

Exercícios de PCD

Alemao: Miguel Sifreia da Rocha Jr.

<http://miguelrochajr.com>

Capítulo 1

1.1-) Como foi dito no enunciado, façamos inicialmente para gerar todos N é dividível pelo número de processadores P . Se termos um vetor de tamanhos m , termos m_count para cada processador:

$$m_count = \frac{m}{P}$$

Note agora que o primeiro index, ou como dito na questão my_first_i não é um múltiplo do m_count de cada processador. Note também que o multiplicador serve a identificação dos processadores, isto é, seu rank.

$$my_first_i = Rank * m_count$$

Para o my_last_i , agora que sabemos quantas rotinas cada processador lleva, terá o seu my_first_i somado ao seu m_count .

$$My_last_i = my_first_i + m_count$$

Agora, quando m não é múltiplo de P , temos uma diferença; nem todos os processadores terão o mesmo número de rotinas. Uma quantidade de $m \% P$ de processadores terá um valor a mais. Assim, considerando que π é o número de processadores com 1 valor a mais e q a quantidade de processadores com m / P rotinas - considerando uma divisão inteira, temos:

$$\pi = m \% \cdot p$$

$$g = m/p$$

Neste caso, dada a divisão de m por p , o valor das cores é igual ao menor número que é maior ou igual a π . Como o resto da divisão é zero, não temos que $rank = n$. Para as demais cores, o valor inteiro de g é igual ao número de valores que recebeu. Este procedimento gera o seguinte código em C++:

$$g = m/p; // Note que este é uma divisão INTEIRA$$

$$\pi = m \% \cdot p;$$

if ($rank < n$) {

$$m_count = g + 1;$$

$$my_first_i = rank * m_count;$$

} else {

$$m_count = g;$$

$$my_first_i = rank * m_count + \pi;$$

}

$$My_last_i = my_first_i + m_count$$

<http://miguelrochajr.com>

O código acima vai colocar os $(n-1)$ primeiros cores com o valor a mais. Os outros cores receberam o valor inteiro de m/p valores.

Para transformar o código acima em uma função, devemos obter os intervalos mutuamente exclusivos do if e do else: $(rank < n) \cup (rank \geq n)$. É importante notar também que estas duas expressões saem resposta é booleana. Desta forma, temos as seguintes funções:

$$my_first_i = (rank < my_p) \left[rank * \left(\frac{m}{p} + 1 \right) \right] + (rank \geq my_p) \left[rank * \left(\frac{m}{p} \right) + my_p \right]$$

$$my_last_i = (rank < my_p) \left[my_first_i + \frac{m}{p} + 1 \right] + (rank \geq my_p) \left[my_first_i + \frac{m}{p} \right]$$

1.2-) Primeiro, observamos a regularidade, para $m=12$, temos:

i	0	1	2	3	4	5	6	7	8	9	10	11
2	4	6	12	20	30	42	56	72	90	110	132	

Agora, para $P=4$, note que se fizéssemos como na questão 1.1, teríamos:

	i	load
Cone 0	0: [0, 1, 2]	$2+4+6 = 12$
Cone 1	1: [3, 4, 5]	$12+20+30 = 62$
Cone 2	2: [6, 7, 8]	$42+56+72 = 170$
Cone 3	3: [9, 10, 11]	$90+110+132 = 332$

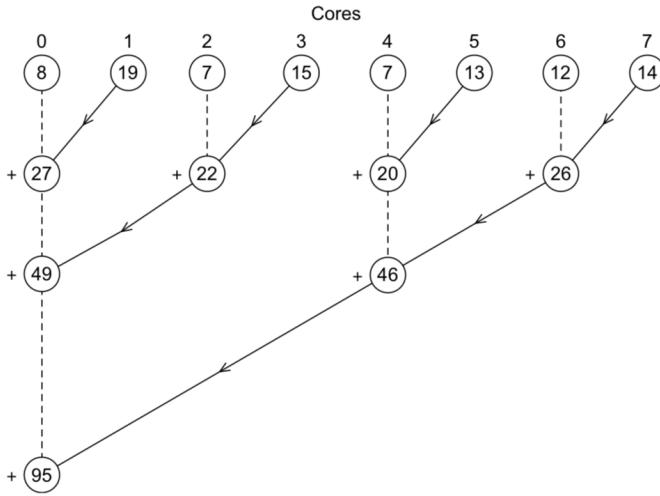
Obviamente não temos uma carga balanceada. Uma alternativa seria assimilar em cada regularidade de $m \%, P$ dentro e depois somar. A tabela abaixo mostra o resultado para $m=12$ e $P=4$.

	i	load	Total load
Cone 0	0 7 8	2 56 72	130
Cone 1	1 6 9	4 42 90	136
Cone 2	2 5 10	6 30 110	146
Cone 3	3 4 11	12 80 132	164

Note que para este approach as cargas ficam bem mais平衡adas. Não perfeitas mas com grande melhoria em relação ao que foi mostrado no início da questão.

1.3-)

Inicialmente, analisemos a figura 1.1:



multiplicado por dois. Isto é, para o core 0, o primeiro representante é 1, depois 2 e 4. Assim, como sugerido pela questão, teremos uma regra de divisão que irá determinar se um core irá receber seu valor, e uma regra chamada core-difference para determinar qual core é parceiro de qual.

O pseudo código está na página seguinte.

Note que separamos os cores para receberem valores para somar. Outro ponto a se notar é que, inicialmente os cores reservados à direita vêm com os dígitos. Entretanto, a cada iteração, o core "representante" possui o nome do anterior.

