritmos de forma mais simples, o que leva a ciclo de desenvolvimento mais curto.

Assim o presente documento surge como uma colecção de fichas de trabalho a realizar dentro e fora das aulas laboratoriais da disciplina de Arquitectura de Computadores leccionada no Departamento de Engenharia Electrotécnica e Computadores. No entanto não se considera um documento acabado, uma vez que esta tecnologia está em constante evolução e por isso serão num futuro próximo adicionadas novas fichas relativas a temas mais avançados. De facto isso não aconteceu antes pois este é o primeiro ano onde todos os alunos que frequentam estas aulas terão passado obrigatoriamente pelas disciplinas como Laboratório de Sistemas Digitais e Sistemas de Microprocessadores.

Este documento servirá assim de apoio às aulas práticas, onde serão seleccionadas apenas determinadas fichas de trabalho, uma vez que estas são em número superior às aulas programadas para o semestre.

# MIPS - Funções e chamadas ao sistema operativo

1

Os exercícios que se seguem devem ser testados com o MARS (simulador do MIPS). Para isso leia a sua documentação previamente. O MARS é uma aplicação escrita em Java e por isso pode ser executada em Windows, Linux ou Mac OS. (http://courses.missouristate.edu/KenVollmar/MARS/) Este simulador simula o MIPS mas também alguns serviços do sistema operativo que estão disponíveis através de chamadas ao sistema (syscall).

## 1.1 Introdução

Vamos começar por rever alguns conceitos de microprocessadores.

### Pilha

A pilha (stack) é uma zona de memória usada para dados temporários. O programador deve-se assegurar que ao terminar uma função, a pilha deve ficar exactamente no mesmo estado em que se encontrava na entrada da mesma função. A linguagem C (e C++) faz uso da pilha para passagem de parâmetros às funções. Assim uma função "int a()"que chama uma função "int b(int p1, int p2, char p3)", deve colocar na pilha os parâmetros p1, p2 e p3, pela ordem inversa dos mesmos. O valor de retorno faz-se normalmente através de um registo.

No caso do processador MIPS, porque há um grande número de registos gerais e se pretende minimizar o mais possível os acessos a memória, os primeiros 4 parâmetros são passados directamente pelos registos $a0 a $a3

| Register name | Number | Usage |
|---|---|---|
| $zero | 0 | constant 0 |
| $at | 1 | reserved for assembler |
| $v0 | 2 | expression evaluation and results of a function |
| $v1 | 3 | expression evaluation and results of a function |
| $a0 | 4 | argument 1 |
| $a1 | 5 | argument 2 |
| $a2 | 6 | argument 3 |
| $a3 | 7 | argument 4 |
| $t0 | 8 | temporary (not preserved across call) |
| $t1 | 9 | temporary (not preserved across call) |
| $t2 | 10 | temporary (not preserved across call) |
| $t3 | 11 | temporary (not preserved across call) |
| $t4 | 12 | temporary (not preserved across call) |
| $t5 | 13 | temporary (not preserved across call) |
| $t6 | 14 | temporary (not preserved across call) |
| $t7 | 15 | temporary (not preserved across call) |
| $s0 | 16 | saved temporary (preserved across call) |
| $s1 | 17 | saved temporary (preserved across call) |
| $s2 | 18 | saved temporary (preserved across call) |
| $s3 | 19 | saved temporary (preserved across call) |
| $s4 | 20 | saved temporary (preserved across call) |
| $s5 | 21 | saved temporary (preserved across call) |
| $s6 | 22 | saved temporary (preserved across call) |
| $s7 | 23 | saved temporary (preserved across call) |
| $t8 | 24 | temporary (not preserved across call) |
| $t9 | 25 | temporary (not preserved across call) |
| $k0 | 26 | reserved for OS kernel |
| $k1 | 27 | reserved for OS kernel |
| $gp | 28 | pointer to global area |
| $sp | 29 | stack pointer |
| $fp | 30 | frame pointer |
| $ra | 31 | return address (used by function call) |

Figura 1.1: Uso de cada um dos registos do MIPS na linguagem C

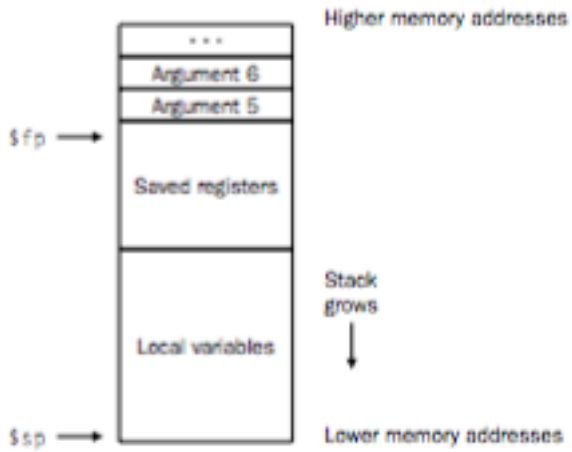



Figura 1.2: Utilização da pilha por uma função, ou quadro (frame) de uma função

($4 a 7$), os restantes se existirem serão colocados na pilha usando a convenção acima descrita. Para o retorno de valores de uma função usa-se o registo $v0 ($2) e em caso de necessidade (números de dupla precisão por ex) o registo $v1 ($3).

Além disso, quando uma função chama outra função, esta necessita de guardar o seu próprio endereço de retorno ($ra) na pilha de forma a poder recuperá-lo quando pretender regressar ao código que a chamou.

Porque os programas podem ter partes escritas em assembly e outras em C, é importante conhecer a convenção usada pelos compiladores de C para o processador MIPS no que diz respeito à utilização dos registos. Esta convenção é descrita pela tabela da figura 8.1.

A figura 8.2 ilustra a utilização da pilha aquando da chamada de uma função. Note-se que o registo auxiliar da pilha ($bp) fica a apontar para onde estava o stack pointer aquando da entrada na função. Ou seja, acima do $bp estão os parâmetros passados a esta função pela função chamante; abaixo está o contexto (variáveis e registos salvos) da função chamada e de outras que esta possa vir a chamar. Note-se que uma função pode chamar-se a si própria, no entanto cada "instância"tem o seu próprio quadro (frame) de execução. Quando uma função está prestes a retornar, deve repor o registo $sp=$bp antes de sair.

## 1.2 Chamadas ao sistema operativo

Por várias razões, entre as quais a segurança do próprio sistema informático, os serviços prestados pelo sistema operativo aos programas só está acessíveis através dum mecanismo próprio, normalmente designado como "chamadas ao sistema". A sua importância requer que os processadores tenham suporte para estas, frequentemente através de instruções específicas.

No processador MIPS existe a instrução "syscall"pare esse efeito. No Mars, uma vez que este simula não só o processador mas também um conjunto de serviços do sistema operativo, as chamadas ao sistema são efectuadas da seguinte forma:

1. coloca-se no registo $v0 o número do serviço pretendido (ver tabela no HELP do Mars)

2. os argumentos, caso existam, são passados através dos registos $a0, $a1, $a2, ou $f12

3. executa-se a instrução "syscall"

4. se houver valores de retorno, estes devem ser recuperados a partir dos registos especificados (tabela do HELP do Mars)

## 1.3 Antes da aula

Faça uma revisão dos conceitos de programação para a arquitetura MIPS e leia a documentação do simulador Mars. Prepare toda a informação necessária para resolver os exercícios que se seguem no decorrer da aula. Comece por analisar o código que se segue e que corresponde ao ficheiro "Fibonacci.asm"disponibilizado na página do simulador Mars.

```
# Compute first twelve Fibonacci numbers and put in array, then print
        .data
fibs:   .word   0 : 12          # "array" of 12 words to contain fib values
size:   .word   12              # size of "array"
        .text
        la    $t0, fibs         # load address of array
        la    $t5, size         # load address of size variable
        lw    $t5, 0($t5)       # load array size
        li    $t2, 1            # 1 is first and second Fib. number
        sw    $t2, 0($t0)       # F[0] = 1
        sw    $t2, 4($t0)       # F[1] = F[0] = 1
        addi  $t1, $t5, -2      # Counter for loop, will execute (size-2) times
loop:   lw    $t3, 0($t0)       # Get value from array F[n]
        lw    $t4, 4($t0)       # Get value from array F[n+1]
        add   $t2, $t3, $t4     # $t2 = F[n] + F[n+1]
        sw    $t2, 8($t0)       # Store F[n+2] = F[n] + F[n+1] in array
        addi  $t0, $t0, 4       # increment address of Fib. number source
```

```
        addi $t1, $t1, -1     # decrement loop counter
        bgtz $t1, loop        # repeat if not finished yet.
        la   $a0, fibs        # first argument for print (array)
        add  $a1, $zero, $t5  # second argument for print (size)
        jal  print            # call print routine.
        li   $v0, 10          # system call for exit
        syscall               # we are out of here.

######## routine to print the numbers on one line.

        .data
space:.asciiz  " "            # space to insert between numbers
head: .asciiz  "The Fibonacci numbers are:\n"
        .text
print:add  $t0, $zero, $a0    # starting address of array
      add  $t1, $zero, $a1    # initialize loop counter to array size
      la   $a0, head          # load address of print heading
      li   $v0, 4             # specify Print String service
      syscall                 # print heading
out:  lw   $a0, 0($t0)        # load fibonacci number for syscall
      li   $v0, 1             # specify Print Integer service
      syscall                 # print fibonacci number
      la   $a0, space         # load address of spacer for syscall
      li   $v0, 4             # specify Print String service
      syscall                 # output string
      addi $t0, $t0, 4        # increment address
      addi $t1, $t1, -1       # decrement loop counter
      bgtz $t1, out           # repeat if not finished
      jr   $ra                # return
```

## 1.4 Na aula

**Exercício 1.1**
Escreva um programa que peça ao utilizador para introduzir um número inteiro e retorne a sequência dos números da série de Fibonacci entre 0 e o inteiro introduzido. Parta do código fornecido no sitio do MARS como base.

**Exercício 1.2**
Escreva e teste um programa que vá pedindo ao utilizador para introduzir números inteiros e vá calculando a sua soma. O programa deve terminar e apresentar o resultado quando o utilizador introduzir o valor 0.

**Exercício 1.3**
Escreva um programa que peça ao utilizador para introduzir um número inteiro e apresente o valor do seu factorial. Nota o calculo do factorial deve ser efectuado de forma iterativa.

**Exercício 1.4**

Escreva um programa que calcule e imprima todos os números primos inferiores a um valor dado pelo utilizador.

**Exercício 1.5**
Reescreva o exercício 1 de forma recursiva. Sugestão: Neste caso deve criar uma função fibo(m) que seguindo a convenção do C recebe dois parâmetros: 1 inteiro e um outro que controla se deve imprimir ou não o resultado. Esta função deve chamar-se a si própria com os parâmetros adequados.

**Exercício 1.6**
Reescreva o exercício 3 de forma recursiva.

**Exercício 1.7**
Escreva um programa que ilustre a resolução do problema das torres de Hanoi com um número de discos dado pelo utilizador.

**Exercício 1.8**
Escreva um programa que dado um número inteiro positivo introduzido pelo utilizador, imprima o nome dos dígitos do mesmo no ecrã. Ex: 352 - três cinco dois.

# MIPS - Manipulação de matrizes

# 2

**Exercício 2.1**
Escreva um programa em linguagem assembly do MIPS para correr no simulador MARS que efectue o seguinte:

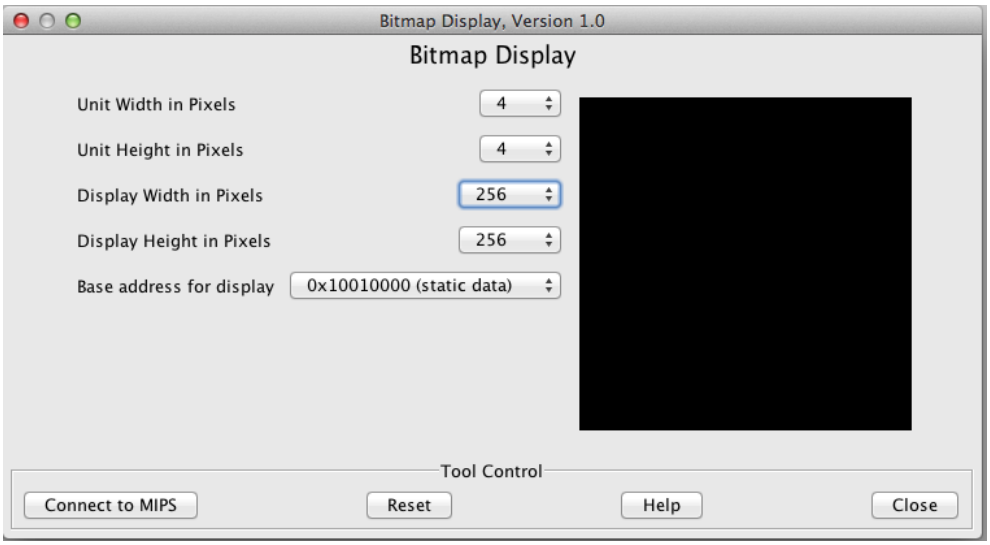
1. Peça ao utilizador o nome de um ficheiro
2. Leia desse ficheiro os valores dos elementos de duas matrizes de inteiros de dimensão 10 por 10. O ficheiro tem o seguinte formato:
   a) 1 numero por linha composto por quatro caracteres ascii. Ex. 0091
   b) Os 100 valores da primeira matriz seguem-se linha após linha.
   c) Uma linha em branco para separar as duas matrizes.
   d) Os 100 valores da segunda matriz com o mesmo formato que a primeira
3. O resultado da multiplicação deve ser escrito num ficheiro de nome resultado e com o mesmo formato que o de entrada (mas apenas com 1 matriz)

# MIPS - Periféricos mapeados em memória

3

Ao contrário de outros processadores, o MIPS não tem espaços de endereçamentos separados para entradas e saídas e por consequência também não tem instruções próprias para tal. Assim sendo, quaisquer periféricos são mapeados no espaço de endereçamento normal. Para isto cada dispositivo periférico terá lógica de descodificação das linhas de endereçamento e, quando o(s) endereço(s) do mesmo surgirem no barramento de endereços, as operações de LOAD ou STORE corresponderão respectivamente a leituras ou escritas no periférico.

Vamos usar como primeiro exemplo um terminal gráfico mapeado na memória, organizado como mapa de bits, ou seja a cada pixel corresponde um conjunto de bits e modificando o seu valor alteramos a cor do pixel correspondente. No menu Tools do simulador MARS encontra-se disponível uma ferramenta (bitmap display) que simula um terminal gráfico deste tipo. Depois de selecionada esta opção, surgirá a janela seguinte.




Nesta surge a area a preto que representa o terminal gráfico, cujas dimensões são definidas nas opções "Display WIdth in Pixels"e "Display Height in Pixels", estas dimensões são em relação aos pixeis do ecrã do PC que estamos a usar. Nas opções "Unit Width in PixelsE e "Unit Height in Pixels"vamos selecionar o tamanho de cada pixel da máquina virtual na máquina real. Ou seja, usando as opções da figura acima temos que cada pixel da máquina virtual ocupa 4 por 4 pixeis da máquina real e uma vez que o terminal ocupa 256 por 256 pixeis da máquina real, a resolução da máquina virtual será de 64 por 64 pixeis.

A cor de cada pixel corresponde à combinação de valores de vermelho, verde e azul armazenados numa palavra de 32 bits, armazenados sequencialmente a partir do endereço base que no exemplo da figura corresponde a $0x10010000$. Ou seja primeiro os 64 pixeis da primeira linha, seguido pelos da segunda linha e assim consecutivamente.

Os 32 bits de cada estão organizados da seguinte forma em termos de cores (RGB): vermelho (bits 16-23), verde (bits 8-15), azul (bits 0-7) e os bits 24 a 31 não são utilizados. Note-se que, por erxemplo, o valor 0 para vermelho significa total ausência de cor vermelha e valores crescentes até um máximo de 255 correspondem a intensidades crescentes dessa cor, o mesmo se passando para o verde e azul.

Assim, as cores são formadas por combinações de intensidades destas componentes e se usarmos valores iguais nas 3 componentes iremos obter tonalidades de cinzendo (do preto até ao branco). Por outro lado tomando como exemplo a cor amarela, para um pixel tomar esta cor deve-se escrever na zona de memória correspondente o valor $0x0000ffff$.

Para os exercícios que se seguem iremos usar a configuração da figura

acima para termos um display de $64 \times 64$ pixeis.

## 3.1 Preparação

**Exercício 3.1**
Escreva um programa em Assembly MIPS que preencha todos os pixeis com a cor branca (vermelho=255, verde=255, azul=255).

**Exercício 3.2**
Escreva um programa em Assembly MIPS que preencha todos os pixeis com um gradiente de cor em que a cor verde aumenta de 4 em 4 ao longo das colunas, e a cor vermelha da mesma forma ao longo das linhas.

**Exercício 3.3**
Repita o exercício anterior preenchendo com o gradiente apenas o interior de um quadrado centrado nas coordenadas (32,32) e com 20 pixeis de lado. O exterior do quadrado deve ser preenchido com a cor azul.

