Projeto 4 – Memória Cache

Gabriel Albuquerque  
 Departamento de Computação  
Universidade Federal Rural de Pernambuco  
 Recife, Pernambuco, Brasil  
 gabriel.oliveira.gt@gmail.com

Fábio Alves  
 Departamento de Computação  
 Universidade Federal Rural de Pernambuco  
 Recife, Pernambuco, Brasil  
 fbio.alves095@gmail.com

Wilson Medeiros  
 Departamento de Computação  
 Universidade Federal Rural de Pernambuco  
 Recife, Pernambuco, Brasil  
 wilneto17@gmail.com

Neste relatório foi feita a comparação dos algoritmos de ordenação, Quicksort e Radix sort, utilizando diferentes configurações de memória cache. Esses dois algoritmos de ordenação funcionam de maneiras bem diferentes. O radixsort ordena comparando as casas decimais de todos números do array. Precisa de um array auxiliar para efetuar as trocas e por isso gasta muita memória. O quicksort é um algoritmo que depende da implemenação para ter resultados ótimos. Pode ter implementações recursivas ou iterativas e gasta menos memória.

radixsort:

addi $sp, $sp, -4

sw $ra, 0($sp)

jal alocar\_array\_auxiliar # $s0

la $s1, bucket # $s1

jal get\_max # $s2

la $a2, array

li $s4, 1 # valor do expoente em $s4

while:

div $t0, $s2, $s4 # enquanto (maior/expoente) > 0

beqz $t0, end\_while

jal zerar\_array\_bucket

jal contar\_quantidade\_de\_digitos

jal incrementar\_indices\_anteriores

jal posicionar\_valores\_array\_auxiliar

jal copiar\_array\_auxiliar

mul $s4, $s4, 10

j while

end\_while:

lw $ra, 0($sp)

addi $sp, $sp, 4

jr $ra

Utilizamos o Data cache simulator do MARS4.5 para a análise dos dados. Os gráficos fazem a comparação do array em relação ao tamanho do array e o tempo de execução. O tempo de execução foi calculado de acordo com o número de hits e misses, considerando que o tempo de hit é um e o do miss mil vezes o tempo do hit.

**RADIX SORT**

O Radix sort foi implementado com base nas comparações dos dígitos menos significativos para os mais significativos. Buscamos o maior valor do array e o algoritmo executará a quantidade de digitos que esse valor possuir. A cada iteração preenchemos o array bucket com os valores do array principal que possuam os valores do dígito atual. Com base no array bucket preenchemos o array auxiliar, que tem o mesmo tamanho do array principal, com os valores ordenados com base no dígito atual. Por fim, copiamos esse array auxiliar para o array principal e comparamos o próximo dígito. Se tivermos comparado todos os dígitos do maior valor do array o algoritmo termina e o array principal estará ordenado.

No código ao lado é possível ver essa implementação do Radix sort em sua parte principal.

**QUICKSORT**

# t0 = &arr[p]

mult $s2, $t7

mflo $t6 # t6 = p \* 4bit

add $t0, $t6, $a0 # t0 = &arr[p]

# t1 = &arr[r]

mult $s1, $t7

mflo $t6 # t6 = r \* 4bit

add $t1, $t6, $a0 # t1 = &arr[r]

# Swap

lw $t2, 0($t0)

lw $t3, 0($t1)

sw $t3, 0($t0)

sw $t2, 0($t1)

# quick(arr, left, r - 1)

# set arguments

move $a2, $s1

addi $a2, $a2, -1 # a2 = r - 1

jal quicksort

# pop stack

lw $a1, 12($sp) # load a1

lw $a2, 16($sp) # load a2

lw $ra, 20($sp) # load ra

# quick(arr, r + 1, right)

# set arguments

move $a1, $s1

addi $a1, $a1, 1 # a1 = r + 1

jal quicksort

# pop stack

lw $a1, 12($sp) # load a1

lw $a2, 16($sp) # load a2

lw $ra, 20($sp) # load ra

# return

lw $s0, 0($sp) # load s0

lw $s1, 4($sp) # load s1

lw $s2, 8($sp) # load s2

addi $sp, $sp, 24 # Adjest sp

jr $ra

If\_quick1\_jump:

# SWAP (arr[l], arr[r])

li $t7, 4 # t7 = 4

# t0 = &arr[l]

mult $s0, $t7

mflo $t6 # t6 = l \* 4bit

add $t0, $t6, $a0 # t0 = &arr[l]

# t1 = &arr[r]

mult $s1, $t7

mflo $t6 # t6 = r \* 4bit

add $t1, $t6, $a0 # t1 = &arr[r]

# Swap

lw $t2, 0($t0)

lw $t3, 0($t1)

sw $t3, 0($t0)

sw $t2, 0($t1)

j Loop\_quick1

Loop\_quick1\_done:

# return

lw $s0, 0($sp) # load s0

lw $s1, 4($sp) # load s1

lw $s2, 8($sp) # load s2

addi $sp, $sp, 24 # Adjest sp

jr $ra

quicksort:

