Universidade Federal de Santa Catarina
Centro de Florianópolis
Departamento de Informática
e Estatística

Lucas P. Tonussi

T1 Computação Paralela

Florianópolis
29 de agosto de 2019

Lucas P. Tonussi

T1 Computação Paralela

Trabalho LAB 01: programação paralela (multiprocessador).

Orientador: Prof. Dr. Odorico Machado Mendizabal

Universidade Federal de Santa Catarina
Centro de Florianópolis
Departamento de Informática
e Estatística

Florianópolis
29 de agosto de 2019

Lucas P. Tonussi

T1 Computação Paralela

Trabalho de Conclusão de Curso apresentado à Universidade Federal de Santa Catarina como requisito parcial para a obtenção do título de Engenheiro de Controle e Automação.

Comissão Examinadora

Prof. Dr. Odorico Machado Mendizabal
Universidade Federal de Santa Catarina
Orientador

Prof. Dr. Odorico Machado Mendizabal
Universidade Federal de Santa Catarina

Prof. Dr. Odorico Machado Mendizabal
Universidade Federal de Santa Catarina

Florianópolis, 29 de agosto de 2019

Dedico este trabalho a todos aqueles que, de alguma forma,
auxiliaram para a concretização desta etapa.

Agradecimentos

Agradeço a meu pai, minha mãe e a você.

"Train yourself to let go of everything you fear to lose."
(Yoda)

Resumo

Resumo em português.

Palavras-Chave: 1. paralelismo. 2. memória-compartilhada. 3. threads. 4. multi-núcleos.

Abstract

Abstract in english.

Keywords: 1. parallel. 2. shared-memory. 3. fork. 4. threads. 5. multi-cores.

Lista de figuras

Lista de tabelas

Lista de Siglas e Abreviaturas

PCS	*Parallel Computing Systems*
UFSC	*Universidade Federal de Santa Catarina*

Sumário

1 Introdução

O que é computação paralela? Computação paralela é uma abordagem de programação que visa distribuir múltiplas tarefas por múltiplos fluxos de processamento. Esses fluxos podem ser múltiplas posix threads, jvm threads, múltiplos FORK(), dentre outras. Essas formas são mecanismos de programação, que por sua vez serão executadas em múltiplos processadores, em uma mesma máquina. Essa máquina com múltiplos núcleos (cores) pode também fazer parte de uma rede de múltiplos computadores interligados por barramento e estarem fazendo parte de um cluster de processamento.

Então podemos ter múltiplos fluxos independentes de processamento dentro de um processo, executar esses mesmos em múltiplos núcleos (CPU ou GPU), e por fim também distribuir tarefas através de uma rede de computadores distribuídos e interligados. Essas ferramentas de software e hardware comentadas nos permitem processar muita informação, e fazer cálculos pesados de maneira mais acelerada a fim de receber resultados antecipadamente.

Nesse trabalho foi utilizado experimentos apenas com multiprocessador Intel® Core™ i5-7300HQ CPU @ 2.50GHz Quad Core (2 Cores Físicos e 2 Cores Virtuais). Na taxonomia de Flynn esse trabalho estaria dentro da categoria **MIMD**, isto é, Múltiplas Instruções para Múltiplos Dados (*Multiple Instruction, Múltiple Data*) onde se encaixam os multi-processadores Intel. Foi utilizado sistema Linux 64 Bits / Ubuntu 18.04.3 LTS. As implementações foram feitas em C (Posix) e Java 8 64 bits (Oracle), justamente para serem feitas comparações de desempenho.

2 Implementações

Os requisitos basicamente giram em torno de implementar programas que usufruem de paralelismo em C, Java, para ordenar lista de vetores de tamanho M por N. Em C os mecanismos utilizado devem ser compartilhamento de memória, threads (pthreads), e processos filhos (fork). Como threads já compartilham memória, então basta utilizar compartilhamento de memória para processos. Java não trabalha com fork então apenas o uso de threads será abordado em Java.

As implementações de código seguem basicamente a mesma tática de divisão de trabalho. Essa tática é organizar as threads/processos da forma que eles possam trabalhar em um conjunto de listas específico para cada thread ou processo em questão. Assim é possível balancear a carga de ordenação das listas de elementos. Optou-se por criar um número de threads equivalente ao número de cores para que não houvessem mais desperdícios de troca de contexto entre threads/processos, menos gastos com criação de threads/processos, menos gastos com swap, e menos overhead que o sistema operacional tem para administrar esses elementos em memória.