## 3.2 A Entregar

Os exercícios que se seguem devem ser entregues num ficheiro zip organizados em pastas separadas, contendo também um relatório no formato PDF ou TXT. O ficheiro com o relatório deve conter o nome e numero dos dois (2) elementos do grupo, e a descrição do que foi e não foi implementado. O relatório deverá ter no máximo 1 página.

**Exercício 3.4**
Pretende-se criar um programa que desenhe gráficos de valores. Para isso siga os seguintes passos:

1. Abra no Mars a ferramenta "Bitmap Display"e selecione a altura e largura em pixeis igual a 256x256, selecionando de seguida 4x4 a dimensão de cada pixel. Isto configura o ecrã como contendo 64 por 64 pixeis.

2. crie uma função **int asciitoint(char * cval)** que recebe o endereço de um array de 3 caracteres e devolve o inteiro neles representado. Por questões de simplicidade o array será composto sempre por um carater + ou - na primeira posição seguido de dois digitos decimais. Exemplos: +00, +02, -23 .

3. Crie uma função **setpixel(int linha, int coluna, unsigned red, unsigned green, unsigned blue)** em linguagem assembly.

4. crie uma função **clear(unsigned red, unsigned green, unsigned blue)** que preencha todos os pixeis com a cor correspondente.

5. crie uma função **drawaxis()** que desenha uma reta horizontal de cor preta a meio do ecrã, usando a função setpixel.

6. Crie uma função **plot(int *valores, unsigned red, unsigned green, unsigned blue)** que receba o endereço de um array com 64 valores (compreendidos entre -31 e +32) e usando a função setpixel desenhe o gráfico dessa sequência de valores centrado em torno do eixo horizontal.

7. Crie um programa que pergunte ao utilizador se quer: 1) limpar o ecrã ou 2) desenhar gráfico. No primeiro caso deve pedir os valores R, G,B e limpar. No segundo deve pedir o nome de um ficheiro que contém 64 linhas com valores que devem ser usadas para desenhar o gráfico e deve também pedir os valores R,G,B da cor que o gráfico deve ter. Chamadas sucessivas desta opção irá sobrepor vários gráficos. *Nota: no ficheiro auxiliar poderá encontrar vários ficheiros coma extensão .txt que deverá usar para testar.*

8. **(Opcional)** Crie duas funções para gerar sequências de valores que correspondam a uma onda quadrada e triangular respetivamente e adicione as respetivas opções no "menu"do programa.

**Exercício 3.5**

Escreva um programa que leia um ficheiro no formato PGM (P5) e mostre a imagem respetiva do mesmo no display. Use as imagens fornecidas no ficheiro auxiliar. Caso as imagens não ocupem toda o terminal gráfico, as restantes zonas deverão ficar com a cor preta.

A especificação do formato PGM diz o seguinte:

> Each PGM image consists of the following:
>
> A "magic number"for identifying the file type. A pgm image's magic number is the two characters "P5". Whitespace (blanks, TABs, CRs, LFs). A width, formatted as ASCII characters in decimal. Whitespace. A height, again in ASCII decimal. Whitespace. The maximum gray value (Maxval), again in ASCII decimal. Must be less than 65536, and more than zero. A single

whitespace character (usually a newline). A raster of Height rows, in order from top to bottom. Each row consists of Width gray values, in order from left to right. Each gray value is a number from 0 through Maxval, with 0 being black and Maxval being white. Each gray value is represented in pure binary by either 1 or 2 bytes. If the Maxval is less than 256, it is 1 byte. Otherwise, it is 2 bytes. The most significant byte is first.

# MIPS - Periféricos e interrupções

# 4

Em qualquer computador, além do processador e da memória, temos dispositivos periféricos com as mais variadas funções: controladoras gráficas, teclados, discos, impressoras, etc.. Uns servem para armazenar dados, outros para comunicar com outros computadores, outros para interação com o utilizador, e por aí fora. Tomemos o caso do teclado. Este dispositivo vai disponibilizar aos programas quais a tecla ou as teclas que o utilizador premiu em cada instante. Mas na realidade o utilizador só muito raramente (em termos de ciclos de instrução) é que prime uma tecla, Ou seja, mesmo para um utilizador experiente e bom dactilógrafo dificilmente ultrapassa as 10 teclas por segundo e mesmo assim entre duas teclas muitas instruções podem ser executadas pelos processadores actuais.

Efectuar um teste periódico de verificação de tecla premida é uma possibilidade, no entanto em periféricos mais rápidos isso pode tornar-se muito pesado computacionalmente e interferir negativamente na execução dos restantes programas.

Além disso incluir em programas complicados esse teste de verificação periódica era extremamente penoso. Por outro lado se for o sistema operativo a verificar periodicamente todos os dispositivos isto resultaria numa diminuição da disponibilidade do processador para executar os programas dos utilizadores.

Por essa razão, utiliza-se o principio de que quando um periférico precisa da atenção do sistema operativo (porque tem dados novos ou por outra razão) gera um pedido de interrupção ao processador. Em consequência deste pedido, o processador interrompe o fluxo normal das instruções do programa em execução e passa a executar um pedaço de código do sistema operativo que deverá tratar da comunicação com o dispositivo que gerou o pedido. A este código chamamos normalmente "rotina de serviço à in-

terrupção"(interrupt handler). Esta rotina ao ser executada vai ver qual a razão pela qual foi activada e processar o pedido do dispositivo, eventualmente invocando outras funções específicas que fazem parte daquilo a que chamamos "device driver".

## 4.1 Rotinas de serviço a interrupção do MIPS

Uma vez que um pedido de interrupção pode ocorrer a qualquer momento, as rotinas de serviço têm algumas particularidades:

1. não podem receber parâmetros, pois estes teriam de ser preparados e a característica assíncrona das interrupções não o permite.

2. todos os registos usados, têm de ser previamente guardados e restaurados antes de terminar o serviço à interrupção.

3. são excepção à regra anterior os registos **$k0** e **$k1**, uma vez que a convenção do uso de registos do MIPS reserva estes para as interrupções. Isto faz com que um programa que os use, pode vê-los alterados de forma inesperada.

No MIPS numa excepção a execução é transferida para o endereço $0x80000080$, pelo que a rotina de serviço deve iniciar aí. Para isso usa-se a directiva .ktext para dizer que a próxima instrução deve ser colocada no endereço indicado na zona do "kernel".

```
.ktext 0x80000080
        move ...
```

No MIPS as interrupções e excepções são tratadas pelo coprocessador 0 (só disponível em modo kernel). O coprocessador 0 tem registos especializados e que podem ser acedidos através das seguintes instruções:

| Instrução | Descrição |
|---|---|
| mfc0 Rdest, C0src | Copia o conteúdo do registo *C0src* do coprocessador para o registo *Rdest* |
| mtc0 Rsrc, C0dest | Copia o conteúdo do registo *Rdest* para o registo *C0src* do comprocessador |
| lwc0 c0dest, address | carrega palavra da memória (endereço address) para o registo *C0dest* |
| swc0 c0dest, address | guarda palavra na memória (endereço address) a partir do conteúdo do registo *C0dest* |

De entre os registos do coprocessador 0, os que são importantes para as interrupções ou excepções são:

| Nº Registo | Nome Registo | Descrição |
|---|---|---|
| 8 | BadVAddr | Endereço de memória onde ocorreu a excepção |
| 12 | Status | Mascara de interrupção, enable bits e status quando ocorreu a excepção |
| 13 | Cause | Tipo de excepção e bits de interrupção pendentes |
| 14 | EPC | Endereço da instrução que causou a excepção |

## O registo BadVAddr

Este registo, cujo nome é Bad Virtual Adderss, contém o endereço de memória onde ocorreu a excepção. Por exemplo, um acesso *não alinhado* a memória, irá gerar uma excepção e o endereço do acesso ficará armazenado neste registo.

## O registo Cause

Este registo fornece a informação de quais interrupções estão pendentes (ainda não tratadas) (IP2 a IP7) e qual a causa da excepção. O código da excepção é armazenado com um inteiro sem sinal usando os bits do 2 ao 6.

| 31-16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | IP7 | IP6 | IP5 | IP4 | IP3 | IP2 | IP1 | IP0 | | ExcCode | | | | | | |

Os bits IPi quando a 1, indicam se a interrupção de nível i ocorreu e está pendente, isto é ainda não foi tratada. O campo Excode indica o que causou a excepção:

| Código | Nome | Descrição |
|---|---|---|
| 0 | INT | Interrupt |
| 4 | ADDRL | Load em endereço ilegal |
| 5 | ADDRS | Store em endereço ilegal |
| 6 | IBUS | Error no bus durante fetch de instrução |
| 7 | DBUS | Error no bus durante acesso a dados |
| 8 | SYSCALL | Instrução *syscall* executada |
| 9 | BKPT | Instrução *break* executada |
| 10 | RI | Instrução reservada |
| 12 | OVF | Transbordo aritmético |

## O registo Status

O registo Status contém uma máscara de interrupção nos registos 15-10 e informação de estado nos bits 5-0. A organização deste registo é a seguinte:

| 31-16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7-6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | IM7 | IM6 | IM5 | IM4 | IM3 | IM2 | IM1 | IM0 | | KUo | IEo | KUp | IEp | KUc | IEc |

Os bits Imi são usados para activar ou desactivar respectivamente as interrupção do nível correspondente, quando colocados respectivamente a 1 ou 0.

Os bits Kuc, KUp, Kuo indicam se o processador está em modo kernel ou user, ou estava no estado anterior ou ainda num estado mais antigo. (uma espécie de stack).

Da mesma forma as instruções IEj ndicam se as interrupções estão activas, ou estavam no estado anterior, ou ainda num estado mais antigo.

## O registo EPC

O registo EPC tem para as excepções a mesma função que o registo $ra tem para as funções, ou seja o endereço de retorno. Assim o retorno de uma excepção externa (interrupção) faz-se da seguinte forma

```
        mfc0 $k0,$14
        rfe
```

```
        jr $k0
```

No caso de excepções internas (traps ou syscalls), EPC contém o endereço da instrução que gerou a excepção, logo o código de retorno neste caso terá de ser diferente:

```
        mfc0 $k0,$14
        addiu $k0,4
        rfe
        jr $k0
```

## 4.2 Exemplos

O código que se segue serve para activar as interrupções geradas por um dispositivo, neste caso um teclado. Note-se que o código tem duas partes, na primeira é a manipulação do status register com a activação apenas do bit correspondente às excepções externas (interrupções) e na segunda é num registo do próprio dispositivo activar o envio de pedidos de interrupção.

```
.data
ALL_INT_MASK: .word 0x0000ff00
KBD_INT_MASK: .word 0x00000100

.text
int_enable:
        mfc0 $t0, $12
        lw $t1, ALL_INT_MASK
        not $t1,$t1
        and $t0,$t0,$t1 # disable all int
        lw $t1, KBD_INT_MASK
        or $t0,$t0,$t1
        mtc0 $t0,$12

        # now enable interrupts on the KBD
        lw $t0,RCR
        li $t1, 0x00000002
        sw $t1 0($t0)
        jr $ra
```

O código seguinte é o que executado em resposta a excepções ou pedidos de interrupção.

```
        .kdata
save_t1: .word
save_t2: .word
save_s1: .word
save_s2: .word
save_s3: .word
save_s4: .word
save_a0: .word
save_v0: .word
save_at: .word

        .ktext 0x80000180
```

```
        #save every used register as needed
        move $k0,$at
        sw $k0, save_at

        sw $t1, save_t1
        sw $t2, save_t2
        sw $s1, save_s1
        sw $s2, save_s2
        sw $s3, save_s3
        sw $s4, save_s4
        sw $a0, save_a0
        sw $v0, save_v0

        mfc0 $k0,$13 # get cause register
        srl $t1,$k0,2
        andi $t1,$t1,0x1f # extract bits 2-6

        bnez $t1, non_int #

        andi $t2,$k0,0x00000100  # is bit 8 set?
        bnez $t2, receive
        andi $t2,$t1,0x00000200 # is bit 9 set?
        bnez $t2, transmit
        b iend

receive:
        # code to be written by you
        b iend

transmit:
        # code to be written by you
        b iend

non_int:
        mfc0 $k0,$14
        addiu $k0,$k0,4
        mtc0 $k0,$14

iend:
        lw $t1, save_t1
        lw $t2, save_t2
        lw $s1, save_s1
        lw $s2, save_s2
        lw $s3, save_s3
        lw $s4, save_s4
        lw $a0, save_a0
        lw $v0, save_v0

        lw $k0, save_at
        move $at,$k0
        mtc0 $zero,$13
        mfc0 $k0,$12
        andi $k0, 0xfffd
        ori $k0,0x0001
        mtc0 $k0,$12
        eret
```

## 4.3 Trabalho a realizar

Neste trabalho iremos usar o simulador de teclado e display existente nas ferramentas do MARS.

Leia atentamente a documentação (Help) desta ferramenta e resolva as questões que lhe são colocadas a seguir.

Como é explicado na documentação, há duas formas de usar este tipo de periférico: por verificação constante (polling) ou por interrupção.

**Exercício 4.1**
Escreva um programa em Assembly MIPS que envie para o display os caracteres introduzidos no teclado, convertendo as letras minúsculas em maiúsculas, mantendo inalterados outros caracteres.

**Exercício 4.2**
Explique claramente qual a diferença entre o acesso a um periférico por "polling"e por interrupção, não esquecendo de referir as vantagens ou desvantagens de cada um de forma clara.

**Exercício 4.3**
Utilizando simultaneamente o teclado e display mencionado acima, e o bitmap display usado na ficha anterior crie um programa que faça o seguinte: Vá preenchendo o "bitmap display"com cores sucessivas, por exemplo incrementando de 10 em 10 um valor que servirá oara preencher os pixeis de todo o "bitmap display". No final de cada ciclo, deverá verificar se foi premida alguma tecla na ferramenta "Keyboard and Display"e, usando o código da primeira alinea, ecoa-lo com a conversão de letras para maiúsculas.

**Exercício 4.4**
Repita a questão anterior usando interrupções para leitura do teclado, conversão e envio para o display, removendo assim a verificação de tecla premida no final de cada preenchimento com cor do "bitmap display".

Nota alguma diferença de desempenho? Porquê?

# Criação de sistema multitarefa

5

Os sistemas multi-tarefa requerem um temporizador que vá gerando interrupções a uma cadência predefinida. Essas interrupções (de relógio) vão activar uma parte importante do sistema operativo que é o escalonador, cuja responsabilidade é a de verificar se o processo em execução já esgotou o tempo máximo de execução e nesse caso trocar esse processo com outro que esteja pronto para executar.

## 5.1 Estados dos processos

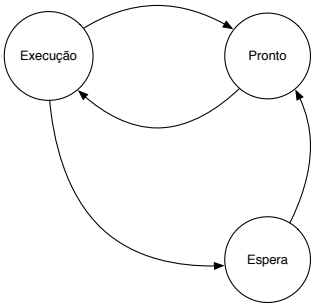



Figura 5.1: Estados dos processos e transições possíveis.

Em qualquer sistema multi-processo, os vários processos vão alternando entre diversos estados. Algumas dessas transições ocorrem a pedido do processo, outras em consequência indireta de pedido do processo, outras ainda involuntárias. A figura 5.1 representa de forma sintética os 3 estados mais importantes por que passa um processo durante a sua existência. A maioria destas transições são na verdade invisíveis para os processos, ou seja

os processos executam como se estivessem sempre no mesmo estado, ou seja nada é preciso fazer do ponto de vista da programação do mesmo para tratar estas transições de estado. Vejamos de seguida a que correspondem cada um dos estados e o que pode ocorrer para provocar a transição entre estes.

**Execução** Este estado corresponde ao de um processo (ou mais no caso multi-processador) que está a correr no CPU. As instruções deste estão ser executadas, como se não existisse nenhum outro processo. A saída deste estado para o estado de *Espera*, pode acontecer por pedir ao sistema operativo que o suspenda durante um determinado período de tempo, ou para aceder a um recurso que ainda não está disponível. A passagem para o estado *Pronto* acontece por o escalonador decidir que outro processo deve passar a executar, devido às políticas de escalonamento em uso.

**Pronto** Neste estado estão todos os processos que estão à espera de ser colocados em execução no processador. Ou seja, tirando o processador não lhes falta mais nada para poderem avançar.

**Espera** Neste estado estão todos os processos que aguardam um qualquer evento ou recurso. São colocados neste estado os processos que pedem para serem suspensos durante um período de tempo, que esperam dados ou acesso a um recurso. Para não haver espera activa são colocados neste estado até o sistema operativo decidir que devem voltar à competição pelo processador, por ter passado o tempo de espera, ou por já se encontrar disponível o dado ou recurso que faltava.

## 5.2 Trocas de processos

Quando o escalonador (activado por uma interrupção de relógio ou por uma chamada ao sistema, por exemplo) decide trocar o processo em execução, a que chamamos (A), por outro (B) e mais tarde volta a colocar o primeiro em execução, deve guardar e restaurar todo a informação do processo tal e qual estava entes das trocas. Isto permite ao processo ser completamente agnóstico em relação à partilha do processador no tempo.

