UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Departamento de Ciência da Computação
DCC641 -- Fundamentos de Sistemas Paralelos e Distribuídos
Primeiro Exercício de Programação
Guilherme de Abreu Lima Buitrago Miranda - 2018054788

1 - Introdução

O primeiro exercício de programação da disciplina de Fundamentos de Sistemas Paralelos e Distribuídos tem como objetivo a paralelização de um programa já previamente implementado de forma sequencial. Tal paralelização deve ocorrer de duas formas distintas: uma usando paralelismo de dados, outra usando paralelismo de funções.

Nas seções a seguir, encontram-se detalhes relativos à implementação do trabalho, análise de desempenho, conclusões e referências bibliográficas.

2 - Implementação

Em razão da natureza do trabalho, a primeira etapa da implementação foi, em verdade, o pleno entendimento do código do arquivo *numcheckseq.c*, a fim de pensar em uma boa solução para o exercício. Dessa forma, em seguida, iniciou-se a implementação da versão usando paralelismo de dados, implementada no arquivo *numcheckdatapar.c,* na qual são criadas threads que executam um lote de números pelas 5 funções do arquivo *numchecks.c*.

Utilizando como base o arquivo *numchecksec.c*, a primeira alteração foi a inclusão de um array do tipo *pthread_t* de tamanho 8, conforme sugerido, assim como a definição e a criação de um array de mesmo tamanho da variável *datapar_args*, que é responsável por armazenar os dados necessários para a execução das threads (o início e o fim do lote de números, assim como a variável *ndigits*). Variáveis para o controle de condição de corridas também são inicializadas (mais detalhes adiante) e, por fim, há dois loops for*,* o primeiro que se dispare as threads e o segundo para realizar o join.

Ao passo que cada thread é iniciada, a execução da função *check_num* dá início. Nela, após a extração das variáveis de execução da thread passadas como argumento, é iniciado um loop for, do primeiro ao último número do lote cujo a thread é responsável e, nele, o processamento é feito de forma análoga ao programa sequencial. Contudo, antes de executar a função *update_max*, é feito um lock na variável mutex *update_max_lock*, seguido de um unlock em seguida. Tal sincronização é necessária pois, sem ela, é possível que duas ou mais threads queiram executar a função ao mesmo tempo, gerando uma condição de corrida que potencialmente atrapalha o resultado esperado para o programa.

Em seguida, fora do for loop dos números do lote, é feito o uso de uma segunda variável mutex: *counters_lock.* A mesma é usada pois, após a execução da thread, é necessário atualizar os contadores gerais do programa, a fim de que a saída contenha as informações obtidas por todas as threads. Assim como para o caso anterior, a sincronização justifica-se pois, sem a variável mutex, é possível que duas threads tentem atualizar as variáveis ao mesmo tempo, gerando uma condição de corrida que pode resultar na saída errada do programa.

Um detalhe menor de implementação mas que vale ser mencionado é a necessidade de criação de variáveis análogas à *all, pal, rep, sum, dou* e *fou* para os batches, já que, em termos de performance, é mais vantajoso só atualizar os contadores gerais do programa após a execução de todos os números de responsabilidade daquela thread.

Assim, após a execução dessa função, é feito o join de todas as threads, seguido da saída padrão do programa.

Já para a versão usando paralelismo de funções, a primeira abordagem pensada foi a execução de cada uma das funções em paralelo e, por fim, o agrupamento desses dados no programa principal. Contudo, conforme o enunciado fornecido, é necessário que o grosso do processamento deve ser executado por threads concorrentes, ficando vetada a combinação dos resultados gerados por cada thread a posteriori. Assim, foi necessário pensar em uma outra abordagem.

A princípio, pensou-se em uma matriz de booleanos de 5 linhas (cada uma das linhas representando uma das funções) por n + 1 colunas (sendo n a entrada do programa). Entretanto, a linguagem C não suporta o tipo booleano por definição e, ainda que suportasse, o endereçamento padrão na CPU é feito com no mínimo 1 byte, o que significaria que 5 bytes seriam usados para o controle de cada número.

Com intenção de buscar uma boa relação entre simplicidade do código e economia de memória, decide-se então trabalhar com um array de chars (menor tipo primitivo de C, com 2 bytes, conforme referência no final do documento). Assim, cada posição desse array representa o número de regras que o número em questão segue, variando, portanto, entre 0 (nenhuma regra) e 5 (todas as regras).