A seção a seguir discutirá pequenas partes dos códigos, isto é, as que são importantes para a compreensão do todo. Para começar podemos comentar que todos as implementações para esse trabalho, tanto: multithread (em java ou c), multiprocesses (em c), tem o mesmo tipo de cálculo para encontrar a lista de trabalho, que cada unidade de processamento paralelo, (thread, processo filho) irá ficar responsável. A quantidade de unidades de processamento está associada à quantida de cores na máquina. Se existem 4 cores então no caso da implementação multithread 4 threads serão criadas e cada thread ficará responsável por ordenar uma certa quantidade de listas. Já no caso da implementação multiprocesses 4 processos filhos serão criados e cada filho ficará responsável por ordenar uma certa quantidade de listas. Assim é preciso calcular um conjunto de trabalho para que cada uma dessas unidades de processamento paralelo possa ficar responsável, segue abaixo como é feito o cálculo:

```
1 int processingPackSize = (m + NUMBER_OF_PROCESSORS - 1) /
    NUMBER_OF_PROCESSORS;
2 if (processingPackSize <= 0)
3 {
4     processingPackSize = 1;
5 }
```

A variável processingPackSize contém o número de listas que deverão ser ordenadas. Mas atenção que o número aqui dá o número máximo de tarefas divididas o mais igualmente possível entre o número de cores disponíveis. NUMBER_OF_PROCESSORS é uma chamada de sistema (sysconf(_SC_NPROCESSORS_ONLN)) para podermos sa-

ber a quantidade de cores disponíveis. A seguir é feito mais cálculos para distribuir perfeitamente a quantidade de tasks (listas para ordenar) entre os cores.

2.1 Multi-Thread em C

A implementação multithread em C não tem muito mistério também, pthreads foram criadas para que cada thread ordene o seu conjunto de listas, nenhuma thread invade a área da outra. Cada thread recebe uma estrutura de dados com propriedades para que a thread possa manter um controle mínimo dos seus limites. O código foi executado 2x uma para Bubble Sort e outro para Quick Sort, apenas foi mudado a função da thread de thread_start_quick para thread_start_bubble. Por fim a implementação com threads permite que os valores de m e n sejam quaisquer valores maiores que 2, 2 porque o objetivo é ordenas valores em listas.

```
1  for (tnum = 0; tnum < num_threads; tnum++)
2  {
3      int startIndex = tnum * processingPackSize;
4      int endIndex = fmin(startIndex + processingPackSize, m);
5
6      tinfo[tnum].thread_matrix_row_position = tnum;
7      tinfo[tnum].number_of_elements = n;
8      tinfo[tnum].thread_num = tnum;
9      tinfo[tnum].processing_time = 0;
10     tinfo[tnum].startIndex = startIndex;
11     tinfo[tnum].endIndex = endIndex;
12
13     s = pthread_create(&tinfo[tnum].thread_id, NULL,
14                        &thread_start_quick, &tinfo[tnum]);
15     if (s != 0)
16         handle_error_en(s, "pthread_create");
17 }
```

É sempre necessário calcular para cada thread/processo o índice inicial e o índice final, o índice aqui é o índice da lista. Exemplo: Thread 1 deve ordenar as listas 0, 1, 2; A Thread 2 deve ordenar as listas 3, 4, 5; A Thread 3 deve ordenar as listas 6, 7, 8; A Thread 4 deve ordenar as listas 9, 10, 11. Cada uma dessas listas pode ter um número N de elementos.