A primeira questão é: **Que informação deve ser guardada na troca de processos?** Se pensar-mos que a troca pode acontecer em consequência de uma interrupção gerada por um temporizador que pode ocorrer num qualquer ponto da execução do processo no estado *Pronto*. A solução pode ser vista como a de fazer uma cópia do estado do processador que será restaurada mais tarde. Por outras palavras, teremos de guardar todos os

registos que tenham (ou possam ter) a ver com o estado do processo, para mais tarde restaurar os seus conteúdos.

Assim cada processo deverá ter uma estrutura em memória onde serão guardados cópias dos registos e eventualmente outras informações úteis sobre o processo. Essa estrutura é frequentemente chamada de Task Control Block (TCB) ou Process Control Block (PCB) e são organizadas normalmente como listas ligadas. A figura 5.2 ilustra o que pode ser uma das listas de processos *Pronto* ou *Espera*.

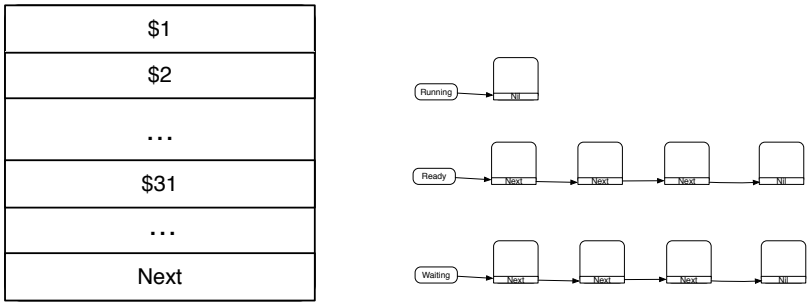



Figura 5.2: Esquerda: Exemplo de PCB minimalista; Direita: Filas de processos.

De notar que no PCB de cada processo há normalmente mais campos de informação, para além do espaço para guardar os registos. Essa informação adicional pode conter: Nome atribuído ao processo, prioridade, informação relativa ao uso de memória, estado de entradas e saídas usadas pelo processo, informação sobre ficheiros usados, etc..

## 5.3 Preparação

**Exercício 5.1**
Faça uma revisão dos conceitos sobre as interrupções. Para isso reveja os slides das aulas e faça uma leitura da ficha 4.

**Exercício 5.2**
Execute o código disponibilizado no ficheiro MMKD.asm e procure compreender o seu funcionamento, ativando previamente a ferramenta "Keyboard and Display MMIO Simulator".

*Sugestão:* Use o cursor "Run speed at max (no interaction)"para selecionar cerca de 2 instruções por segundo e coloque em execução. Repare o

que acontece quando prime uma tecla depois do programa iniciar e ficar no ciclo "forever". Procure perceber os detalhes.

## 5.4 Trabalho a realizar

Neste projeto pretende-se criar um sistema multi-programado com comutação entre tarefas por expiração de um temporizador. Esse temporizador normalmente seria assegurado pelo "timer"do MIPS, que por não estar disponível no MARS iremos usar a interrupção do teclado para simular um "clock tick".

**Exercício 5.3**
Crie uma função **Init** que deverá imprimir a mensagem "Iniciando Kernel Multi-tarefa". De seguida poderá iniciar quaisquer estruturas que sejam eventualmente necessárias.

**Exercício 5.4**
Crie a estrutura do PCB a usar e escreva o código que permite preencher os seus campos para a primeira execução de cada uma das tarefas que sejam adicionadas.

Faça isto criando uma função **AddThread** que recebe como parâmetro o endereço de inicio da tarefa.

*Nota:* Não esqueça de lhe atribuir uma zona de memória para a pilha cujo endereço deverá colocar no PCB na zona de salvaguarda do "stack pointer"tal como o endereço de inicio da tarefa deve ser armazenado na zona destinada ao "program counter".

**Exercício 5.5**
Escreva o código da rotina de serviço à interrupção que permitirá guardar o estado da tarefa corrente e trocá-la com outra pronta.

**Exercício 5.6**
Crie o código da função **startMultiThreading** que ao ser chamada vai activar o multiprocessamento, colocando todas as tarefas em execução paralela por partilha de tempo.

Note-se que também deverá ser criada a estrutura para a threadMain que corresponde código que chama esta função.

**Exercício 5.7**

Adicione e teste com o código abaixo num ficheiro separado "tasks.asm"que poderá ser substituído durante a apresentação do trabalho para conter tarefas mais complexas.

*Nota:* Teste usando a opção do MARS "Assemble all files in directory".

```
        .data
STRING_main: .asciiz "Tarefa Principal\n"
STRING_T0: .asciiz "Tarefa 0\n"
STRING_T1: .asciiz "Tarefa 1\n"
STRING_T2: .asciiz "Tarefa 2\n"

        .text
main:
#codigo de iniciacao
        jal Init

# AddThread (t0)
        la $a0,t0
        jal AddThread

# AddThread(t1)
        la $a0,t1
        jal AddThread

# AddThread(t2)
        la $a0,t2
        jal AddThread

# startMultiThreading()
        jal startMultiThreading

infinito:
        # faz qualquer coisa
        la $a0, STRING_main
        li $v0, 4
        syscall
        b infinito

# as tarefas
t0:     la $a0, STRING_T0
        li $v0, 4
        syscall
        b t0

t1:     la $a0, STRING_T1
        li $v0, 4
        syscall
        b t1

t2:     la $a0, STRING_T2
        li $v0, 4
        syscall
        b t2

# FIM
```

# MIPS - Instruções de virgula flutuante

6

O presente trabalho consiste em adaptar progressivamente o código fornecido para funcionar no simulador MARS. O código fornecido carrega um ficheiro de dados que contem os níveis de cinzento de uma imagem armazenados como números em vírgula flutuante. Essa imagem é de seguida convolvida com uma mascara que é também carregada a partir de um ficheiro. A imagem resultado é comparada com a armazenada num terceiro ficheiro. Se as duas imagens forem iguais considera-se que o resultado está correcto.

## 6.1 Revisão das operações aritméticas no MIPS

As operações aritméticas com inteiros podem ser feitas usando qualquer dos registos $0 a $31 do processador. Estas operações usam operandos de 32 bits sendo o resultado de 32 bits também.

As multiplicações podem gerar resultados maiores do que os representáveis por 32 bits. A arquitectura MIPS fornece dois registos especiais que são (sempre) usados como destino das operações de multiplicação (e divisão). Estes registos são designados por **hi** e **lo** indicando que contém os 32 bits mais significativos e os 32 bits menos significativos do resultado. No caso da divisão o registo **lo** fica com o quociente e o registo **lo** com o resto. Existem instruções dedicadas a mover os valores entre estes registos e os registos gerais do MIPS.

| Instrucão | resultado |
|---|---|
| mfhi %X | %X ← hi |
| mflo %X | %X ← lo |
| mthi %X | hi ← %X |
| mtlo %X | lo ← %X |

As operações aritméticas podem ser efectuadas considerando ou não a representação de números negativos. Assim dizemos que estamos a usar aritmética com sinal ou sem sinal.

No caso da aritmética com sinal os números de 32 bits consideram-se como representados na notação de complemento para 2. Neste caso as operações (adição e subtracção) podem gerar "overflow".

No caso da aritmética sem sinal, os números são considerados estando na representação binária padrão. As instruções neste caso nunca geram o erro de "overflow"mesmo que o "overflow"ocorra.

## 6.2 Aritmética em virgula flutuante

As operações aritméticas em vírgula flutuante são realizadas pelo comprocessador 1 do MIPS (IEEE-754 floating point standard). Este dispõe de 32 registos designados de $f0 a $f31, sendo cada um deles de 32 bits. Para as operações com números de dupla precisão (double) agrupam-se os registos aos pares, 0 com 1, 2 com 3, ..., 30 com 31. Por questões de simplicidade de arquitectura, as operações em vírgula flutuante são realizadas sobre registos pares. Os registos impares são usados para os 32 bits menos significativos de números de dupla precisão. No caso em que o resultado de uma operação em vírgula flutuante gera um expoente demasiado grande para ser representado no campo correspondente, é gerado um erro de overflow.

Além das operações aritméticas em virgula flutuante, o conjunto de instruções do MIPS contém ainda outras operações como comparações, saltos condicionais e acesso à memória (load, store). Ao passo que nas comparações entre inteiros, o resultado vai para um registo designado, na unidade de vírgula flutuante estas afectam uma "flag"de condição, a qual pode ser testada pelos saltos condicionais.

| Instrução | Efeito |
|---|---|
| mfc1 Rdest, FPsrc | Copia o conteúdo do registo de virgula flutuante FPsrc para o registo inteiro Rdest |
| mtc1 Rsrc, FPdst | O conteúdo do registo inteiro Rsrc é copiado para o registo de virgula flutuante FPdst |
| mov.x FPdst, FPsrc | Copia entre registos de virgula flutuante |
| lwc1 FPdest, C($y) | Carrega palavra do endereço dado por $C + \$y$ para o registo FPdest |
| swc1 FPsrc, C($y) | Guarda palavra contida no registo FPsrc no endereço dado por $C + \$y$ |
| add.x FPdest, FPsrc1, FPsrc2 | adição |
| sub.x FPdest, FPsrc1, FPsrc2 | subtração |
| mul.x FPdest, FPsrc1, FPsrc2 | multiplicação |
| div.x FPdest, FPsrc1, FPsrc2 | divisão |
| abs.x FPdest, FPsrc1, FPsrc | valor absoluto |
| neg.x FPdest, FPsrc1, FPsrc | negar valor |
| c.eq.x FPsrc1, FPsrc2 | Põe a 1 a flag de condição FP se $FPsrc1 == FPsrc2$ |
| c.le.x FPsrc1, FPsrc2 | Põe a 1 a flag de condição FP se $FPsrc1 <= FPsrc2$ |
| c.lt.x FPsrc1, FPsrc2 | Põe a 1 a flag de condição FP se $FPsrc1 < FPsrc2$ |
| bclt label | salta se a flag de condição FP for verdadeira |
| bclf label | salta se a flag de condição FP for falsa |
| cvt.x.w FPdest, FPsrc | converte o inteiro em FPsrc para virgula flutuante |
| cvs.w.x FPdest, FPsrc | converte o numero em virgula flutuante em FPsrc para inteiro |
| cvs.d.s FPdest,FPsrc | converte o número em precisão simples em FPsrc para precisão dupla |
| cvs.s.d FPdest,FPsrc | converte o número em precisão dupla em FPsrc para precisão simples |

Nota: o x em instruções como mov.x será substituído por d ou s consoante a operação seja de precisão dupla ou simples.

## 6.3 Exercícios

### Antes da aula

**Exercício 6.1**
Compile e teste o código fornecido nas máquinas mips.deec.uc.pt (10.10.120.2) e lsrc.deec.uc.pt (10.10.120.1). Que resultados obteve e porquê?

**Exercício 6.2**
Na máquina mips execute o comando "make seebytes". Este irá criar um programa progseebytes.mips (a seguir ao ponto vem o nome do computador em uso). De seguida execute o programa criado.

Repita a operação na maquina lsrc. Procure explicar as diferenças nos resultados.

**Exercício 6.3**
Escreva uma função em assembly mips que receba um número no formato double e o devolva com os seus bytes trocados.

**Exercício 6.4**
Escreva uma função em assembly que transforme as matrizes carregadas para a ordem de bytes correcta. Note que as matrizes contêm valores armazenados como double e foram gravadas numa máquina Intel. Essa função deverá ter o seguinte cabeçalho void (double * data, int rows, int cols); Modifique a função *main* de forma a chamar a função acima para permitir o bom funcionamento do programa na máquina **mips**.

**Exercício 6.5**
Comente a função edge_detector escrita em C e escreva uma função em assembly do MIPS que faça a mesma operação

### Na aula

**Exercício 6.6**
Escreva em assembly MIPS as funções que permitam substituir as funções **open**, **read**, **close**, e que funcionem no simulador MARS.

**Exercício 6.7**
Escreva a função main em assembly e ajuste o restante código de forma a que possa ser executado no simulador MARS

**Exercício 6.8**
Analise o seu funcionamento no simulador sobretudo no que diz respeito ao desempenho da cache.

**Exercício 6.9**
Elabore um relatório onde deverá apresentar e discutir os resultados.

# Assembly Mips em Linux 7

O processador MIPS é um processador com registos de uso geral, ou seja qualquer registo pode ser usado em qualquer instrução. No entanto , linguagens de alto nível como o C têm requisitos que levaram ao estabelecimento de convenções no uso desses registos. Vejamos alguns aspectos que advêm da forma como a linguagem C define para o uso de funções.

- Uma função poderá ter qualquer número de parâmetros de entrada
- uma função poderá retornar um valor de saída
- Os parâmetros de entrada poderão ser valores a utilizar directamente ou endereços (ponteiros) para dados a processar
- Uma função poderá usar variáveis locais, i.e. variáveis que são criadas quando a função é chamada e destruídas quando esta termina a execução.
- Não há qualquer limite predefinido para o encadeamento de chamadas de funções, ou seja uma função pode chamar outra, que por sua vez chama outra, e essa chama outra...

Sendo o número de registos do processador limitado, como garantimos que os valores que estavam num registo antes da chamada de uma função não foram alterados quando essa função termina?

Para isto foram definidas algumas convenções e regras. A primeira convenção é o uso de uma zona de memória chamada pilha. A pilha, além das zonas para código (instruções) e dados. Esta é acedida através de um ponteiro para o topo dessa pilha, que vai crescendo ou diminuindo ao longo da execução de um programa. Assim, cada função ao ser chamada vai acrescentar à pilha dados que serão retirados quando termina. Percebe-se assim facilmente que é o lugar natural para armazenar as variáveis locais.

Também quando é necessário preservar o valor de um registo, este deve ser guardado na pilha antes de ser usado em qualquer operação, sendo possível no final recuperar o seu valor inicial.

A pilha é igualmente usada para guardar o endereço de retorno na chamada de funções. Quando em determinada parte do código chamamos uma função, pretendemos que, quando a execução desta terminar, a execução continue na instrução que se segue à que chamou essa função. Por isso é necessário guardar o endereço de retorno. Embora o MIPS guarde esse endereço num dos registos ($ra), a possibilidade de encadeamento na chamada de funções faz com que sejamos obrigados a guardar esses endereços na pilha.

Também a passagem de parâmetros é feita através da pilha. A convenção do C diz que os parâmetros são colocados na pilha do último para o primeiro antes da chamada de uma função e são retirados após o retorno da função. (Note que outras linguagens como o Pascal ditam que seja a própria função que foi chamada quem deve libertar da pilha o espaço ocupado pelos parâmetros recebidos).

## 7.1 Passagem de parâmetros

Dado que o MIPS tem muitos registos, convencionou-se que até 4 parâmetros, estes serão passados através de registos. Todos os restantes seguem a convenção normal, que é, de os passar pela pilha. No entanto, compiladores como o gcc reservam na pilha espaço para todos os parâmetros, mesmo aqueles que são passados por registos.

Exemplo de uma função em assembly MIPS que pode ser chamada à partir de código em C

```
        .globl soma
        .ent soma
        .type soma,@function
        .text
soma:
        # os dois inteiros estao em $a0 e $a1
        # o resultado deve sair no $v0
        add $3,$4,$5
        jr $31
        .end soma
```

Note-se que no código acima não houve qualquer cuidado em salvar o valor dos registos na pilha pois, dada a simplicidade do código nenhum registo extra é utilizado. Esta função pode ser utilizada a partir do seguinte código

```
extern int soma(int,int);
```

```
/* declaracao da funcao, que normalmente
e' feita num ficheiro .h*/

int main (){
        int res;
        res=soma(3,5);
        printf("O resultado e: %d\n", res);
}
```

Vejamos agora um exemplo onde se usam 5 parâmetros de entrada

```
        .globl soma5
        .ent soma5
        .type soma5,@function
        .text
soma5:
        #reservar espaco na pilha e salvar $sp e $fp
        addiu   $sp,$sp,-8
        sw      $fp,0($sp)
        move    $fp,$sp

        addu    $2,$4,$5
        addu    $2,$2,$6
        addu    $2,$2,$7
        # vamos buscar o quinto
        lw      $8,24($sp)
        nop
        addu    $2,$2,$8

        #recuperar $sp e  $fp e libertar espaco na pilha
        move    $sp,$fp
        lw      $fp,0($sp)
        addiu   $sp,$sp,8
        jr      $31
        nop
        .end soma5
```

## 7.2 Código PIC

Todo o código dos programas que correm em Linux (user mode) é obrigatoriamente PIC (Position Independent Code). Isto significa que esse código não deve fazer referências a posições absolutas da memoria (seja para dados ou saltos) mas usar sempre deslocamentos em relação à posição corrente. O código PIC segue a convenção de que qualquer função ao ser chamada recebe o seu próprio endereço no registo $t9 ($25). Usando assim o endereço guardado em $t9 pode-se calcular o endereço da tabela GOT (global offset table). Isso tem o seguinte aspecto

```
function:
        .set    noreorder
        .cpload $25
        .set    reorder
```

.cpload calcula o endereço de GOT. O assembler obriga a que o a macro .cpload esteja numa secção "noreorder". Note-se que .cpload é apenas necessária em funções que referenciem endereços globais (dados ou código). Assim a função seguinte não requer uso de .cpload:

```
int add(int a, int b) {
        return a + b;
}
```

Mas por sua vez a função que se segue já obriga ao seu uso.

```
int var;
int get_ptr(void) {
        return &var;
}
```

O registo $gp ($28), chamado global pointer, pertence ao grupo dos registos cujo valor à saída de uma função deve ser exactamente o mesmo que tinha aquando da entrada. Assim sempre que o seu valor é modificado este deve ser restaurado antes de terminar a função. O assembler faz isto automaticamente quando se usa .cprestore. Esta macro usar um argumento que corresponde ao offset na pilha onde o valor do GP é armazenado. Juntando tudo, o assembler reclama se não colocarmos .frame, .ent e .end.

```
foo:    .ent foo
        .frame $29,32,$31
        .set    noreorder
        .cpload $25
        .set    reorder
        subu    $29, $29, 32
        .cprestore 16
...
        .end foo
```

os argumentos de .frame são o "framepointer", o tamanho do stack frame e o registo $ra.

