Trabalho 1 para a Disciplina de Programação Paralela

Discentes

- Rubens Zandomenighi Laszlo GRR20206147
- Gabriel Razzolini Pires De Paula GRR20197155

Implementação

Implementação utilizando barreiras tal que cada thread executa o algoritmo de findKLeastProgram (acharKMenores). Utilizamos a função *parallel_findKLeast*, a qual é responsável pela criação da barreira e das threads, inicializando cada thread com a função findKLeastPartialElmts para achar os K menores elementos de suas faixas e após o término das threads em achar os K menores elementos de sua faixa de valores (utilizando max-threads), aguardar na barreira.

```
    findKLeastProgram(myIndex);
    pthread_barrier_wait( &parallelFindKLeast_barrier );
```

Sendo que o vetor de entrada (Input) é divido entre as threads tal que cada thread obtém os K menores elementos de sua faixa, sendo as faixas de valores por thread definidas como:

```
int firstFromRange = myIndex * nElements;
int lastFromRange = MIN( (myIndex+1) * nElements, nTotalElements ) -
1;
```

Após todas as threads terem concluído seu trabalho e estarem sincronizadas, utilza-se a função *concatenateOutputPortions*, executando mais uma vez o algoritmo de findKLeast(acharKMenores), porém com os max-heaps obtidos a partir de todas as threads, assim construindo a heap final com os K menores elementos de toda a faixa do vetor de Input. Nessa função são inseridos os K elementos da max-heap obtida a partir da thread 0, e após isso efetua a função *decreaseMax* para todos os elementos das max-heaps obtidas nas threads restantes.

A função de verificação do resultado *verifyOutput* é construída ordenando o vetor input com a função qsort da stdlib e comparadando quantos dos k primeiros elementos foram obtidos no heap-máximo obtido a partir da implementação descrita acima, caso o número de elementos achados = k primeiros elementos do vetor ordenado então está ok.

Descrição do Processador

```
Hostname:   j7
Processador:    i7-4770
CPU MHz 798,228
CPU max MHz 3900
CPU min MHz 800
L1d cache    128 KiB
L1i cache    128 KiB
L2 cache    1 MiB
L3 cache    8 MiB
flags:  fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx
pdpe1gb rdtscp lm constant_tsc arch_perfmon
```

Descrição dos experimentos e como foram feitas as medidas

Execução com nTotalElementos=100.000.000 e k=2048, rodando 10 vezes com 1-8 threads, utilizamos a biblioteca fornecida *chrono*, utilizando as funções da biblioteca, contabilizando em segundos o tempo percorrido e a vazão em n° operações/tempo total OPS :

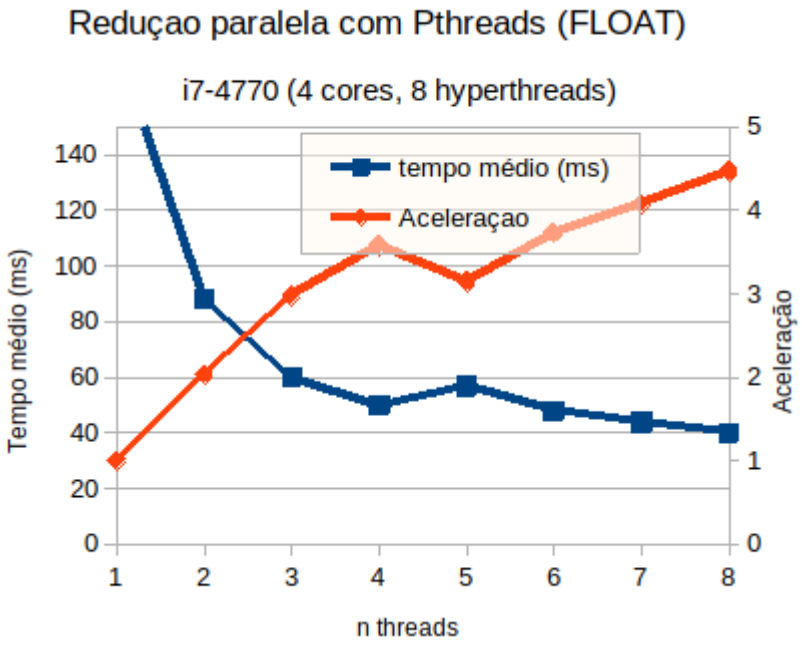
```
    chrono_reset( &runningTime );
    chrono_start( &runningTime );
    ...
        chrono_stop( &runningTime );

    verifyOutput(Input, Output, nTotalElements, k);

    chrono_reportTime( &runningTime, "runningTime" );

    // calcular e imprimir a VAZAO (numero de operacoes/s)

    //  A vazão deve ser reportada em MOPs (Mega Operacoes por
segundo)
    //  que é o numero de operações de insersão+decreaseMax
    //  feitas por segundo no Max-Heap pelo seu algoritmo paralelo
para
    //  uma dada quantidade de threads.

    double total_time_in_seconds = (double) chrono_gettotal(
&runningTime ) /
                                       ((double)1000*1000*1000);
    printf( "\nTotal time in seconds: %lf s\n", total_time_in_seconds
);


    double OPS = (nTotalElements)/total_time_in_seconds;
    printf( "Throughput: %lf MOPs/s\n", OPS );
```

Planilha de resultados sumarizando a vazao e aceleração

Planilha.ods

Gráfico (obtido da planilha) mostrando:

Tempo



Vazão