Vale observar também que é necessário compartilhar separadamente a estrutura de dados "struct"com informações adicionais para ajudar as threads/processos a se organizar. As informações são:

- **m**: Quantidade de listas
- **n**: Quantidade de elementos para cada lista

- **valatile unfinished_tasks_counter**: Quantidade de listas para ordenar

```
struct shared_use_st
{
    int m;
    int n;
    volatile int unfinished\_tasks\_counter;
};
```

No caso do C é criado um vetor dinâmico (malloc) para alocar memória do tamanho de *m * n inteiros*. Foi escolhido a forma de uma vetor unidimensional para facilitar o compartilhamento de memória. Simplesmente pelo fato que é mais fácil calcular o tamanho da memória compartilhada, em qualquer lugar que seja necessário acessarla. O uso de volatile aqui é para que o processo **Produtor.c** quando travar na linha WHILE(UNFINISHED_TASKS_COUNTER > 0), vai, então, pegar sempre um valor atualizar independente se perder o processador em um momento inoportuno.

2.2 Multi-Processo e Memória Compartilhada em C

Produtor.c (programa separado). Foi feito separadamente Produtor.c para realmente mostrar a comunicação através da memória compartilhada. O código abaixo é apenas para ilustrar a ordem em que o compartilhamento de memória de **int *unfinished_tasks_data;** foi feito. A ordem importa muito para fazer funcionar o compartilhamento. O compartilhamento da outra estrutura de dados foi omitido aqui, pois não é de maior complexidade explicativa. Por fim, por razões que não ficaram claras com respeito ao compartilhamento de memória em C, só é possível que a Matriz representando lista de vetores seja **quadrada**. E novamente, $M_{(2,2)}$ pois o objetivo é ordenar pelo menos 2 elementos.

```
int *unfinished_tasks_data;

shmid2 = shmget(kayv, sizeof(int) * (m * n), 0666 | IPC_CREAT);

shared_memory2 = (int *)shmat(shmid2, NULL, 0);

unfinished_tasks_data = malloc(sizeof(int) * (m * n));

// (m * n) tem que quadrado i.e 3,3; 6,6; 12,12; etc...

unfinished_tasks_data = shared_memory2;

while (shared_stuff->unfinished_tasks_counter > 0);

shmdt(shared_memory2)
```

```
16
17 shmctl(shmid2, IPC_RMID, 0)
```

Consumidor.c (programa separado). Aqui está a parte de atrelamento dos processos filhos "Consumidor.c"o pai faz todo o trabalho de configurar esse atrelamento e os filhos apenas herdam tudo isso depois que o fork foi realizado. Tanto pai como filhos executam ao final de suas tarefas comandos para se desatrelar da memória compartilhada.

```
1  int shmid2;
2
3  void *shared_memory2 = (void *)0;
4
5  shmid2 = shmget(kayv, sizeof(int) * (shared_stuff->m * shared_stuff->n),
       0666 | O_RDWR | IPC_CREAT);
6
7  shared_memory2 = (int *)shmat(shmid2, (void *)0, 0);
8
9  pid_t pids[NUMBER_OF_PROCESSORS];
10
11 for (size_t i = 0; i < sizeof(pids) / sizeof(pids[0]); i++) {
12     int startIndex = i * processingPackSize;
13     int endIndex = min(startIndex + processingPackSize, shared_stuff->m);
14 }
```

É preciso apenas garantir que o contador de tarefas pendentes está sendo decrementado de maneira sincronizada entre os processos. Pois cada processo, que acessa a memória compartilhada pelo PRODUTOR.C, vai apenas trabalhar em uma região do *array* de listas e não vai se intrometer na região da memória que outro processo estará trabalhando. Isso foi feito propositalmente, para facilitar a implementação dos ordenadores. Senão ordenadores bem mais sofisticados teriam que ser implementados. A proposta aqui é simplesmente quebrar a tarefas em várias linhas de processamento independentes e concorrentes.

Uma segunda observação é necessária ser feita aqui. Veja que no código que segue abaixo está implementado a forma de inicialização do semáforo. Os parâmetros dessa função *sem_init* são o endereço do semáforo ("mutex"), depois um booleano 0, ou 1 para sinalizar se o processo quer compartilhar o semáforo com seus filhos (que é algo que queremos) e o terceiro é o valor que o contador do semáforo terá.

```
1  // No processo pai (os filhos copiam o mesmo)
2  if (sem_init(&mutex, 1, 1) < 0)
3  {
4      perror("semaphore initilization");
5      exit(0);
6  }
7
8  // Dentro do processo filho pid == 0
9  sem_wait(&mutex);
```

```
10 shared_stuff->unfinished_tasks_counter -= (tinfo->endIndex - tinfo->
       startIndex);
11 sem_post(&mutex);
```