Embora por definição as instruções de salto tenham um "delay slot" no MIPS, aparentemente isso não é verdade. De facto experimentamos colocar a seguir a uma instrução branch uma outra qualquer instrução e verificamos que quando o salto é tomado essa instrução não é executada.

No exemplo seguinte temos a função "void nope(void);" escrita em assembly que é chamada do programa em C abaixo.

```
        .globl nope
        .type nope,@function
nope:
        beq     $4,$0,zero
        addi    $4,$4,100
zero:   move    $2,$4
        jr      $31
```

```
#include <stdio.h>
main(){
  int a,b;
  a=nope(0);
  b=nope(1);
  printf("%d %d\n",a,b);
}
```

O que seria de esperar é que os valores impressos no ecrã fossem 100 e 101 quando na realidade aparece 0 e 101. Ou seja aparentemente a instrução addi não foi executada no "delay slot".

Se no entanto após compilarmos o código da função nop com "gcc –c nope.s", podemos usar o comando objdump –d nope.o e eis que obtemos o seguinte:

```
pm@mips:~/AC$ objdump -d nope.o

nope.o:     file format elf32-tradbigmips

Disassembly of section .text:

00000000 <nope>:
   0:   10800002        beqz    a0,c <zero>
   4:   00000000        nop
   8:   20840064        addi    a0,a0,100

0000000c <zero>:
   c:   03e00008        jr      ra
  10:   00801021        move    v0,a0
        ...
```

Como podemos observar surgiu uma instrução nop a seguir à instrução de salto. No entanto se acrescentarmos a directiva .set noreorder no código acima

```
        .globl nope
        .type nope,@function
nope:   .set noreorder
        beq $4,$0,zero
        addi $4,$4,100
zero: move $2,$4
        jr $31
```

o resultado será

```
objdump -d nope.o

nope.o:     file format elf32-tradbigmips

Disassembly of section .text:

00000000 <nope>:
   0:   10800001        beqz    a0,8 <zero>
   4:   20840064        addi    a0,a0,100

00000008 <zero>:
```

```
8:   00801021        move    v0,a0
c:   03e00008        jr      ra
```

Agora como se poderá esperar o resultado a aparecer no ecrã é 100 101.

## 7.3 Exercícios

Os exercícios que se seguem destinam-se a ser executados numa máquina mips com sistema operativo linux. Para isso deve-se ligar a partir de linux ou Mac OS usando ssh aXXXXXXXX@mips.deec.uc.pt (ou mips.labs.deec.uc.pt se a ligação for feita dentro da rede do DEEC), onde XXXXXXXX deve ser substituído pelo número de aluno e cuja senha é a mesma da alunos.deec.uc.pt.

Ao ligar-se desta forma, terá acesso a uma "shell"com todos os comandos do linux (sem interface gráfica). Para editar os ficheiros use o emacs ou o nano.

### Antes da aula

O compilador a utilizar é o gcc, que tanto suporta código C como assembly, desde que os ficheiros tenham as extensões **.c** ou **.s** respectivamente.

Leia a documentação do MIPS no que diz respeito às convenções de passagem de parâmetros, e convenções de gestão da pilha (stack).

**Exercício 7.1**
Crie um ficheiro com uma função em C que receba dois números como parâmetros e devolva o resultado da soma destes.

**Exercício 7.2**
Usando o gcc gere o código assembly MIPS (terá de ser numa máquina MIPS com linux) usando o comando "gcc -O0 -S funcao.c -o funcao.s". analise cada uma das linhas do ficheiro resultante. Procure explicar cada uma destas.

Prepare roda a informação necessária para resolver os exercícios que se seguem no decorrer da aula.

### Na aula

**Exercício 7.3**
Usando o código fibonacci da aula anterior, modifique-o de forma a torná-lo numa função escrita em assembly que respeite as convenções do C. De

seguida crie um ficheiro main.c com a função main que deverá pedir um número ao utilizador e chamar a função fibonacci (do ficheiro assembly) para obter o resultado e imprimi-lo no ecrã. Compile-os com "gcc fibo.s main.c -o meuprog1". Teste a sua execução e se ocorrer algum erro, utilize o debugger "gdb".

**Exercício 7.4**
Escreva e teste um programa que vá pedindo ao utilizador para introduzir números inteiros e vá calculando a sua soma. A função que calcula a soma deverá estar escrita em assembly. O programa deve terminar e apresentar o resultado quando o utilizador introduzir o valor 0.

**Exercício 7.5**
Escreva um programa que peça ao utilizador para introduzir um número inteiro e apresente o valor do seu factorial. Nota o calculo do factorial deve ser efectuado de forma iterativa numa função escrita em assembly, O restante programa deverá ser escrito em C.

**Exercício 7.6**
Escreva um programa que calcule e imprima todos os números primos inferiores a um valor dado pelo utilizador. O ficheiro em C deve conter além da função main uma função printint que imprime no ecrã o inteiro recebido como parâmetro. Esta função deverá ser chamada do código assembly sempre que for encontrado um número primo.

# A pilha e seu uso na programação com funções

8

A pilha (stack) é uma zona de memória usada para dados temporários. O programador deve-se assegurar que ao terminar uma função, a pilha deve ficar exactamente no mesmo estado em que se encontrava na entrada da mesma função. A linguagem C (e C++) faz uso da pilha para passagem de parâmetros às funções. Assim uma função "int a()"que chama uma função "int b(int p1, int p2, char p3)", deve colocar na pilha os parâmetros p1, p2 e p3, pela ordem inversa dos mesmos. O valor de retorno faz-se normalmente através de um registo.

No caso do processador MIPS, porque há um grande número de registos gerais e se pretende minimizar o mais possível os acessos a memória, os primeiros 4 parâmetros são passados directamente pelos registos $4a7$, os restantes se existirem serão colocados na pilha usando a convenção acima descrita. Para o retorno de valores de uma função usa-se o registo $2 e em caso de necessidade (números de dupla precisão por ex) o registo $3. Vejamos um exemplo:

O ficheiro call.c contém o seguinte código.

```
extern int soma(int, int, int, int, int, int,int);
int call(){
        int res;
        res=soma(1,2,3,4,5,6,7);
        return res;
}
```

E o ficheiro soma.c contém o seguinte código:

```
int soma(int p1, int p2, int p3, int p4, int p5, int p6, int p7){ i
        nt res;
        res=p1+p2+p3+p4+p5+p6+p7;
        return res;
}
```

Compilando agora ambos da seguinte forma:

```
gcc -mno-explicit-relocs -mno-abicalls -S call.c
gcc -mno-explicit-relocs -mno-abicalls -S soma.c
```

Obtemos o ficheiro call.s com o seguinte conteúdo

```
        .file 1 "call.c"
        .section .mdebug.abi32
        .previous
        .text
        .align 2
        .globl call
        .ent call
        .type call, @function
call:
        .frame $fp,48,$31
        .mask 0xc0000000,-4 .fmask 0x00000000,0
        addiu $sp,$sp,-48
        sw $31,44($sp)
        sw $fp,40($sp)
        move $fp,$sp
        li $2,5
        sw $2,16($sp)
        li $2,6
        sw $2,20($sp)
        li $2,7
        sw $2,24($sp)
        li $4,1
        li $5,2
        li $6,3
        li $7,4
        jal soma
        sw $2,32($sp)
        lw $2,32($sp)
        move $sp,$fp
        lw $31,44($sp)
        lw $fp,40($sp)
        addiu $sp,$sp,48
        j $31
        .end call
        .ident "GCC: (GNU) 4.1.2 # vars= 8, regs= 2/0, args= 32, gp= 0 # 0x5 # 0x6 # 0x7 # 0
prerelease) (Debian 4.1.1-21)"
```

e o ficheiro soma.s contém

```
        .file 1 "func.c"
        .section .mdebug.abi32
        .previous
        .text
        .align 2
        .globl soma
        .ent soma
        .type soma, @function
soma:
        .frame $fp,16,$31 # vars= 8, regs= 1/0, args= 0, gp= 0 .mask 0x40000000,-8
        .fmask 0x00000000,0
        addiu $sp,$sp,-16
        sw $fp,8($sp)
        move $fp,$sp
        sw $4,16($fp)
```

```
        sw $5,20($fp)
        sw $6,24($fp)
        sw $7,28($fp)
        lw $3,16($fp)
        lw $2,20($fp)
        addu $3,$3,$2
        lw $2,24($fp)
        addu $3,$3,$2
        lw $2,28($fp)
        addu $3,$3,$2
        lw $2,32($fp)
        addu $3,$3,$2
        lw $2,36($fp)
        addu $3,$3,$2
        lw $2,40($fp)
        addu $2,$3,$2
        sw $2,0($fp)
        lw $2,0($fp)
        move $sp,$fp
        lw $fp,8($sp)
        addiu $sp,$sp,16
        j $31
        .end soma
        .ident "GCC: (GNU) 4.1.2 20061115 (prerelease) (Debian 4.1.1-21)"
```

É importante conhecer a convenção usada pelos compiladores de C para o processador MIPS no que diz respeito à utilização dos registos. Esta convenção é descrita pela tabela da figura 8.1.

A figura 8.2 ilustra a utililização da pilha aquando da chamada de uma função.

## 8.1 Exercícios

**Exercício 8.1**
Quais os valores que ficam colocados em cada uma das posições da pilha?

**Exercício 8.2**
Escreva uma função em assembly que receba 6 parâmetros inteiros e devolva a soma dos quadrados desses parâmetros. Teste o seu funcionamento na máquina mips chamando essa função a partir da linguagem C.

**Exercício 8.3**
Execute o exercício anterior usando o simulador MARS.

| Register name | Number | Usage |
|---|---|---|
| $zero | 0 | constant 0 |
| $at | 1 | reserved for assembler |
| $v0 | 2 | expression evaluation and results of a function |
| $v1 | 3 | expression evaluation and results of a function |
| $a0 | 4 | argument 1 |
| $a1 | 5 | argument 2 |
| $a2 | 6 | argument 3 |
| $a3 | 7 | argument 4 |
| $t0 | 8 | temporary (not preserved across call) |
| $t1 | 9 | temporary (not preserved across call) |
| $t2 | 10 | temporary (not preserved across call) |
| $t3 | 11 | temporary (not preserved across call) |
| $t4 | 12 | temporary (not preserved across call) |
| $t5 | 13 | temporary (not preserved across call) |
| $t6 | 14 | temporary (not preserved across call) |
| $t7 | 15 | temporary (not preserved across call) |
| $s0 | 16 | saved temporary (preserved across call) |
| $s1 | 17 | saved temporary (preserved across call) |
| $s2 | 18 | saved temporary (preserved across call) |
| $s3 | 19 | saved temporary (preserved across call) |
| $s4 | 20 | saved temporary (preserved across call) |
| $s5 | 21 | saved temporary (preserved across call) |
| $s6 | 22 | saved temporary (preserved across call) |
| $s7 | 23 | saved temporary (preserved across call) |
| $t8 | 24 | temporary (not preserved across call) |
| $t9 | 25 | temporary (not preserved across call) |
| $k0 | 26 | reserved for OS kernel |
| $k1 | 27 | reserved for OS kernel |
| $gp | 28 | pointer to global area |
| $sp | 29 | stack pointer |
| $fp | 30 | frame pointer |
| $ra | 31 | return address (used by function call) |

Figura 8.1: Uso de cada um dos registos do MIPS na linguagem C

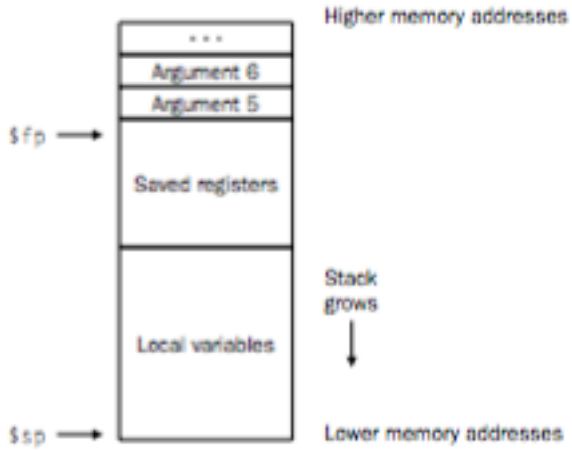



Figura 8.2: Utilização da pilha por uma função, ou quadro (frame) de uma função

# Representação de dados 9

## 9.1 Exercícios

**Exercício 9.1**
Escreva uma função em assembly MIPS que dado um inteiro produza uma cadeia de caracteres (string) que contenha a representação desse inteiro numa base à escolha. Assim a função deve receber três argumentos int numascii(int num, char * outstr, int base);

O primeiro parâmetro corresponde ao número de entrada, o segundo à cadeia de caracteres de saída e o terceiro à base em que se pretende representar o número (até 16). A função deverá retornar o número de caracteres que foram necessários para representar o número, ou -1 em caso de erro.

A seguir lista-se o código em C que deverá chamar essa função

```
#include <stdio.h>
extern int numascii(int num, char * outstr, int base);
int main() {
        char str [200];
        int num,base;
        printf("Introduza o numero\n");
        scanf ("%d",&num);
        printf("Qual a base a utilizar (2-16)?");
        scanf ("%d " , &base);
        if (numascii(num,str,base)<0)
                printf("Erro na conversao\n");
        else
                printf("O numero %d na base %d e' %s\n",
                                num,base,str) ;
}
```

**Exercício 9.2**
Diferentes processadores, usam ordens para os bytes diferentes. Isto significa que a forma como os bytes de um inteiro estão armazenados na memória,

depende do processador. Assim temos essa ordem pode ser "big endian"ou "little endian". Faça uma pequena função em C que devolva 1 ou 0 consoante o processador onde foi executada seja ou não "big endian". Compile e teste nas máquinas lsrc e mips.

**Exercício 9.3**

Pelas razões referidas na questão anterior, um ficheiro de inteiros criado numa máquina poderá ter de ser convertido em outra máquina. Porquê?

**Exercício 9.4**

Escreva a função "int swapint(int a)"em assembly mips que receba um inteiro e o devolva depois de alterada a ordem dos bytes. Escolha a forma mais adequada de demonstrar que o seu funcionamento está correcto.

**Exercício 9.5**

Escreva uma função (em assembly MIPS) semelhante à anterior mas que receba o endereço de um array de inteiros e o número de elementos desse array. Essa função deverá trocar a ordem dos bytes de cada um dos inteiros desse array.

**Exercício 9.6**

Compile e execute o seguinte programa na máquina lsrc,

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

int main(){
int i, fd;
int array[100];

        for (i=0;i<100;i++)
                array[i]=i*i;

        fd=open("nums.dat",O_CREAT|O_WRONLY|O_TRUNC,0666);
        if(fd<0){
                perror("Abertura do ficheiro");
                exit(1);
        }
        i=write(fd,array,100*sizeof(int));
        if(i<0){
                perror("Escrita no ficheiro");
                exit(1);
        }
        close(fd);
        printf("Ficheiro de dados criado com sucesso\n");
        exit(0);
}
```

de seguida complete o programa seguinte, compile-o juntamente com a função desenvolvida na alinea anterior e verifique o resultado ao corre-lo na máquina mips.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

int main(){
int i, fd;
int array[100];

        fd=open("nums.dat",O_RDONLY);
        if(fd<0){
                perror("Abertura do ficheiro");
                exit(1);
        }
        i=read(fd,array,100*sizeof(int));
        if(i<0){
                perror("Leitura do ficheiro");
                exit(1);
        }
        close(fd);

// imprima os valores dos inteiros

// chame a funcao swaparray da questao anterior

// imprima novamente os inteiros
}
```

# Processamento de imagem usando funções escritas em assembly

10

Algumas das funções mais frequentes aplicadas ao processamento de imagem baseiam-se na convolução das mesmas com "kernels". Esta operação consiste em aplicar sobre cada pixel o "kernel"centrado nesse pixel e multiplicar os pixeis sob o kernel pelos valores do mesmo.