Dessa forma, a função main do arquivo *numcheckfuncpar.c* é bastante parecida com a do arquivo *numcheckdatapar.c*, com exceção da struct criada para passar os argumentos para a thread. Para o paralelismo de funções, as variáveis passadas são o *ndigits, maxnum* e *func_num*, a última representando qual é a função que a thread executa (sendo 0 a função que checa se é um palíndromo, 1 a função que checa sequências repetidas, etc.).

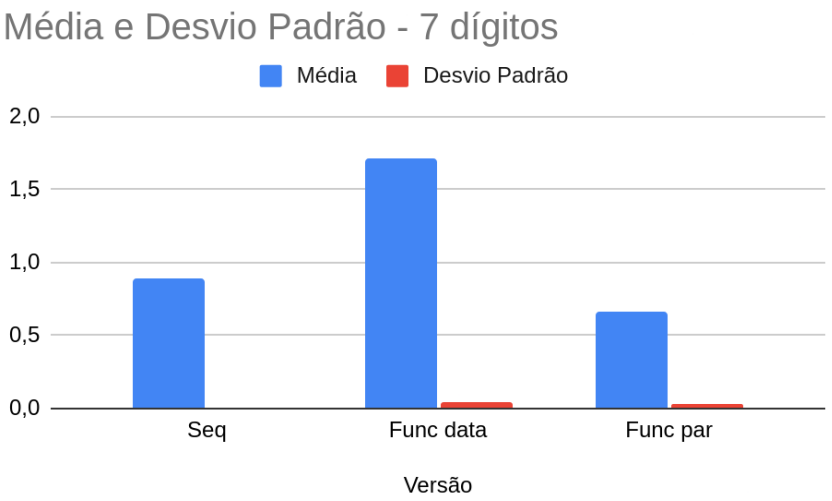
Após serem criadas, destarte, cada uma das 5 threads executa a função *check_func_par* (que, como dito anteriormente, verifica funções diferentes para cada uma das threads). O principal ponto de atenção é, portanto, após a obtenção de resultado das funções do arquivo *numcheckseq.c*. Se a condição for satisfeita, é necessário que se trave o mutex *counters_lock*, para, então, atualizar as variáveis de controle do programa, ou seja, os contadores (palindromes, repeated_seqs…), o *cond_array* (array de n + 1 posições de chars supracitado) e, enfim, a execução da função *update_max*. Por fim, é feito o destravamento da thread e a função termina sua execução. Após a execução das 5 threads, o main é responsável por juntar todas as threads e por último, imprimir a saída esperada do programa.