Um pedaço de código comum à todos os processos criados produtor.c (pai), consumidor.c (pai e filhos) é o que segue abaixo, esse código é necessário para todos os processos para cada processo fechar seu respectivo acesso à memória compartilhada. Basicamente é isso que é feito, se falhar simplesmente a tática adotada foi finalizar o processo que está tentando encerrar seu acesso a memória compartilhada.

```
1  if (shmdt(shared_memory2) == -1)
2  {
3      fprintf(stderr, "shmdt failed\n");
4      exit(EXIT_FAILURE);
5  }
6
7  if (shmctl(shmid2, IPC_RMID, 0) == -1)
8  {
9      fprintf(stderr, "shmctl(IPC_RMID) failed\n");
10     exit(EXIT_FAILURE);
11 }
12
13 if (shmdt(shared_memory) == -1)
14 {
15     fprintf(stderr, "shmdt failed\n");
16     exit(EXIT_FAILURE);
17 }
18
19 if (shmctl(shmid, IPC_RMID, 0) == -1)
20 {
21     fprintf(stderr, "shmctl(IPC_RMID) failed\n");
22     exit(EXIT_FAILURE);
23 }
```

2.3 Multi-Thread em Java

A implementação usando apenas *Threads* Java ofereceu muitas facilidades, de codificação. É assim para aumentar a produtividade do programador.

```
1  int tasksQuantity = 31, n = 5, processingPackage = 0, numberOfCores = 0;
2  numberOfCores = Runtime.getRuntime().availableProcessors();
3
4  processingPackage = (tasksQuantity + numberOfCores - 1) / numberOfCores;
5
6  if (processingPackage <= 0) {
7      numberOfCores = tasksQuantity;
```

```
8 }
9
10 Matrix mat1 = new Matrix(tasksQuantity, n);
11 Matrix mat2 = new Matrix(tasksQuantity, n);
12
13 launchOrdenadoresBubble(mat1, tasksQuantity, processingPackage,
       numberOfCores);
14
15 joinOrdenadoresBubble();
16
17 showTimesOrdenadoresBubble();
18
19 launchOrdenadoresQuick(mat2, tasksQuantity, processingPackage,
       numberOfCores);
20
21 joinOrdenadoresQuick();
22
23 showTimesOrdenadoresQuick();
```

O código em Java acima, basicamente, faz o mesmo que já foi comentado anteriormente, para C. O cálculo da quantidade de tarefas é feito do mesmo jeito. Dois objetos Matrix m1, e m2, foram criados, ou seja, m1 é usado para testar o Bubble Sort e m2 é usado para testar o Quick Sort.

O código a seguir ilustra como é feito em java para criar um número de OrdenadorBubbleSort obs igual ao número de cores. O mesma maneira é feita para OrdenadorQuickSort oqs. A quantidade de cores que Java detecta como ativos define a quantidade de Threads utilizadas.

```
1 private static void launchOrdenadoresBubble(Matrix mat, int m, int
      processingPackSize, int cores) {
2     for (int i = 0; i < cores; i++) {
3         int startIndex = i * processingPackSize;
4         int endIndex = Math.min(startIndex + processingPackSize, m);
5
6         OrdenadorBubbleSort ordenadorBubbleSort = new OrdenadorBubbleSort(
      mat, startIndex, endIndex);
7         ordenadoresBubble.add(ordenadorBubbleSort);
8
9         Thread t = new Thread(ordenadorBubbleSort);
10        bubbleThreads.add(t);
11        t.start();
12    }
13 }
```

3 Resultados

Antes de prever os resultados, sabe-se que C tem maior efetividade para poder fazer computações paralelas e distribuídas com mais eficiência. Até mesmo quando se trata de GPUs programar em "C"para GPUs é muito eficiente, sendo possível dar vazão para teraflops de teraflops de cálculos em muito pouco tempo. As métricas dos resultados serão apresentados a seguir, com base em marcação de tempo de execução livre de entrada e saída da seção de código que faz a ordenação vetor-a-vetor. Ao final é feita uma somatória dos resultados e então apresentado para comparação, evidentemente.