Considerando como exemplo uma imagem $10 \times 10$ cujos valores de luminância (nível de cinzento) são os seguintes

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $aa$ | $ab$ | $ac$ | $ad$ | $ae$ | $af$ | $ag$ | $ah$ | $ai$ | $aj$ |
| $ba$ | $bb$ | $bc$ | $bd$ | $be$ | $bf$ | $bg$ | $bh$ | $bi$ | $bj$ |
| $ca$ | $cb$ | $cc$ | $cd$ | $ce$ | $cf$ | $cg$ | $ch$ | $ci$ | $cj$ |
| $da$ | $db$ | $dc$ | $dd$ | $de$ | $df$ | $dg$ | $dh$ | $di$ | $dj$ |
| $ea$ | $eb$ | $ec$ | $ed$ | $ee$ | $ef$ | $eg$ | $eh$ | $ei$ | $ej$ |
| $fa$ | $fb$ | $fc$ | $fd$ | $fe$ | $ff$ | $fg$ | $fh$ | $fi$ | $fj$ |
| $ga$ | $gb$ | $gc$ | $gd$ | $ge$ | $gf$ | $gg$ | $gh$ | $gi$ | $gj$ |
| $ha$ | $hb$ | $hc$ | $hd$ | $he$ | $hf$ | $hg$ | $hh$ | $hi$ | $hj$ |
| $ia$ | $ib$ | $ic$ | $id$ | $ie$ | $if$ | $ig$ | $ih$ | $ii$ | $ij$ |
| $ja$ | $jb$ | $jc$ | $jd$ | $je$ | $jf$ | $jg$ | $jh$ | $ji$ | $jj$ |

e considerando um kernel $3 \times 3$ com valores

| | | |
|---|---|---|
| $R$ | $S$ | $T$ |
| $U$ | $V$ | $X$ |
| $Y$ | $Z$ | $W$ |

o calculo do píxel da imagem destino na posição (1,1) é dado por $aa \times R + ab \times S + ac \times T + ba \times U + bb \times V + bc \times X + ca \times Y + cb \times Z + cc \times W$.

## 10.1 Trabalho a realizar

Partindo do código fornecido crie as funções em assembly MIPS necessárias.

**Exercício 10.1**
Crie em assembly uma função de binarização, ou seja que a partir de uma imagem com valores de 0 a 255 produza uma imagem apenas com valores 0 e 255. Se o valor de um píxel for inferior ao limiar dado é atribuido ao pixel correspondente na imaem de saída o valor 0, caso contrário o valor será 255. Essa função será utilizada a partir de código em C e terá o seguinte cabeçalho

void binariza(Image * in, unsigned char threshold, Image * out)

O tipo Image é definido como sendo uma estrutura com os seguintes campos

```
struct IMAGE {
        int rows;
        int cols;
        unsigned char * data;
        int widthStep;
        };
typedef struct IMAGE Image;
```

É de notar que o campo widthStep corresponde à separação em bytes dos primeiros pixeis de duas linhas consecutivas, e é usado por questões de alinhamento das linhas da imagem.

**Exercício 10.2**
Acrescente agora uma função que efectue a convolução explicada anteriormente. Essa função terá como cabeçalho

void conv(Image * int, Image * kernel, Image * out)

**Exercício 10.3**
Crie uma nova versão que efectue a mesma operação de convolução, mas admitindo kernel com valores expressos em virgula flutuante. As imagens de entrada e de saída mantém o tipo de dados.

void convf(Image * int, Imagef * kernel, Image * out)

# Optimização e análise de desempenho 11

Em muitos casos por razões de projecto não é possível utilizar processadores de última geração. Isto obriga, muitas vezes, a que o código desenvolvido seja optimizado de forma tirar o máximo partido do processador utilizado. Para isso teremos de por vezes sacrificar a estruturação ou a legibilidade do código que vamos escrever, a favor da rapidez de execuão do mesmo. Muitas das técnicas de optimização de código baseiam-se em reformular os algoritmos ou reescrever as expressões de forma a eliminar (sempre que possível) chamadas a funções que sejam consumidoras de tempo tais como: exponenciais, raízes quadradas e funções transcendentais.

As notas que se seguem são apenas um resumo de algumas das técnicas mais utilizadas, e não dispensam a consulta de outras fontes de informação especializadas.

## 11.1 Optimizar ou não optimizar

O sobredimensionamento não é necessariamente bom! Imaginemos que queremos controlar a temperatura de uma sala. Não faz sentido efectuar as medições e produzir os respectivos comandos de controlo a uma taxa de 1Hz, porque devido à inércia térmica da massa de ar envolvida quaisquer variações acontecerão a taxas muito mais lentas. Por outro lado, se quisermos utilizar para o controlo uma contribuição da derivada da variação da temperatura para obter um efeito antecipativo e se o período de amostragem for demasiado pequeno as variações medidas serão praticamente nulas.

Pelo que acabámos de ver, torna-se evidente que antes de enveredar por qualquer tentativa de optimização de código para o tornar mais rápido, devemos certificarmo-nos da absoluta necessidade disto. Pois, se o processo

que pretendemos colocar em funcionamento necessitar apenas de uma taxa de execução de 0.1Hz poderá não ser necessário optimizar o código de forma a torná-lo capaz de executar 10 vezes mais rápido.

## 11.2 Optimizações simples

### Aritmética inteira

Na grande maioria dos casos, as operações de aritmética inteira são bastante mais rápidas do as efectuadas em vírgula flutuante pelo que as primeiras deverão ter escolha preferencial sempre que possível.

No entanto, se numa aplicação for absolutamente necessário utilizar números décimais, podemos aplicar um factor de escala ($\times 10^n$ ou $\times 2^m$) nos valores utilizados, convertendo-os em inteiros e efectuando todas as operações como se de inteiros se tratasse.

Claro que na realidade teremos de introduzir operações adicionais para manter o número de casas decimais. Exemplo: considerando 3 casas decimais fixas, então 1000 representa 1.000. Mas $1000 \times 1000 = 1000000$ e 1000000, nesta notação, é 1000.000 e não 1.000 como seria o resultado esperado, daí que seja necessário efectuar um ajuste do número de casas decimais depois de cada multiplicação ou divisão.

### Eliminar indução nos ciclos

Como exemplo podemos ver o código seguinte

```
for (i=0; i<=10;i++)
        a[i+1]=1;
```

o qual pode ser substituído sem qualquer prejuízo por

```
for (i=1; i<=11;i++)
        a[i]=1;
```

tendo eliminado uma soma desnecessária.

### Optimizar o caso mais frequente.

As porções de código que sáo mais vezes executadas devem merecer atenção especial, pois são estas, na maioria dos casos, que consomem mais tempo de processamento.

### Testes encadeados

Um conjunto de instruções **if** deve ser rearranjado de forma a que o caso que se verifíca mais vezes seja o primeiro a ser testado

### Usar expressões simples

Substituir comparações entre resultados de funções monótonas, por testes sobre os parâmetros destas funções, eliminando assim o calculo das mesmas. Por exemplo, em vez de utilizar a comparação

```
if (ex < ey) then{
...
}
```

por

```
if (x < y) then {
 ...
 }
```

uma vez que produz o mesmo resultado.

## 11.3 Reduzir a carga computacional nos ciclos

É nos ciclos que se faz a maior parte do processamento e é por isso que os devemos simplificar. A sobrecarga dos ciclos advém em primeiro lugar do uso de variáveis de controlo e de instruções de teste e salto, por isso é aqui que deve incidir a optimização.

Note-se que se reduzirmos o número de saltos necessários num determinado ciclo estamos também a melhorar o desempenho do pipeline do processador, quando este exista.

### Variáveis de controlo

As variáveis de controlo dos ciclos deverão ser, sempre que possível, variáveis escalares passíveis de ser colocadas em registos do processador. A linguagem C permite ao programador seleccionar as variáveis que são apenas armazenadas em registos eliminando a sobrecarga das leituras e escritas na memória. Exemplo:

```
register int i;
for (i=0;i<N;i++){
....
}
```

### Desenrolar os ciclos (loop unrolling)

Esta técnica consiste em duplicar (ou n-tuplicar) as instruções num ciclo de forma a reduzir o número de iterações e a consequente sobrecarga associada. Por exemplo

```
for (i=0;i<6;i++)
        a[i]=a[i]*8;
```

pode ser substituído por

```
a[0]=a[0]*8;
a[1]=a[1]*8;
a[2]=a[2]*8;
a[3]=a[3]*8;
a[4]=a[4]*8;
a[5]=a[5]*8;
```

Claro que nem todos os ciclos poderão ser desenrolados desta forma, no entanto em muitos casos poderemos efectuar a seguinte modificação.

```
for (i=0; i<6;i++){
        a[i]=a[i]*8;
        i++;
        a[i]=a[i]*8;
}
```

Aqui melhorámos pois reduzimos a sobrecarga associada aos testes e saltos do ciclo para metade.

### Fusão de ciclos

Consiste em fundir dois ciclos num único ciclo, reduzindo a sobrecarga dos ciclos para metade. Por exemplo, o código seguinte

```
for (i=0;i<1000;i++)
        x[i]=y[i]*8;

for (i=0;i<1000;i++)
        z[i]=x[i]*y[i];
```

pode ser substituido por

```
for (i=0;i<1000;i++){
        x[i]=y[i]*8;
        x[i]=x[i]*y[i];
}
```

## 11.4 Optimização em sistemas com cache

Embora a velocidade dos processadores continue a aumentar, este aumento de velocidade não tem sido acompanhado pelas memórias. Para reduzir

esta discrepância, a arquitectura dos computadores tem sofrido modificações através da introdução de memórias cache. Estas têm como objectivo o de fornecer rapidamente ao processador os dados ou as porções de código mais utilizadas, em vez de ser necessário ir buscá-los à memória principal o que obriga a vários ciclos de espera por parte do processador. A optimização fornecida pelas caches, baseia-se no principio de que um programa em execução exibe sempre um certo nível de localidade temporal e espacial. Estes princípios consistem no seguinte:

- localidade espacial - se um elemento de memória foi acedido num determinado instante, existe uma tendência para nos instantes seguintes, se efectuarem acessos a elementos de memória vizinhos.
- localidade espacial - se um elemento de memória foi acedido num determinado instante, existe uma tendência para nos instantes seguintes, este voltar a ser acedido.

Podemos então calcular os tempos médios de acesso à memória sabendo o tempo que demora um acesso a memória principal, se a informação pretendida já se encontrar na cache ($T_{ca}$), a penalização sofrida se este não se encontrar na cache ($T_{pen}$) e a razão a que os acessos à memória não são acelerados pela cache ($r_f$), através de

$$T_{ma} = T_{ca} + r_f \times T_{pen}.$$

Na expressão anterior, o único valor que não é fixo é $r_f$ e é por isso o que pode ser reduzido através da optimização do código. Vejamos então algumas técnicas.

## Juntar arrays

Este método consiste em combinar dois arrays de forma a aumentar a localidade espacial. O exemplo seguinte

```
int val[SIZE];
int key[SIZE];
for (i=0; i<SIZE; i++){
        val[i]=i;
        key[i]=i-1;
}
```

pode ser modificado da seguinte forma

```
struct combo{
        int val;
        int key;
};
```

```
struct combo tab[SIZE];
for (i=0; i<SIZE; i++){
     tab[i].val=i;
     tab[i].key=i-1;
}
```

## Troca entre ciclos

Como se poderá ver no exemplo seguinte, a troca dos ciclos faz com que os elementos da matriz X sejam acedidos ao longo das linhas e não ao longo das colunas como acontecia antes da modificação. Resultado: cria localidade espacial. Assim o código

```
for (j=0;j<100;j++)
    for (i=0;i<5000;i++)
        x[i][j]=2*x[i][j];
```

passará a ser

```
for (i=0;i<5000;i++)
        for (j=0;j<100;j++)
                x[i][j]=2*x[i][j];
```

## Criação de blocos

Por vezes quando efectuamos operações sobre duas matrizes onde uma é acedida por colunas e a outra por linhas, a troca dos ciclos referida acima não resolve o problema. Neste caso a solução consiste em efectuar as operações em pequenos blocos das matrizes de forma a que estes blocos "caibam"na cache do processador criando assim localidade temporal e espacial.

Vejamos o caso da multiplicação de 2 matrizes $N \times N$ .

```
for (j=0;j<N;j++)
    for (i=0;i<N;i++){
        x[i][j]=0;
        for (k=0;k<N;k++)
                x[i][j] += y[i][k]*z[k][j];
    }
```

poderá ser transformado em

```
for (jj=0;jj<N;jj+=B)
        for (ii=0;ii<N;ii+=B)
                for (i=0;i<N;i++)
                        for (j=jj;j<min(jj+B-1,N);j++){
                                soma=0;
                                for (k=kk;k<min(kk+B-1,N);k++)
                                        soma+=y[i][k]*z[k][j];
                                x[i][j]+=soma;
                        }
```

De facto aumentámos a sobrecarga em termos de ciclos e instruções, mas a melhoria no desempenho da cache irá superar isso.

### Outras optimizações possíveis

1. Juntar os procedimentos ou funções mais utilizadas de forma a aumentar a localidade espacial durante a referência às mesmas (em sistemas paginados ou com cache)..

2. Armazenar elementos dos dados que são utilizados em conjunto de forma a aumentar a localidade das referências (em sistemas paginados ou com cache).

3. Armazenar procedimentos e funções em sequência de forma a que tanto as funções chamantes como as chamadas sejam carregadas simultaneamente (em sistemas paginados ou com cache).

## 11.5 Nem sempre é evidente quais as partes do código a optimizar

Quando escrevemos um programa nem sempre temos a noção correcta de quais são as partes do código que vão consumir mais tempo. A estruturação em funções e o uso de ferramentas de análise de perfis de execução permitem-nos obter essa informação.

Este trabalho centra-se no uso da ferramenta GPROF da GNU.

## 11.6 Exercícios

### Antes da aula

**Exercício 11.1**
Leia atentamente o artigo "gprof - a call graph execution profiler"[1] e faça um resumo do mesmo.

**Exercício 11.2**
Analise o código que se segue e explique o que faz.

```
/* Example program for time profiling. */


#include <stdio.h>
#include <stdlib.h>
```

```
struct AList {
    int             first;
    struct AList *  rest;
};
typedef  struct AList *  List;

List  data;

List make_empty()
{
    return NULL;
}

List make_nonempty(int first, List rest)
{
    List  list = (List) malloc(sizeof(struct AList));
    if (list == NULL) {
        fprintf(stderr,"Couldn't allocate cons-cell.\n");
        exit(1);
    }

    list->first = first;
    list->rest  = rest;

    return list;
}

/* Returning an int, instead of bool, so that we can use the
 * non-C99 compliant cc. */
int is_empty(List list)
{
    return (list == NULL);
}

List make_list_helper(unsigned int size, List accum)
{
    if (size == 1)
        return make_nonempty(size,accum);
    else
        return make_list_helper(size-1,make_nonempty(size,accum));
}

List make_list(unsigned int size)
{
    return make_list_helper(size,make_empty());
}

List append_lists(List list1, List list2)
{
    List  return_list;

    if (is_empty(list1))

        return list2;
    else {
        return_list = make_nonempty(list1->first,
                                    append_lists(list1->rest,list2));
        free(list1);
        return return_list;
    }
}
```

```
unsigned int get_num()
{
    unsigned int  n;
    scanf("%u",&n);
    return n;
}

/* Do interesting stuff to get some data. */
void get_data()
{
    unsigned int  size;

    data = make_empty();

    printf("To create some data, enter a sequence of positive numbers.\n");
    printf("Terminate the list with a zero.\n");
    printf("Use at least moderately large numbers to create enough data for good profiling.\n");
    while ((size=get_num()) != 0)
        data = append_lists(data,make_list(size));
}

List insert_list(int n, List list)
{
    List  return_list;

    if (is_empty(list))
        return make_nonempty(n,make_empty());
    else if (n <= list->first)
        return make_nonempty(n,list);
    else {
        return_list = make_nonempty(list->first,insert_list(n,list->rest));
        free(list);
        return return_list;
    }
}

List sort_list(List list)
{
    List  return_list;

    if (is_empty(list))
        return list;
    else {
        return_list = insert_list(list->first,sort_list(list->rest));
        free(list);
        return return_list;
    }
}

void sort_data()
{
    data = sort_list(data);
}

void print_list(List list)
{
    if (is_empty(list))
        printf("\n");
    else {
        printf("%d ",list->first);
        print_list(list->rest);
```

```
    }
}

/* Print out whatever data we have. */
void print_data()
{
    printf("The data:\n");
    print_list(data);
}


int main()
{
    get_data();
    print_data();

    /* Do something interesting on the data. */
    sort_data();

    print_data();

    return 0;
}
```

## Na aula

### Exercício 11.3

Crie o ficheiro de texto "dados " com o seguinte conteúdo:

```
10000
20000
30000
0
```

de seguida compile o código fornecido e listado acima e execute o programa resultante da seguinte forma

```
/usr/bin/time sample < dados
```

Registe os tempos mostrados e explique a que corresponde cada um deles.