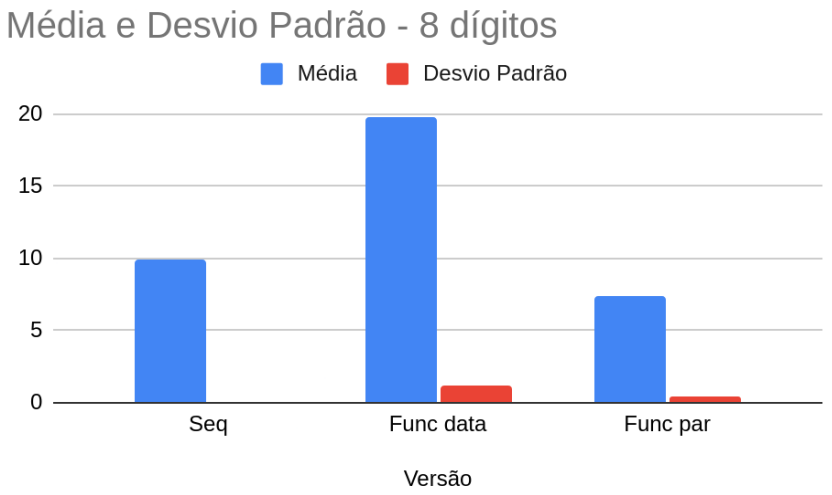
3 - Análise de Desempenho

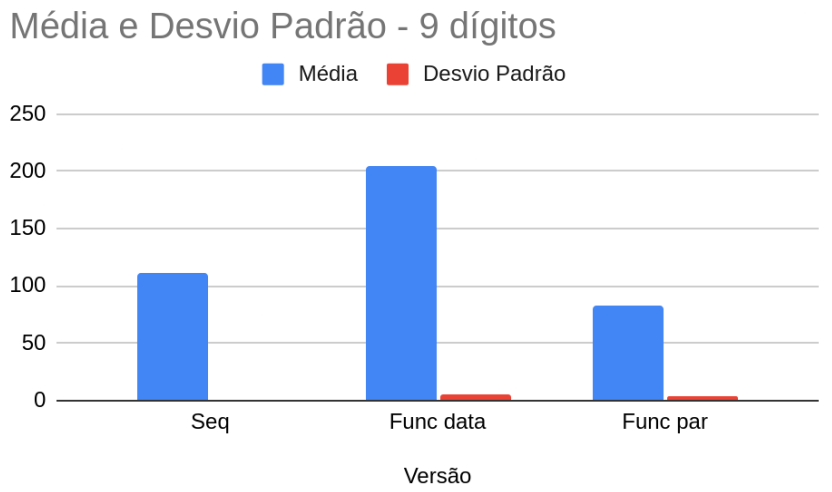
Para as medições analisadas abaixo, foram criados os seguintes cenários:

- Cenário 1: 9999999 (7 dígitos);
- Cenário 2: 99999999 (8 dígitos);
- Cenário 3: 999999999 (9 dígitos).

Cada um desses cenários executou para as três versões do programa (sequencial, paralelismo de dados e paralelismo de funções) por 10 vezes na seguinte máquina: i5-8250U (4 núcleos físicos e 8 núcleos lógicos) @ 1.6GHz com Turbo Max de 3.40 GHz, 8 GB de RAM no Sistema Operacional Ubuntu 20.04 LTS. As médias obtidas estão apresentadas abaixo (o eixo y está sempre em segundos):







Vê-se, em primeiro lugar, como a versão sequencial é estável - mesmo para a execução de 9 dígitos, em que os tempos obtidos são maiores, o desvio padrão segue muito baixo (0,05s). Isso ocorre pois, por se tratar de uma thread única

executando, o sistema operacional não tem dificuldade em gerenciá-la em conjunto com outras tarefas executadas concomitantemente (levando em consideração que os testes foram feitos numa CPU com 4 núcleos físicos).

Além disso, a implementação mais lenta é a de paralelismo de dados. Em razão da natureza do problema e, portanto, da necessidade do uso da variável mutex ao redor da execução da função *update_max*, a versão paralela ficou até mais lenta que a sequencial. Inclusive, em testes realizados durante a implementação do trabalho, para essa versão, quanto menos threads eram criadas, mais rápido o programa executava, aproximando dos tempos da versão sequencial, o que corrobora a tese de que o gargalo na execução está sendo causado pelo mutex.

Por fim, vê-se que a versão mais veloz é a paralelizada por funções. Esta é cerca de 30% mais rápida que a versão sequencial e 150% mais rápida que a versão paralelizada em decomposição por domínio. O principal motivo para tal é que os testes foram executados em uma máquina que lida bem com 5 threads e a versão implementada tem poucas verificações de variáveis mutexes, o que não gera um gargalo como observado na versão por domínio. Assim, foi obtido, portanto, um speedup interessante para o problema em questão.

4 - Conclusão

A implementação do trabalho foi bastante proveitosa para efetivamente fixar os conceitos expostos até o presente momento na disciplina. Dessa forma, o aprendizado se deu, sobretudo, durante o enfrentamento dos desafios no gerenciamento das condições de corrida apresentadas no código, especialmente na versão de decomposição de funções.

Não obstante, o trabalho também me possibilitou aprender, na prática, ainda mais a respeito da programação de threads com linguagem C, que só havia tido contato na disciplina de Sistemas Operacionais. Com isso, considero que o trabalho foi de grande proveito para o processo de aprendizagem.

5 - Referências Bibliográficas

- Why is a boolean 1 byte and not 1 bit of size? - https://stackoverflow.com/questions/4626815/why-is-a-boolean-1-byte-and-not-1-bit-of-size
- C data types - https://en.wikipedia.org/wiki/C_data_types#Main_types