# store $s and $ra

addi $sp, $sp, -24 # Adjest sp

sw $s0, 0($sp) # store s0

sw $s1, 4($sp) # store s1

sw $s2, 8($sp) # store s2

sw $a1, 12($sp) # store a1

sw $a2, 16($sp) # store a2

sw $ra, 20($sp) # store ra

# set $s

move $s0, $a1 # l = left

move $s1, $a2 # r = right

move $s2, $a1 # p = left

# while (l < r)

Loop\_quick1:

bge $s0, $s1, Loop\_quick1\_done

# while (arr[l] <= arr[p] && l < right)

Loop\_quick1\_1:

li $t7, 4 # t7 = 4

# t0 = &arr[l]

mult $s0, $t7

mflo $t0 # t0 = l \* 4bit

add $t0, $t0, $a0 # t0 = &arr[l]

lw $t0, 0($t0)

# t1 = &arr[p]

mult $s2, $t7

mflo $t1 # t1 = p \* 4bit

add $t1, $t1, $a0 # t1 = &arr[p]

lw $t1, 0($t1)

# check arr[l] <= arr[p]

bgt $t0, $t1, Loop\_quick1\_1\_done

# check l < right

bge $s0, $a2, Loop\_quick1\_1\_done

# l++

addi $s0, $s0, 1

j Loop\_quick1\_1

Loop\_quick1\_1\_done:

# while (arr[r] >= arr[p] && r > left)

Loop\_quick1\_2:

li $t7, 4 # t7 = 4

# t0 = &arr[r]

mult $s1, $t7

mflo $t0 # t0 = r \* 4bit

add $t0, $t0, $a0 # t0 = &arr[r]

lw $t0, 0($t0)

# t1 = &arr[p]

mult $s2, $t7

mflo $t1 # t1 = p \* 4bit

add $t1, $t1, $a0 # t1 = &arr[p]

lw $t1, 0($t1)

# check arr[r] >= arr[p]

blt $t0, $t1, Loop\_quick1\_2\_done

# check r > left

ble $s1, $a1, Loop\_quick1\_2\_done

# r--

addi $s1, $s1, -1

j Loop\_quick1\_2

Loop\_quick1\_2\_done:

# if (l >= r)

blt $s0, $s1, If\_quick1\_jump

# SWAP (arr[p], arr[r])

li $t7, 4 # t7 = 4

# t0 = &arr[p]

mult $s2, $t7

mflo $t6 # t6 = p \* 4bit

add $t0, $t6, $a0 # t0 = &arr[p]

# t1 = &arr[r]

mult $s1, $t7

mflo $t6 # t6 = r \* 4bit

add $t1, $t6, $a0 # t1 = &arr[r]

# Swap

lw $t2, 0($t0)

lw $t3, 0($t1)

sw $t3, 0($t0)

sw $t2, 0($t1)

# quick(arr, left, r - 1)

# set arguments

move $a2, $s1

addi $a2, $a2, -1 # a2 = r - 1

jal quicksort

# pop stack

lw $a1, 12($sp) # load a1

lw $a2, 16($sp) # load a2

lw $ra, 20($sp) # load ra

# quick(arr, r + 1, right)

# set arguments

move $a1, $s1

addi $a1, $a1, 1 # a1 = r + 1

jal quicksort

# pop stack

lw $a1, 12($sp) # load a1

lw $a2, 16($sp) # load a2

lw $ra, 20($sp) # load ra

# return

lw $s0, 0($sp) # load s0

lw $s1, 4($sp) # load s1

lw $s2, 8($sp) # load s2

addi $sp, $sp, 24 # Adjest sp

jr $ra

If\_quick1\_jump:

# SWAP (arr[l], arr[r])

li $t7, 4 # t7 = 4

# t0 = &arr[l]

mult $s0, $t7

mflo $t6 # t6 = l \* 4bit

add $t0, $t6, $a0 # t0 = &arr[l]

# t1 = &arr[r]

mult $s1, $t7

mflo $t6 # t6 = r \* 4bit

add $t1, $t6, $a0 # t1 = &arr[r]

# Swap

lw $t2, 0($t0)

lw $t3, 0($t1)

sw $t3, 0($t0)

sw $t2, 0($t1)

j Loop\_quick1

Loop\_quick1\_done:

# return

lw $s0, 0($sp) # load s0

lw $s1, 4($sp) # load s1

lw $s2, 8($sp) # load s2

addi $sp, $sp, 24 # Adjest sp

jr $ra

REFERÊNCIAS

[1] Harkjong Son, QuickSort em Assembly MIPS em: https://github.com/hakjong/mips-qsort

Conference Name:ACM Woodstock conference

Conference Short Name:WOODSTOCK’18

Conference Location:El Paso, Texas USA

ISBN:978-1-4503-0000-0/18/06

Year:2018

Date:June

Copyright Year:2018

Copyright Statement:rightsretained

DOI:10.1145/1234567890

RRH: F. Surname et al.

Price:$15.00