### Exercício 11.4

Compile novamente o código com "gcc –pg sample.c –o sample" e corra-o novamente da mesma forma. A execução deverá ser mais lenta que a anterior, porquê?

### Exercício 11.5

Execute o comando "gprof sample > sample.profile" e analise o conteúdo do mesmo.

1. Explique o conteúdo desse ficheiro
2. Quais as partes do código que consomem mais tempo? Porquê?
3. Pode optimizar essas partes? Se sim qual o tempo de execução obtido depois da optimização?

**Exercício 11.6**
Veja no manual do GCC o que fazem as opções de optimização –O e –O2 explicando que tipo de optimizações são efectuadas em cada uma delas?

**Exercício 11.7**
Analise os tempos de execução do código inicial e optimizado usando essas optimizações.

# Multi-threading 12

Num sistema multi-processo temos normalmente os processos que correspondem a instâncias dos programas em execução e onde cada um destes tem um espaço de endereçamento completamente separado dos restantes. Isso leva a que durante a comutação de processos todos os registos sejam salvos numa estrutura apropriada para o processo que está a sair e sejam carregados todos os registos da estrutura equivalente para o processo que entra. Além disso entre dois processos não há qualquer tipo de localidade nos acessos a memória exceptuando eventuais zonas partilhadas.

Para permitir a redução da sobrecarga na troca de processos foi introduzido o conceito de "thread"ou "fio de execução"que referiremos doravante por fio. Um fio pode ser considerado como um processo leve. Na realidade é necessário um processo para albergar um ou mais fios de execução, mas a troca entre estes é mais simples e rápida do que a troca entre processos. Uma segunda vantagem que existe no uso de fios é que a comunicação entre estas é directa uma vez que partilham o mesmo espaço de memória, não necessitando por isso de envolver o sistema operativo nas comunicações ou no seu estabelecimento.

Uma das formas de criar programas com vários fios é usando a biblioteca "pthreads". Para isso deve-se incluir no inicio a linha

```
#include <pthread.h>
```

e para compilar deve-se usar uma linha como a que se segue

```
gcc -Wall -D_REENTRANT prog.c -lpthread -o program
```

Podemos agora usar as seguintes funções

```
int pthread_create(pthread_t * id, pthread_attr_t * atr,
voi *(*mythread)(void *), void * arg);

void pthead_exit(void * retval);
```

```
int pthread_join(pthread_t th, void **thread_return);

pthread_t  thread_self(void);
```

## 12.1 Exemplo

O seguinte exemplo cria um programa com 6 fios de execução.

```
#include <pthread.h>
#include <stdio.h>

void * helloworld(void * arg){
        int i;
        for (i=0;i<20;i++){
                printf("Hello world\n");
                usleep(13000);
        }
}

void * bonjour(void * arg){
        int i;
        for (i=0;i<20;i++){
                printf("Bonjour monde\n");
                usleep(20000);
        }
}

void * bomdia(void * arg){
        int i;
        for (i=0;i<20;i++){
                printf("Bom dia mundo\n");
                usleep(10000);
        }
}

void * qqcoisa(void * arg){
        int i;
        for (i=0;i<20;i++){
                printf("%s \n",(char *) arg);
                usleep(10000);
        }
}

main(){
        pthread_t t1, t2, t3,t4,t5,t6;
        printf("Arranque\n");
        pthread_create(&t1,NULL, helloworld,NULL);
        pthread_create(&t2,NULL, bonjour,NULL);
        pthread_create(&t3,NULL, bomdia,NULL);

        pthread_create(&t4,NULL, qqcoisa,"Ola");
        pthread_create(&t5,NULL, qqcoisa,"Hello");
        pthread_create(&t6,NULL, qqcoisa,"Salut");

        printf("Espera fim\n");
        pthread_join(t1,NULL);
        pthread_join(t2,NULL);
        pthread_join(t3,NULL);
```

```
        pthread_join(t4,NULL);
        pthread_join(t5,NULL);
        pthread_join(t6,NULL);
}
```

Note-se que são criados 3 fios a partir da função "qqcoisa", mas cada um deles com um argumento de entrada diferente. É também importante usar "pthread_join", caso contrário programa o processo iria terminar a execução antes de todos os fios terem terminado.

## 12.2 Fora da aula

**Exercício 12.1**
Parta do código fornecido para geração de imagens de fractais. Compile-o e execute-o. Verifique como os tempos de execução se podem tornar extremamente longos quando se geram imagens de resolução considerável. Para isso execute

```
make
./fractal 10000 10000
```

**Exercício 12.2**
Crie uma versão do programa que use 4 fios (threads), modificando para isso a função "void Generate(int)"sem mexer nas funções julia e mandelbrot.

**Exercício 12.3**
Repita a questão anterior, criando uma versão que crie tantos fios quantos os necessários para que cada um deles gere no máximo blocos da imagem de 512 por 512.

## 12.3 Na aula

Explique ao professor todos os detalhes e escolhas que fez para a implementação atingida.

# Programação paralela com OpenMP

# 13

O OpenMP [2] pode ser visto como uma extensão da linguagem C/C++ (e Fortran) que permite tirar partido das capacidades de processamento dos actuais sistemas "multi-processador"(multi-core × multi-chip). Através do seu uso é possível escrever código que usa todos os processadores disponíveis, desde que esse código seja "paralelizável".

Lançado em 1997, o OpenMP, que vai já na versão 3.X, permite escrever programas paralelos de uma forma simples e padronizada. Uma boa parte dos compiladores actuais incluem suporte para OpenMP, de entre eles o conjunto da GNU (gcc, g++ e gfortran), os compiladores da Intel (icc, icpc, ifort) e os compiladores do Portland Group (pgc, pgCC, pgf77) e estão disponíveis para sistemas operativos como linux, Windows e Mac OS X.

Esta ficha serve de introdução ao OpenMP permitindo explorar algumas das suas capacidades. Os exemplos serão dados em C ou C++. A melhor forma de aprender é experimentando os exemplos apresentados e introduzir modificações por sua própria iniciativa.

## 13.1 O primeiro programa

Antes de qualquer explicação, sugiro copiar o código que se segue para um editor para criar um simples programa em OpenMP-C.

```
#include <stdio.h>

int main(int argc, char **argv)
{
    #pragma omp parallel
    {
        printf("Hello OpenMP!\n");
```

```
    }

    return 0;
}
```

De seguida poderá compilá-lo com o gcc (de versão 4.2 ou superior), o icc, ou o pgc, da forma seguinte

- **gcc :** gcc -fopenmp hellomp.c -o hellomp
- **icc :** icc -openmp hellomp.c -o hellomp -cxxlib-icc
- **pgc :** pgc -mp hellomp.c -o hellomp

compilando e executando na máquina lsrc obtemos o seguinte.

```
pm@lsrc:~/OpenMP$ gcc -fopenmp hellomp.c -o hellomp
pm@lsrc:~/OpenMP$ ./hellomp
Hello OpenMP!
Hello OpenMP!
Hello OpenMP!
Hello OpenMP!
pm@lsrc:~/OpenMP$
```

Surpreendentemente a mensagem surgiu 4 vezes. Porque razão?

Um programa normal executa uma linha de código de cada vez, que num programa em C começa com a função main. Neste caso dizemos que temos um fio de execução, o fio de execução principal (main). Como sabemos todos os programas têm um fio de execução principal e na maior parte dos programas este fio é o único que existe, por isso o programa só pode fazer uma coisa de cada vez.

Na realidade no código acima, definimos que o bloco de código que contém a função printf pertence a uma equipa de fios (team of threads) definida pela secção "parallel"do OpenMP. Se mais nada for dito, o fio principal será dividido numa equipa de tantos fios quantos os processadores da máquina. Podemos no entanto especificar o número de fios pretendido definindo uma variável de ambiente antes de executar o programa, da seguinte forma:

```
export OMP_NUM_THREADS=8
```

## 13.2 Directivas (Pragmas)

Uma secção paralela é definida em OpenMP usando directivas do compilador. Estas directivas são instruções para o compilador indicando-lhe como criar a equipa de fios, como associar os fios a tarefas, entre outras. Estas directivas são apenas tomadas em conta se o programa for compilado

com suporte OpenMP. Caso esse suporte não exista, essas directivas são ignoradas.

Existem várias directivas para o OpenMP. Aqui iremos explorar apenas um subconjunto destas composto por:

- **parallel :** usada para cria um bloco de código paralelo que será executado por uma equipa de fios.
- **section :** usada para especificar diferentes secções de código a ser executadas em paralelo por diferentes fios.
- **for :** usada para expecificar ciclos em que diferentes iterações dos mesmos serã executadas por diferentes fios.
- **critical :** usada para definir um bloco que só pode ser executado por um fio de cada vez.
- **reduction :** usada para combinar (reduzir) os resultados de vários cálculos locais em um único resultado.

Estas directivas fazem na verdade com que o compilador introduza o código necessário para activar o paralelismo através de OpenMP, mas também podemos usar directamente funções deste suporte como no código que se segue.

```
#include <stdio.h>

#ifdef _OPENMP
    #include <omp.h>
#else
    #define omp_get_num_threads() 0
    #define omp_get_thread_num() 0
#endif

int main(int argc, char **argv)
{
    int nthreads, thread_id;

    printf("I am the main thread.\n");

    #pragma omp parallel private(nthreads, thread_id)
    {
        nthreads = omp_get_num_threads();
        thread_id = omp_get_thread_num();

        printf("Hello. I am thread %d out of a team of %d\n",
                thread_id, nthreads);
    }

    printf("Here I am, back to the main thread.\n");

    return 0;
}
```

A novidade aqui em termos de directivas é a definição de que as variaveis nthreads e thread_id são privadas, ou seja cada fio tem uma variável com este nome em vez de ser global e partilhada.

## 13.3 Directiva *section*

A directiva "section"define um bloco de código que é associado a um único fio dentro de uma equipa. O exemplo seguinte ilustra o seu uso.

```
#include <stdio.h>
#include <unistd.h>

#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_thread_num() 0
#endif

void times_table(int n)
{
    int i, i_times_n, thread_id;

    thread_id = omp_get_thread_num();

    for (i=1; i<=n; ++i)
    {
        i_times_n = i * n;
        printf("Thread %d says %d times %d equals %d.\n",
               thread_id, i, n, i_times_n );

        sleep(1);
    }
}

void countdown()
{
    int i, thread_id;

    thread_id = omp_get_thread_num();

    for (i=10; i>=1; --i)
    {
        printf("Thread %d says %d...\n", thread_id, i);
        sleep(1);
    }

    printf("Thread %d says \"Lift off!\"\n", thread_id);
}

void long_loop()
{
    int i, thread_id;
    double sum = 0;

    thread_id = omp_get_thread_num();

    for (i=1; i<=10; ++i)
```

```
    {
        sum += (i*i);
        sleep(1);
    }

    printf("Thread %d says the sum of the long loop is %f\n",
            thread_id, sum);
}

int main(int argc, char **argv)
{
    printf("This is the main thread.\n");

    #pragma omp parallel
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                times_table(12);
            }
            #pragma omp section
            {
                countdown();
            }
            #pragma omp section
            {
                long_loop();
            }
        }
    }

    printf("Back to the main thread. Goodbye!\n");

    return 0;
}
```

Neste exemplo temos 3 fios um para a chamada da função times_table(), outro para a chamada da função countdown() e outro para long_loop().

## 13.4 Directiva *for*

O OpenMP permite ainda paralelizar a execução de ciclos. Se por exemplo tivermos um ciclo que execute 1000 iterações, essas iterações poderão ser divididas por vários fios. Assim com dois fios teriamos cada um a executar 500 iterações, quatro fios executariam 250 iterações cada, etc.

É no entento importante notar que isto apenas é possível se os ciclos forem independentes, ou seja não dependam dos resultados uns dos outros e assim possam ser executados em qualquer ordem.

```
#include <stdio.h>

#ifdef _OPENMP
    #include <omp.h>
#else
```

```
    #define omp_get_thread_num() 0
#endif

int main(int argc, char **argv)
{
    int i, thread_id, nloops;

    #pragma omp parallel private(thread_id, nloops)
    {
        nloops = 0;
        thread_id = omp_get_thread_num();

        #pragma omp for
        for (i=0; i<1000; ++i)
        {
                        if (nloops==0)
                                printf("Thread %d started with i=%d\n",thread_id,i);

                ++nloops;
        }

        thread_id = omp_get_thread_num();

        printf("Thread %d performed %d iterations of the loop.\n",
                thread_id, nloops );
    }

    return 0;
}
```

## 13.5 Directiva *critical*

A directiva critical permite definir um bloco de código que deverá ser executado de forma exclusiva, ou seja só poderá ser executado por um fio de cada vez. Isto é importante no acesso a recursos partilhados como por exemplo a actualização de variáveis globais. O código que se segue é um exemplo disso, onde a variavel global_nloops é actualizada pelos diferentes fios, mas dependendo das condições de execução poderá ter um resultado final que não é o desejado devido a não ser garantidas as condições para que as actualizações sejam atómicas.

```
#include <stdio.h>

#ifdef _OPENMP
    #include <omp.h>
#else
    #define omp_get_thread_num() 0
#endif

int main(int argc, char **argv)
{
    int i, thread_id;
    int global_nloops, private_nloops;
```

```
    global_nloops = 0;

    #pragma omp parallel private(private_nloops, thread_id)
    {
        private_nloops = 0;

        thread_id = omp_get_thread_num();

        #pragma omp for
        for (i=0; i<100000; ++i)
        {
            ++private_nloops;
        }

        printf("Thread %d adding its iterations (%d) to the sum (%d)...\n",
               thread_id, private_nloops, global_nloops);

        global_nloops += private_nloops;

        printf("...total nloops now equals %d.\n", global_nloops);
    }

    printf("The total number of loop iterations is %d\n",
           global_nloops);

    return 0;
}
```

Resolvendo com a directiva critical, obtemos o seguinte código, onde o acesso a essa mesma variável só é feito por um fio de cada vez.

```
#include <stdio.h>

#ifdef _OPENMP
    #include <omp.h>
#else
    #define omp_get_thread_num() 0
#endif

int main(int argc, char **argv)
{
    int i, thread_id;
    int global_nloops, private_nloops;

    global_nloops = 0;

    #pragma omp parallel private(private_nloops, thread_id)
    {
        private_nloops = 0;

        thread_id = omp_get_thread_num();

        #pragma omp for
        for (i=0; i<100000; ++i)
        {
            ++private_nloops;
        }

        #pragma omp critical
        {
            printf("Thread %d adding its iterations (%d) to the sum (%d)...\n",
```

```
                thread_id, private_nloops, global_nloops);

            global_nloops += private_nloops;

            printf("...total nloops now equals %d.\n", global_nloops);
        }
    }

    printf("The total number of loop iterations is %d\n",
           global_nloops);

    return 0;
}
```

## 13.6 Directiva *reduction*

A redução é o processo de combinar os resultados de vários "sub-cálculos"num único resultado. A técnica "map-reduce"é de facto uma forma muito eficiente de, através de programação paralela, efectuar cálculos complexos dividindo-os no cálculo paralelo de vários resultados parciais que no final são combinados (reduzidos) a um único resultado.

O OpenMP fornece uma forma de efectuar esta operação (muitas vezes complexa), de forma eficiente através da directiva reduce

```
reduce( operator : lista de variaveis)
```

onde o operador poder ser qualquer operador binário (ex: +,-,*), a lista de variáveis pode ser uma única variável ou uma lista de variáveis usadas para a redução. Por exemplo

```
reduction( + : sum )
```

neste caso o compilador vai combinar na variavel sum a soma dos resultados parciais de cada um dos fios. O exemplo seguinte ilustra o seu uso.

```
#include <stdio.h>

int main(int argc, char **argv)
{
    int i;
    int private_nloops, nloops;

    nloops = 0;

    #pragma omp parallel private(private_nloops) \
                         reduction(+ : nloops)
    {

        #pragma omp for
        for (i=0; i<100000; ++i)
        {
            ++nloops;
        }
```

```
            }

    printf("The total number of loop iterations is %d\n",
            nloops);

    return 0;
}
```

## 13.7 Antes da aula

1. Teste todos os programas exemplo fornecidos acima e analise o seu comportamento.

2. Visite o sitio e veja como tratar os problemas relacionados com a consistência da memória.

3. Prepare a resolução dos dois exercícios seguintes. Pode testar na máquina lsrc.deec.uc.pt (lsrc.labs.deec.uc.pt).

## 13.8 Na aula

Estes dois exercícios deverão ser realizados completamente na aula.