3.0.1 Pthreads (Posix) em C - Quick Sort

int m = 40, n = 40;

```
Timings for processing by each thread.
Thread Using Quick Sort [id: 0], Accumulated time: [0.000188 seconds].
Thread Using Quick Sort [id: 1], Accumulated time: [0.000203 seconds].
Thread Using Quick Sort [id: 2], Accumulated time: [0.000124 seconds].
Thread Using Quick Sort [id: 3], Accumulated time: [0.000075 seconds].

Sum of timings 0.000590 seconds.
```

3.0.2 Pthreads (Posix) em C - Bubble Sort

int m = 40, n = 40;

```
Timings for processing by each thread.
Thread Using Bubble Sort [id: 0], Accumulated time: [0.000380 seconds].
Thread Using Bubble Sort [id: 1], Accumulated time: [0.000360 seconds].
Thread Using Bubble Sort [id: 2], Accumulated time: [0.000203 seconds].
Thread Using Bubble Sort [id: 3], Accumulated time: [0.000110 seconds].

Sum of timings 0.001053 seconds.
```

3.0.3 Processos + M. Compart. em C - Quick Sort

Os endereços dão diferentes pois se trata de endereços de memória virtual diferentes. O sistema operacional sabe recuperar a partir desses endereços a memória física comum e compartilhada entre os processos.

Producer.c

int m = 40, n = 40;

```
Memory 1 attached at 7F76E000
Memory 2 attached at 7F76C000

(Não Ordenados)

100 79 85 10 46 79 2 1 43 54 20 89 37 98 32 73 90 9 44 32 99
59 16 36 98 100 40 88 5 10 79 56 41 63 66 38 94 19 38 36

24 10 24 60 7 7 33 97 15 28 80 65 86 95 100 83 46 39 22 51 1
100 6 41 15 23 78 8 41 68 95 65 29 70 24 87 28 8 35 95

...

83 15 83 11 41 19 48 16 93 60 4 36 14 90 15 58 95 6 38 82 52
97 56 58 44 93 55 99 86 47 51 68 62 33 78 54 4 77 69 48

37 25 35 50 14 49 60 60 6 49 93 10 45 48 67 40 40 73 90 25 72
93 44 85 77 73 38 32 50 59 79 38 83 14 87 48 14 98 7 72


(Ordenados)

1 2 5 9 10 10 16 19 20 32 32 36 36 37 38 38 40 41 43 44 46
54 56 59 63 66 73 79 79 79 85 88 89 90 94 98 98 99 100 100 (primeira linha)

1 6 7 7 8 8 10 15 15 22 23 24 24 24 28 28 29 33 35 39 41 41
46 51 60 65 65 68 70 78 80 83 86 87 95 95 95 97 100 100 (segunda linha)

...

4 4 6 11 14 15 15 16 19 33 36 38 41 44 47 48 48 51 52 54 55
56 58 58 60 62 68 69 77 78 82 83 83 86 90 93 93 95 97 99 (penúltima linha)

6 7 10 14 14 14 25 25 32 35 37 38 38 40 40 44 45 48 48 49 49
50 50 59 60 60 67 72 72 73 73 77 79 83 85 87 90 93 93 98 (ultima linha)
```

Consumer.c

```
Memory 1 attached at E83B1000
Memory 2 attached at E83AF000

[son] pid 12280 from [parent] pid 12279 and
value ou i=0, startIndex 0 endIndex 10.

time i took 0.000017 sorting 400 elements (quick sort)

[son] pid 12281 from [parent] pid 12279 and
value ou i=1, startIndex 10 endIndex 20.

time i took 0.000017 sorting 400 elements (quick sort)

[son] pid 12283 from [parent] pid 12279 and
value ou i=3, startIndex 30 endIndex 40.

time i took 0.000017 sorting 400 elements (quick sort)

[son] pid 12282 from [parent] pid 12279 and
value ou i=2, startIndex 20 endIndex 30.

time i took 0.000018 sorting 400 elements (quick sort)

Sum of all timings 0,000069 seconds
```

3.0.4 Processos + M. Compart. em C - Bubble Sort

Producer.c

int m = 40, n = 40;