<http://miguelrochajr.com>

division = 2;

cone-difference = 15;

sum = my_rank;

while (division <= ~~10~~ (ones)) {

if (my_rank % division == 0) {

Partner = my_rank + cone-difference;

recebe valor do parceiro local

3 else {

Partner = my_rank - cone-difference;

Envia valor para Partner local;

}

division * = 2;

cone-difference * = 2;

}

Na próxima página, há três interações do loop acima.

<http://miguelrochajr.com>

~~CORE~~

Iteration	0	1	2	3	4	5	6	7	Comments
1	PARTNER = 0+1=1 RCV FROM 1 SUM = 27	PARTNER = 1+1=2 SEND TO 0	PARTNER = 2+1=3 RCV FROM 3 SUM = 22	PARTNER = 3+1=2 SEND TO 2	PARTNER = 4+1=5 RCV FROM 5 SUM = 20	PARTNER = 5+1=6 SEND TO 4	PARTNER = 6+1=7 RCV FROM 7 SUM = 26	PARTNER = 7+1=6 SEND TO 6	div: SOR = 2 cone diff = 1
2	PARTNER = 0+2=2 RCV FROM 2 SUM = 29	END	PARTNER = 2+2=0 SEND TO 0	END	PARTNER = 4+2=6 RCV FROM 6 SUM = 46	END	PARTNER = 6+2=4 SEND TO 4	END	div: SOR = 4 cone diff = 2
3	PARTNER = 0+4=4 RCV FROM 4 SUM = 99	END	END	END	PARTNER = 4+4=8 SEND TO 0	END	END	END	div: SOR = 8 cone diff = 4

<http://miguelrochajr.com>

1.4-)

bitmask = 1;

division = 2;

sum = my_rank;

while (bitmask < #cores) {

partner = my_rank & bitmask;

if (my_rank % division == 0) {

receive value do partner;

sum += value recebido;

} else {

enviar my_sum para partner;

}

bitmask <<= 1; // 1 left bitshift. multiply by 2
division *= 2;

}

Note que o código acima tem o mesmo efeito do código da 1.3. Aqui a diferença é que o partner é definido antes. Esta forma se dá pela tabela mostrada no encadeado e recebe todo o core-difference pelo bitmask.

O bitmask também vai ser multiplicado por 2.

O bitshift para esse efeito garante isto.

Abaixo temos a demonstração dos parceiros de core zero:

[Iteration 1]: Partner = 000 & 001 = 001 = 1

[Iteration 2]: Partner = 000 & 010 = 010 = 2

[Iteration 3]: Partner = 000 & 100 = 100 = 4

Note que a mesma seleção entre "remetentes" e destinatários também é verdadeira.

1.5-)

O problema mais evidente é que os ultimos cones, nesse caso, para $P=7$ o 7º cone, não terão ou receberão cones que não existem. Isso pode ocorrer em 1º código ficar usados/não esperados o cone 7, que está esperando o resultado de um cone não existente.

Para corrigir isso, podemos adicionar a simples condição:

{ if [portman < # cones] }

Recebe valor de portman;

sempre + = valor recebido

}

1.6-)

c) Iniciaremos pelo C para termos a tabela
e ficar mais clara a dedução para a)
e b).

P	original	Truee
2	1	1
4	3	2
8	7	3
16	19	4
32	31	5
64	63	6
128	127	7
256	255	8
512	511	9
1024	1023	10
↓	↓	↓
P	P-1	log ₂ (P)

a) e b) As respostas estão
na tabela acima lado

1.7 -)

O exemplo é uma combinação de paralelismos de tempo e de dízimo. Em cada parte da soma em círculo, os cones competem os somos paralelismo. Isso pode ser visto como uma paralelismo de dízimos. Também, em cada fase, existe dois tipos de tarefas. Algumas cores estão encarregando suas somas e outras recebendo as somas parciais das outras cores. Esta parte pode ser vista como paralelismo de tarefas.

1.8 -)

a) As atividades paralelas seriam: limpeza do local, trazer a comida, fazer os panfletos para a festa e montar o palco, por exemplo.

Todas estas tarefas diferentes podem ser feitas ao mesmo tempo.

b) Problemas nessa paralelismo de dízimos para limpar o salão. Imagine que o salão é o dízimo a ser tratado (limpo) e cada pessoa tem a sua tarefa (varrer). Basta dividirmos o salão em quantidades iguais para cada pessoa.

c) Problemas designar a tarefas de preparar a comida para alguns professores. Neste grupo, eles não podem cozinhar de acordo com o tipo de comida: sanduíches, salgados e doces, por exemplo.

1.9.)

Na área de engenharia de computação, há muitos problemas que podem ser beneficiados com o uso de paralelismo. Um excelente exemplo é o processamento de imagens.

Aqui, uma imagem filtrada, ao invés de ser parseada completamente por um único filtro, poderia ser dividida em partes e elas serem processadas simultaneamente. Isto seria um exemplo de data-paralellism, onde um dado (a matriz) é dividido para que várias instâncias de uma tarefa possam ser executadas de forma simultânea.

Um exemplo destas tarefas concomitantes seria uma expressão aplicada em todos os pixels, como um filtro gaussiano. O filtro usado neste exemplo conta com uma máscara 3×3 . Sabendo que temos uma imagem de 120×120 pixels e que temos um processador quad-core, podemos dividir a em quatro partes de 30×30 que seriam processadas por cada um dos processadores. Isto é, ao invés de um filtro 3×3 parsear a matriz 120×120 , podemos fazer com que cada core, carregado com a instrução do filtro, execute a instrução de forma simultânea dos outros, acelerando o processo.