**Exercício 13.1**
Em qualquer tabela de derivadas encontramos que

$$\frac{d\,(tan^{-1}(x))}{dx} = \frac{1}{1+x^2}.$$

Uma vez que $tan(0) = 0$ e $tan(\pi/4) = 1$, se calcularmos a primitiva

$$\int_0^1 \frac{1}{1+x^2} = tan^{-1}(1) - tan^{-1}(0) = \frac{\pi}{4} - 0 \tag{13.1}$$

Assim podemos obter o valor de $\pi/4$ através do integral anterior, que por sua vez pode ser aproximado pela regra do trapézio. Esta aproximação consiste em dividir o intervalo de integração em subintervalos, sendo a função aproximada por segmentos de recta que unem os valores da função nas extremidades desses subintervalos. Finalmente o integral da função é aproximado pela soma das áreas dos trapézios obtidos. Por consequência a aproximação é tanto melhor quanto maior for o número de subdivisões do intervalo de integração.

$$\int_a^b f(x) = \sum_{i=0}^{k-1} (x_{i+1} - x_i) \frac{(f(x_i) + f(x_{i+1}))}{2},$$

em que se dividiu em $k$ intervalos, logo $d = (b-a)/k$ e $x_i = a + i \times d$. Simplificando obtém-se

$$\int_a^b f(x) = \frac{d}{2} \sum_{i=0}^{k-1} f(x_i) + f(x_{i+1}). \tag{13.2}$$

1. Escreva um programa que faça obtenha uma aproximação de do valor de $\pi$ aproximando o integral 13.1 através da equação 13.2. Deve pedir ao utilizador o número de intervalos a utilizar na aproximação e ao apresentar o valor calculado mostrar também o erro em relação ao valor de referência 3.14159265358979323846264 3.

2. Execute o programa anterior tomando nota dos tempos de execução para vários valores de $k$. Os tempos de execução podem ser obtidos pelo comando **time ./programa_pi**.

3. Modifique o programa anterior para, usando OpenMP. tirar partido dos vários núcleos que possam existir na máquina onde executar. O programa deve indicar quantos fios de execução (*threads*) são usados.

4. Tome nota dos tempos de execução para os mesmos valores de k usados na alinea 2 e discuta os resultados obtidos.

**Exercício 13.2**

Neste exercício pretende-se simular a distribuição de temperaturas num lago e sua evolução. Considerando que todo o lago está inicialmente a uma temperatura de 16 graus Celsius, vamos introduzir um ponto quente numa extremidade do lago e ver como a temperatura se propaga ao longo da superfície do lago.

Para fazer a simulação vamos dividir área do lago em células quadradas (cuja dimensão não interessa para este trabalho). A temperatura de uma célula numa época $k$, é calculada através da média dos valores de temperatura desta e das suas 4 vizinhas (não considerando as dos cantos) na época $k-1$.

1. Escreva um programa que receba 5 parâmetros: número de linhas, número de colunas, linha calor, coluna calor, numero de épocas. O programa deve gerar 1 ficheiro imagem para cada época, onde os valores dos píxeis correspondem aos valores de temperatura das células

correspondentes. Os nomes dos ficheiros deverão permitir identificar a época a que correspondem, ou seja o ficheiro lake004.pgm corresponde à quarta época simulada.

**Nota:**Para gerar o ficheiro use a seguinte função:

```
 /* This function saves a rectangular array of integer
values  as an image of grey levels. Note that only
values between 0 and 255 are valid. */
void savepgm(int * img,int rx, int ry, char * fname){
        FILE * fp;
        int color,i,j;
        fp=fopen(fname,"w");
        /* header for PPM output */
        fprintf(fp,"P5\n# CREATOR: AC Course, DEEC-UC\n");
        fprintf(fp,"%d %d\n255\n",rx,ry);

        for (i=0;i<ry;i++){
                for (j=0;j<rx;j++){
                        color=img[i*rx+j];
                        if (color>255)
                                color=255;
                        else if (color <0)
                                color=0;
                        fputc((char)(color),fp);
                }
        }
        fclose(fp);
}
```

2. Usando o OpenMP modifique o programa anterior para que o cálculo das temperaturas seja distribuído pelos núcleos disponíveis.

3. Modifique o programa anterior para que a escrita em ficheiro fique apenas da responsabilidade de um fio (thread) enquanto as restantes estão a calcular a época seguinte.

4. Analise os tempos de execução para cada um dos casos.

# Trabalho com OpenMP 14

Os problemas candidatos ao uso de processamento paralelo são sempre aqueles que apresentam um peso computacional bastante elevado. De entre esses podemos referir a simulação meteorológica e sísmica. O que se propõe aqui como trabalho é a simulação de uma galáxia.

Assim vamos considerar um número elevado (muitos milhares) de corpos que se atraem segundo a equação bem conhecida

$$\mathbf{f}_{i,j} = \frac{G.m_i.m_j}{||\mathbf{p}_j - \mathbf{p}_i||^2} \frac{\mathbf{p}_j - \mathbf{p}_i}{||\mathbf{p}_j - \mathbf{p}_i||}$$

onde a negrito temos os vectores posição de cada um dos corpos e força, $G$ é uma constante gravitacional e $m_i$ e $m_j$ as massas dos corpos. Note-se que a segunda fração representa o versor da força calculada.

O que se pretende obter é a evolução das posições de cada um dos corpos sabendo que cada um sofre forças de atração de todos os outros. Ou seja a força que o corpo i sofre é dada por

$$\mathbf{f}_i = \sum_{j=0, j!=i}^{N} \mathbf{f}_{i,j}.$$

Por razões de simplicidade podem-se ignorar as colisões, podendo assim um corpo atravessar outro (ou absorvê-lo opcionalmente). Os calculos devem ser discretizados no tempo, ou seja a simulação ser feita para pequenos incrementos temporais. Em cada passo, o programa calcula as forças gravitacionais exercidas em cada corpo por todos os outros, actualizando a sua posição e velocidade.

A força gravitacional é então usada para calcular as novas posições e velocidades dos corpos da seguinte forma: Para cada corpo, a força $f$ é decomposta nas componentes $f_{x_i}$, $f_{y_i}$ e $f_{z_i}$. As componentes correspondentes a todos os corpos são somadas, o que dá o vector força gravitacional

aplicado ao corpo. A partir destes valores são calculados os valores das componentes da aceleração $(x_x.a_y, a_z)$ e velocidade $(v_x, v_y, v_z)$. Para o caso das componentes segundo o eixo dos $x$ temos

$$a_x = f_x/m,$$

$$v_x(k) = v_x(k-1) + a_x \times \Delta t$$

e

$$p_x(k) = p_x(k-1) + v_x \times \Delta t$$

.

O programa deve para cada passo escrever num ficheiro as coordenadas de cada um dos corpos (x y z de cada um por linha), de forma a poder ser lido pelo programa gnuplot. Use e modifique o script disponibilizado no Inforestudante para efeitos de visualização

## 14.1 Devem entregar

1. o programa comentado (sem comentários vale 0)
2. a descrição da estratégia de paralelização
3. Análise de comparação de resultados para 1, 2 ou mais threads.
4. Video opcional com animação dos resultados de uma simulação.

# Programação distribuída com MPI

15

Quando necessitamos de muita potência de cálculo para resolver um qualquer problema de grandes dimensões, uma possível forma de resolver este problema é usando programação paralela distribuída. A principal diferença em relação à programação com "threads"tradicional é que a memória deixa de ser centralizada e por isso não pode ser usada para troca de dados entre essas threads. Assim temos uma nova forma de programar baseada em nós rápidos de computação que comunicam entre si através de ligações de rede relativamente lentas.

A norma MPI (Message Passing Interface)[3, 4, 5, 6] vem definir a forma como se pode obter essa programação paralela distribuída e como os nós comunicam entre si.

As seis funções de MPI que são absolutamente necessárias são descritas a seguir.

**int MPI_Init(int *argc, char **argv)** Esta função deve ser colocada logo a seguir à declaração das variáveis e antes de chamar qualquer outra função MPI.

**int MPI_Finalize(void)** A colocar no final.

**int MPI_Comm_size(MPI_Comm comm, int *size)** Vai descobrir quantos processos distribuídos vamos ter.

**int MPI_Comm_rank(MPI_Comm comm, int *rank)** Devolve o identificador do processo corrente.

**int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)** Envia a mensagem de tamanho

count bytes e tipo definido para o processo de id *dest* do comunicador com.

Exemplo: MPI_Send(&x, 1, MPI_DOUBLE, manager, me, MPI_COMM_WORLD)

**int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)** Recebe uma mensagem de um processo.

Exemplo: MPI_Recv(&x, 1, MPI_DOUBLE, source, source, MPI_COMM_WORLD, &status)

Em relação à troca de mensagens é importante ter em atenção o seguinte:

- Ambas as funções padrão send e receive são bloqueantes.
- MPI_Recv apenas retorna quando o buffer de receção contiver a mensagem requerida.
- MPI_Send pode ou não bloquear até que a mensagem seja recebida (normalmente bloqueia)
- Deve-se ter cuidado com a possibilidade de deadlocks.

## 15.1 Deadlocks em MPI

Os deadlocks podem ser criados por uso não cuidado com a forma como são escalonadas as trocas de mensagens entre processos.

No exemplo que se segue haverá sempre a ocorrência de deadlocks pois todos os processos ficarão bloqueados à espera de receber uma mensagem sem nenhum poder avançar para a enviar.

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
    int me, np, q, sendto;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    if (np%2==1) return 0;
    if (me%2==1) {sendto = me-1;}
    else {sendto = me+1;}
    MPI_Recv(&q, 1, MPI_INT, sendto, sendto, MPI_COMM_WORLD, &status);
    MPI_Send(&me, 1, MPI_INT, sendto, me, MPI_COMM_WORLD);
    printf("Sent %d to proc %d, received %d from proc %d\n", me,
            sendto, q, sendto);
    MPI_Finalize();
    return 0;
}
```

No exemplo seguinte, poderão ocorrer deadlocks nos casos em que o envio se torna bloqueante até as mensagens sejam recebidas.

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
    int me, np, q, sendto;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    if (np%2==1) return 0;
    if (me%2==1) {sendto = me-1;}
    else {sendto = me+1;}
    MPI_Send(&me, 1, MPI_INT, sendto, me, MPI_COMM_WORLD);
    MPI_Recv(&q, 1, MPI_INT, sendto, sendto, MPI_COMM_WORLD, &status);
    printf("Sent␣%d␣to␣proc␣%d,␣received␣%d␣from␣proc␣%d\n", me,
            sendto, q, sendto);
    MPI_Finalize();
    return 0;
}
```

No exemplo seguinte não haverá deadlocks pois estamos a garantir que uns processos começam por enviar as mensagens a outros que começam por ficar à espera de as receber.

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
    int me, np, q, sendto;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    if (np%2==1) return 0;
    if (me%2==1) {sendto = me-1;}
    else {sendto = me+1;}
    if (me%2 == 0) {
        MPI_Send(&me, 1, MPI_INT, sendto, me, MPI_COMM_WORLD);
        MPI_Recv(&q, 1, MPI_INT, sendto, sendto, MPI_COMM_WORLD, &status);
        } else {
        MPI_Recv(&q, 1, MPI_INT, sendto, sendto, MPI_COMM_WORLD, &status);
        MPI_Send(&me, 1, MPI_INT, sendto, me, MPI_COMM_WORLD);
    }
    printf("Sent␣%d␣to␣proc␣%d,␣received␣%d␣from␣proc␣%d\n", me, sendto, q,                sendto);
    MPI_Finalize();
    return 0;
}
```

Os programas podem bloquear se a primitiva send não tiver uma receive correspondente ou vice-versa. Outro caso que pode acontecer é quando os pares send/receive não têm concordam na origem/destino ou na "tag".

## 15.2 Primeiro teste

Existem diversas implementações de MPI e o que vamos usar nas aulas é o MPICH2 que está instalado nas máquinas do laboratório 10.10.128.1 a 10.10.128.5 (lado esquerdo) e 10.10.128.10 a 10.10.128.15 (lado direito). Para acesso do exterior ou da rede "eduroam"pode fazer ssh para a máquina lsrc.deec.uc.pt com

```
$ ssh aXXXXXX@lsrc.deec.uc.pt
```

As máquinas partilham o mesmo diretório de utilizador pelo que temos parte das condições necessárias para utilizar o MPI. Agora é necessário permitir à conta do utilizador na segunda máquina a partir da primeira sem requerer password. Para isso execute os seguintes passos:

```
$ ssh-keygen -t rsa
Generating public/private rsa key pair....
$ cd .ssh
$ cat id_rsa.pub >> authorized_keys
```

Teste o acesso com

```
$ ssh 10.10.128.4 date
```

se não pedir password e mostrar a hora e data é porque foi bem sucedido.

A primeira vez que fazem login nessas máquinas é preciso confirmar a autenticidade das mesmas e por isso deverão fazer ssh para cada uma das máquinas do cluster e responder afirmativamente a

```
 $ ssh 10.10.128.1
The authenticity of host '10.10.128.1 (10.10.128.1)' can't be established.
RSA key fingerprint is cd:e3:92:87:f0:6d:04:34:57:31:e1:3e:43:37:b5:79.
Are you sure you want to continue connecting (yes/no)?
```

Se não fizermos isto uma vez, os programas em MPI não poderão ser executados.

Vamos agora criar o ficheiro de configuração do cluster de execução. Para isso basta criar um ficheiro com os endereços das máquinas e o número de "threads", que por defeito será 1, que queremos executar em cada uma delas. Vejamos o exemplo do ficheiro a criar para a fila do lado direito da sala, a que iremos chamar *cluster_right*.

```
10.10.128.1:4
10.10.128.2
10.10.128.3
10.10.128.4
10.10.128.5
```

Vamos agora criar o nosso "helloworld.c"com o seguinte código:

```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

Depois de gravado vamos compilá-lo e executá-lo com

```
$ mpicc helloworld.c -o mpihello
$ mpiexec -f cluster_right ./mpihello
```

## 15.3 Debugging

A depuração de programas paralelos e em particular distribuídos é difícil. No entanto apresenta-se aqui uma forma de o fazer em ambiente linux.

Comece por criar um ficheiro com os comandos que irá passar ao debugger, poderá colocar breakpoints, etc. Chamemos a esse ficheiro gdb.txt, que no exemplo seguinte apenas tem o comando para executar o programa

```
r
```

De seguida execute-o usando o seguinte comando

```
mpirun -f cluster_right xterm -e gdb -q -tui -x gdb.txt ./mpihello
```

Isto deverá abrir uma janela para cada processo criado onde se poderá ver o resultado ou controlar a execução de cada um, se tivermos introduzido um breakpoint, por exemplo, no ficheiro de comandos.

## 15.4 Na aula

**Exercício 15.1**

Escreva, teste e descreva o funcionamento do código que se segue.

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"     /* includes MPI library code specs */

#define MAXSIZE 100

int main(int argc, char* argv[])
```

```
{
  int  myRank;          /* rank (identity) of process      */
  int  numProc;         /* number of processors            */
  int  source;          /* rank of sender                  */
  int  dest;            /* rank of destination             */
  int  tag = 0;         /* tag to distinguish messages     */
  char mess[MAXSIZE];   /* message (other types possible) */
  int  count;           /* number of items in message      */
  MPI_Status status;    /* status of message received      */
  MPI_Init(&argc, &argv);          /* start MPI   */

  /* get number of processes  */
  MPI_Comm_size(MPI_COMM_WORLD, &numProc);

  /* get rank of this process */
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

  /*  code to send, receive and process messages */
  if (myRank != 0){/* all processes send to root        */

    /* create message */
    sprintf(mess, "Hello from %d", myRank);
    dest  = 0;                  /* destination is root  */
    count = strlen(mess) + 1;  /* include '\0' in message */

    MPI_Send(mess, count, MPI_CHAR,
             dest, tag, MPI_COMM_WORLD);
  }
  else{/* root (0) process receives and prints messages */
       /* from each processor in rank order               */
    for(source = 1; source < numProc; source++){

      MPI_Recv(mess, MAXSIZE, MPI_CHAR,
               source, tag, MPI_COMM_WORLD, &status);

      printf("%s\n", mess);
    }
  }
  MPI_Finalize();                /* shut down MPI */
}
```

## Exercício 15.2

Escreva, teste e descreva o funcionamento do código que se segue.

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"     /* includes MPI library code specs */
#include <math.h>

int main( int argc, char *argv[] ) {
    int n, myid, numprocs, i;
    double PI25DT = 3.14159265358979323846264 3;
    double mypi, pi, h, sum, x;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    while (1) {
        if (myid == 0) {
            printf("Enter the number of intervals: (0 quits) ");
            scanf("%d",&n);
```

```
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0)
            break;
        else {
            h   = 1.0 / (double) n;
            sum = 0.0;
            for (i = myid + 1; i <= n; i += numprocs) {
                x = h * ((double)i - 0.5);
                sum += (4.0 / (1.0 + x*x));
            }
            mypi = h * sum;
            MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                       MPI_COMM_WORLD);
            if (myid == 0)
                printf("pi is approximately %.16f, Error is %.16f\n",
                       pi, fabs(pi - PI25DT));
        }
    }
    MPI_Finalize();
    return 0;
}
```