Muitos vetores foram omitidos aqui no trabalho por causa que senão é muita informação redundante.

```
Memory 1 attached at 36367000
Memory 2 attached at 36365000

(Desordenados)

5 70 44 59 61 85 51 85 73 55 27 95 99 47 20 34 54 1 24
54 17 4 11 91 86 38 45 14 19 3 61 75 25 56 33 37 92 35 21 64
```

```
41 99 11 39 45 82 24 98 34 47 52 3 50 62 93 87 99 89 52 69
44 64 43 68 20 75 56 11 9 28 75 1 26 37 39 71 18 62 68 52

...

58 28 98 13 27 51 79 25 64 63 49 84 70 90 54 64 15 96 64 25
16 87 37 77 30 54 38 17 77 34 33 34 61 82 99 39 32 29 63 48

92 64 31 61 53 36 77 67 31 40 43 99 27 79 75 8 84 64 24 60
97 8 46 9 89 44 47 73 24 62 20 15 25 2 28 29 37 4 95 20

(Ordenados)

1 3 4 5 11 14 17 19 20 21 24 25 27 33 34 35 37 38 44 45 47
51 54 54 55 56 59 61 61 64 70 73 75 85 85 86 91 92 95 99 (primeira linha)

1 3 9 11 11 18 20 24 26 28 34 37 39 39 41 43 44 45 47 50 52
52 52 56 62 62 64 68 68 69 71 75 75 82 87 89 93 98 99 99 (segunda linha)

...

13 15 16 17 25 25 27 28 29 30 32 33 34 34 37 38 39 48 49 51
54 54 58 61 63 63 64 64 64 70 77 77 79 82 84 87 90 96 98 99 (penúltima linha)

2 4 8 8 9 15 20 20 24 24 25 27 28 29 31 31 36 37 40 43 44 46
47 53 60 61 62 64 64 67 73 75 77 79 84 89 92 95 97 99 (ultima linha)
```

Consumer.c

```
Memory 1 attached at BA19000
Memory 2 attached at BA17000

[son] pid 12804 from [parent] pid 12803 and
value ou i=0, startIndex 0 endIndex 10.

time i took 0.000029 sorting 400 elements (bubble sort)

[son] pid 12805 from [parent] pid 12803 and
value ou i=1, startIndex 10 endIndex 20.
```

```
time i took 0.000029 sorting 400 elements (bubble sort)

[son] pid 12806 from [parent] pid 12803 and
value ou i=2, startIndex 20 endIndex 30.

time i took 0.000031 sorting 400 elements (bubble sort)

[son] pid 12807 from [parent] pid 12803 and
value ou i=3, startIndex 30 endIndex 40.

time i took 0.000029 sorting 400 elements (bubble sort)

Sum of all timings 0,000118 seconds.
```

3.0.5 Threads Java - Quick Sort

m = 40, n = 40;

```
Sum Of All Ordenadores Using Quick Sort: Exec Duration (seconds): 0.0005531280
```

3.0.6 Threads Java - Bubble Sort

```
Sum Of All Ordenadores Using Bubble Sort: Exec Duration (seconds): 0.0015333090
```

4 Conclusões

São objetivos diferentes entre C e Java, e não é um problema comparar implementações visando o mesmo objetivo para averiguar qual se desempenha melhor. Todavia, C oferece uma liberdade de programação avançada, e Java esconde muita coisa do programador, essa pequena diferença mostra o motivo de C ser mais eficaz que Java para paralelismo.

Na implementação, é possível que eu tenha falhado na implementação das threads em c e por isso a implementação com threads em c perdeu para o quick sort implementado em Java $0.00055 < 0.00059$. Existe o risco, não muito óbvio, de que a execução não foi equivalente (i.e. beneficiação por cache-hits desigual entre as execuções). Todavia, fork+shm em C ganhou até mesmo das threads em C, e também ganhou das threads em Java. Foram feitas várias execuções e o cenário abaixo, na tabela, se repetiu de maneira bem semelhante.

tempo (segundos)	**implementação**	**ranking**
0.000590	pthreads quick sort	4º
0.001053	pthreads bubble sort	5º
0,000069	fork+shm quick sort	1º
0,000118	fork+shm bubble sort	2º
0,0002247000	java threads quick sort	3º
0,0020767500	java threads bubble sort	6º

Tabela 1 – Comparações Finais m = 40, n = 40

Referências Bibliográficas