## Exercício 15.3

Escreva, teste e descreva o funcionamento do código que se segue.

```
/* compute pi using Monte Carlo method */
#include <math.h>
#include "mpi.h"
//#include "mpe.h"
#define CHUNKSIZE       1000
#define INT_MAX 1000000000

/* message tags */
#define REQUEST  1
#define REPLY    2
int main( int argc, char *argv[] ) {
    int iter;
    int in, out, i, iters, max, ix, iy, ranks[1], done, temp;
    double x, y, Pi, error, epsilon;
    int numprocs, myid, server, totalin, totalout, workerid;
    int rands[CHUNKSIZE], request;
    MPI_Comm world, workers;
    MPI_Group world_group, worker_group;
    MPI_Status status;
   MPI_Init(&argc,&argv);
    world  = MPI_COMM_WORLD;
    MPI_Comm_size(world,&numprocs);
    MPI_Comm_rank(world,&myid);
    server = numprocs-1;        /* last proc is server */
    if (myid == 0)
        sscanf( argv[1], "%lf", &epsilon );
    MPI_Bcast( &epsilon, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD );
    MPI_Comm_group( world, &world_group );
    ranks[0] = server;
    MPI_Group_excl( world_group, 1, ranks, &worker_group );
    MPI_Comm_create( world, worker_group, &workers );
    MPI_Group_free(&worker_group);

if (myid == server) {   /* I am the rand server */
```

```
        do {
            MPI_Recv(&request, 1, MPI_INT, MPI_ANY_SOURCE, REQUEST,
                     world, &status);
            if (request) {
                for (i = 0; i < CHUNKSIZE; ) {
                        rands[i] = random();
                        if (rands[i] <= INT_MAX) i++;
                }
                MPI_Send(rands, CHUNKSIZE, MPI_INT,
                         status.MPI_SOURCE, REPLY, world);
            }
        }
        while( request>0 );
    }
else {                  /* I am a worker process */
        request = 1;
        done = in = out = 0;
        max  = INT_MAX;          /* max int, for normalization */
   MPI_Send( &request, 1, MPI_INT, server, REQUEST, world );
   MPI_Comm_rank( workers, &workerid );
        iter = 0;
        while (!done) {
            iter++;
            request = 1;
            MPI_Recv( rands, CHUNKSIZE, MPI_INT, server, REPLY,
                      world, &status );
            for (i=0; i<CHUNKSIZE-1; ) {
                x = (((double) rands[i++])/max) * 2 - 1;
                y = (((double) rands[i++])/max) * 2 - 1;
                if (x*x + y*y < 1.0)
                    in++;
                else
                    out++;
            }
            MPI_Allreduce(&in, &totalin, 1, MPI_INT, MPI_SUM,
                          workers);
            MPI_Allreduce(&out, &totalout, 1, MPI_INT, MPI_SUM,
                          workers);
            Pi = (4.0*totalin)/(totalin + totalout);
            error = fabs( Pi-3.14159265358979323846264 3);
            done = (error < epsilon || (totalin+totalout) > 1000000);
            request = (done) ? 0 : 1;
            if (myid == 0) {
                printf( "\rpi = %23.20f", Pi );
                MPI_Send( &request, 1, MPI_INT, server, REQUEST,
                          world );
            }
            else {
                if (request)
                    MPI_Send(&request, 1, MPI_INT, server, REQUEST,
                             world);
            }
        }
        MPI_Comm_free(&workers);
    }
 if (myid == 0) {
        printf( "\npoints: %d\nin: %d, out: %d, <ret> to exit\n",
                totalin+totalout, totalin, totalout );
        getchar();
    }
    MPI_Finalize();
}
```

**Exercício 15.4**
Implemente em MPI o problema da simulação da evolução da temperatura do lago. Para esta implementação deve testar com grelhas de 10000x10000 para 10000 épocas, gravando o ficheiro para as épocas múltiplas de 1000.

# Bibliografia

Poderá consultar as páginas para as funções MPI em
http://www.mcs.anl.gov/research/projects/mpi/www/www3/.

Os documentos sobre as várias versões do standard MPI podem ser encontradas em
http://www.mpi-forum.org/docs/ .

Uma introdução ao MPI pode ser encontrada em
https://computing.llnl.gov/tutorials/mpi/ ou http://mpitutorial.com/ .

# Exercícios com OpenMP e MPI 16

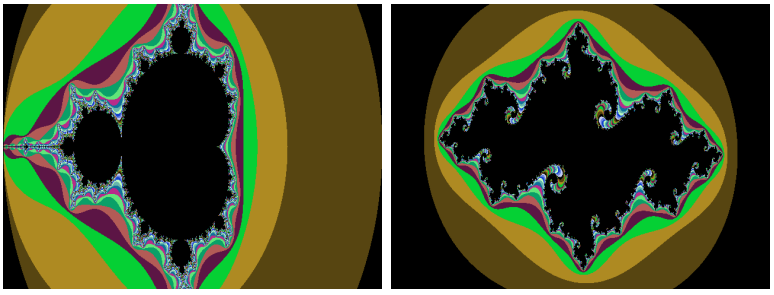


Figura 16.1: Left: Julia fractal; Right: Mandelbrot fractal

Nesta ficha é fornecido o código para gerar fartais e gravá-las sob a forma de ficheiros imagem do tipo PGM, como os da figura 16.1. Se pretender visualizar no windows esses ficheiros pode começar por convertê-los para o formato PNG ou BPM com o comando convert na máquina LSRC.

```
$ convert julia.pgm julia.png
```

## 16.1 Trabalho a realizar

### Exercício 16.1

Parta do código fornecido para geração de imagens de fractais. Compile-o e execute-o. Verifique como os tempos de execução se podem tornar extremamente longos quando se geram imagens de resolução considerável.Veja o exemplo seguinte.

```
$ make
$ ./fractal 10000 10000
```

Com o comando *time* pode contabilizar o tempo de execução do programa.

```
$ time ./fractal 10000 10000
```

**Exercício 16.2**

Crie uma versão modificada deste código que usando MPI distribua o calculo pelos nós disponíveis. Deve modificar apenas a função "void Generate(int)"sem mexer nas funções julia e mandelbrot.

**Exercício 16.3**

Crie uma versão modificada deste código que usando OpenMP distribua o calculo pelos processadores disponíveis. Deve modificar apenas a função "void Generate(int)"sem mexer nas funções julia e mandelbrot.

# Programação com OpenCL

# 17

Os processadores gráficos, também conhecidos como GPU, começaram a ser utilizados em aplicações de cálculo intensivo no final da década passada. Isto levou a que fabricantes e construtores de computadores passassem a olhar para esta tecnologia como uma nova oportunidade e surgiram linguagens de programação específicas para estes novos dispositivos computacionais, já que as linguagens tradicionais como o C ou o C++ não se prestam a esta utilização. Um dos exemplos de sucesso foi a criação pela Nvidia do CUDA. No entanto esta linguagem, tal como outras, era não podia ser usada em programas que pudessem ser executados em outro tipo de processadores. Foi aí que surgiu a OpenCL (Open Computing Language), com a qual podem ser criadas rotinas que poderão ser executadas em CPUs, e GPUs dos principais fabricantes. Trata-se de uma linguagem não proprietária, baseada numa norma pública. Uma vez obtido o suporte de execução o código será compilado e executado em processadores AMD Fusion, Intel Core, Nvidia Fermi, ou IBM Cell. A proposta de criação de uma interface de programação única veio da Apple e o seu poder fez com que os construtores não virassem costas à ideia.

## Programação paralela

O OpenCL permite tirar partido dos princípios de programação paralela, podendo criar uma tarefa que é executada por vários processadores em simultâneo. Em linguagem do OpenCL a estas tarefas chamamos "kernels". Um kernel é uma função escrita para ser executada em um ou mais dispositivos suportados pelo OpenCL. Os kernels são enviados para os dispositivos pelas aplicações hospedeiras (host applications). Uma aplicação hospedeira é uma aplicação normal escrita em C ou C++ que corre no sistema a que

chamamos hospedeiro (host). Em muitos casos o hospedeiro despacha os kernels para o GPU, podendo no entanto ser executados no mesmo CPU onde está a correr a aplicação.

As aplicações fazem a gestão dos dispositivos de computação através de uma estrutura a que se chama *contexto*. Para criar um kernel, o hospedeiro selecciona uma função de um "programa". De seguida associa-lhe os dados como argumentos e despacha-o para uma fila de comandos (command queue). Esta fila de comandos é o mecanismo pelo qual o hospedeiro diz ao dispositivo o que deve executar. Quando um kernel é colocado na fila, o dispositivo executa a função correspondente. O OpenCL pode configurar dispositivos diferentes para tarefas diferentes o que é superior em relação a outros kits de programação que só exploram paralelismo de dados.

Documentos úteis poderão ser encontrados aqui:
https://www.dropbox.com/sh/9rxayk3j2alplbr/gQK5_WXJpv

Uma operação que é muito frequente utilizar-se no processamento de imagens é a convolução com máscaras bidimensionais. De facto variando os valores da máscara podem-se obter efeitos completamente diferentes. Uma média simples pode ser obtida com a seguinte máscara $3 \times 3$

$$\begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}.$$

Podemos aplicar uma filtragem Gaussiana para remover o ruído com

$$\begin{bmatrix} 6.9625 \times 10^{-8} & 2.8089 \times 10^{-5} & 0.00020755 & 2.8089 \times 10^{-5} & 6.9625 \times 10^{-8} \\ 2.8089 \times 10^{-5} & 0.011332 & 0.083731 & 0.011332 & 2.8089 \times 10^{-5} \\ 0.00020755 & 0.083731 & 0.61869 & 0.083731 & 0.00020755 \\ 2.8089 \times 10^{-5} & 0.011332 & 0.083731 & 0.011332 & 2.8089 \times 10^{-5} \\ 6.9625 \times 10^{-8} & 2.8089 \times 10^{-5} & 0.00020755 & 2.8089 \times 10^{-5} & 6.9625 \times 10^{-8} \end{bmatrix}.$$

Podemos ainda aproximar o gradiente com uma máscara de diferenciação vertical e uma horizontal

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Como último exemplo temos o Gaussiano que calcula a soma das derivadas de 2ª ordem.

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

## 17.1 Trabalho a realizar

**Exercício 17.1**
Escreva o código do "kernel"em OpenCL que dada uma imagem de dimensão $M \times N$ retorne a mesma convolvida com o kernel dado.

Nota: poderão começar com uma máscara fixa tipo Laplaciano e só depois passar a usar uma máscara passada como argumento.

**Exercício 17.2**
Escrever uma versão OpenCL da simulação do lago. Para isso deve receber como argumento a matriz do lago que deverá ser devolvida modificada, as dimensões da mesma e quantas épocas é que se deve simular (de cada vez que for chamada).

# Questões típicas de exame

# 18

Procure responder às questões que se seguem. Note que apenas respostas completas e totalmente certas são válidas.

**Exercício 18.1**
Alguns microprocessadores utilizam um esquema de variação dinâmica da frequência de trabalho por forma a controlar a potência dissipada. Isto permite que a frequência seja aumentada "a pedido"do sistema operativo para que uma aplicação possa "avançar"mais rapidamente. No entanto isto leva a uma dissipação de calor que não é sustentável por muito tempo. Assim o processador usando um sensor de temperatura pode reduzir a frequência quando detecta um sobreaquecimento. Considerando que é necessário reduzir a potência dissipada de 200W para 100W qual a redução em frequência necessária?

**Resposta:**
Sabendo que a potência dinâmica dissipada é dada por

$$Power_{dynamic} = \frac{1}{2} Capacitive\_load \times Voltage^2 \times Frequency$$

Então para reduzir a potência para metade teremos de reduzir a frequência para metade.

**Exercício 18.2**
Porque razão a variação da frequência do relógio de um processador pode ser particularmente importante para os dispositivos móveis? Qual o efeito que isso poderá ter na execução de um determinado programa computacionalmente intensivo?

**Resposta:**
Sabendo que a potência dissipada está relacionada com a razão com que se consome a energia, isto pode ser aproveitado nos dispositivos móveis para tentar reduzir o consumo dos mesmos durante os períodos de inactividade. Assim pode-se aumentar a frequência do processador nos períodos em que há trabalho para executar e nos periodos em que não há tarefas pode-se reduzir a frequência de comutação para assim reduzir a potência dissipada e por consequência a energia consumida. Respondendo à segunda parte da questão para um determinado programa do tipo mencionado a redução da frequência não irá reduzir a energia consumida pois irá aumentar o tempo de execução. Ou seja a energia consumida será a mesma para executar o programa independentemente da frequência de trabalho de determinado processador. Como já foi dito, podemos apenas poupar energia reduzindo a frequência nos períodos em que o processador não está a fazer trabalho útil.

**Exercício 18.3**
Explique detalhadamente quais os tipos de organização de caches existentes e como funcionam recorrendo a exemplo(s).

**Exercício 18.4**
Porque razão existem nos processadores actuais dois (ou três) níveis de cache?

**Exercício 18.5**
A introdução de sistemas com múltiplos processadores obrigou a que houvesse meios de comunicação entre os processadores. Diga em que aspecto isso pode ter a ver com as caches dos mesmos.

**Exercício 18.6**
A introdução de pipelining nos processadores foi possível graças à introdução de buffers entre etapas. Qual a função desses buffers?

**Exercício 18.7**
De que forma o forwarding permite reduzir os bloqueamentos (stalls) num pipeline?

**Exercício 18.8**
Como funciona um tournament predictor?

**Exercício 18.9**
Para que servem e qual é a vantagem dos preditores.

**Exercício 18.10**
Considerando um buffer de predição de saltos (BTB) onde as penalizações associadas são

| Inst. no buffer | Predição | Salto foi | Ciclos de penalização |
|---|---|---|---|
| sim | tomado | tomado | 0 |
| sim | tomado | não tomado | 2 |
| não | | tomado | 2 |
| não | | não tomado | 0 |

Se considerarmos que 90% das predições estão correctas e que 90% dos saltos tomados estão no buffer, qual é a penalização média do BTB?

**Resposta:**
De acordo com a tabela, 10% das predições tomadas estão erradas, logo isto é $10\% \times 90\%$. Há ainda 10% de saltos tomados que não estão no buffer. Daí que

$$\text{penalização} = (0.1 \times 0.9 + 0.1) \times 2 = 0.38 \text{ ciclos}$$

**Exercício 18.11**
Sabendo que um preditor (n,m) usa o comportamento dos n últimos saltos para juntamente com $l$ bits do endereço, selecionar de entre os $2^{(n+l)}$ predadores de m bits, aquele que se vai usar no salto corrente. Quantos bits existem nos predadores seguintes (0,2), (1,2) e (2,2) sabendo que o total de entradas na tabela de preditores é 4096?

**Resposta:**
Dado que o numero de entradas da tabela é fixo, ao aumentar o número de bits do histórico dos saltos que é usado, reduz-se o mesmo numero de bits do endereço. Ou seja no caso (2,2) como $4096 = 2^{12}$, então usaríamos apenas 10 bits do endereço. Assim o número de bits seria sempre de $4096 \times 2 = 8192$.

**Exercício 18.12**
Numa máquina com a arquitetura de Tomasulo, sabendo que apenas a primeira instrução terminou. Qual o estado de cada uma das instruções da sequência listada?

```
l.d f1,32(r1)
l.d f2,44(r2)
mul.d f0,f2,f1
sub.d f8,f2,f1
add.d f9,f1,f4
```

```
div.d f10,f1,f3
```

**Resposta:**
Dado que apenas a primeira instrução completou, o valor a usar nas instruções sub,add e div já está disponível. No entanto a segunda instrução ainda não completou pelo que o valor de f2 ainda não está disponível. Logo estão em execução as instruções l.d, add.d e div.d, tendo sido lançadas mas não iniciadas as instruções mul.d e sub.d, por dependerem de f2.

**Exercício 18.13**
Considerando um sistema com 8 processadores, qual será o tempo médio até à falha (MTTF) considerando que: cada processador tem MTTF de 4500 horas, o disco duro tem MTTF de 9 milhões de horas, e a fonte de alimentação tem MTTF de 30000 horas.

**Resposta:**

$$\begin{aligned} FIT &= 8 \times \frac{1}{4500} + \frac{1}{9000000} + \frac{1}{30000} \\ &= \frac{16000}{9000000} + \frac{1}{9000000} + \frac{300}{30000} \\ &= \frac{16301}{9000000} \end{aligned}$$

$$MTTF = \frac{9000000}{16301} = 552 \text{ horas}$$

**Exercício 18.14**
Considerando um processador cujas estatísticas de utilização típica mostram que: das instruções executadas 50% são de aritmética inteira(A), 20% de aritmética de virgula flutuante(B) e 30% são instruções de salto(C). Sabendo que em média as instruções A demoram 3 ciclos e as B demoram 30 ciclos. Se num determinado momento os arquitetos tiverem de escolher entre melhorar a velocidade de execução de $1,3$ vezes para as A ou 10 vezes para as B, qual a escolha mais vantajosa.

**Resposta:**
TBD

# Bibliografia

[1] “GNU gprof manpage.” [Online]. Available: http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html

[2] “The OpenMP® API specification for parallel programming.” [Online]. Available: http://www.openmp.org

[3] “Mpi documents.” [Online]. Available: http://www.mpi-forum.org/docs/

[4] MPI, “Web pages for mpi routines.” [Online]. Available: http://www.mcs.anl.gov/research/projects/mpi/www/www3/

[5] “Mpi tutorial.” [Online]. Available: http://mpitutorial.com/

[6] “Mpi tutorials.” [Online]. Available: https://computing.llnl.gov/tutorials/mpi/