Paralelização híbrida e em múltiplos níveis de um algoritmo de contabilização de frequências de *k-mer*

Fabrício Vilasbôas[1], **Micaella Coelho**[1],
Carla Osthoff[1], **Kary Ocaña**[1], **Ana Tereza Vasconcelos**[1]

[1]Laboratório Nacional de Computação Científica

Resumo. *Este trabalho apresenta um estudo sobre o ganho de desempenho gerado com o aumento do nível de paralelismo de execução de um algoritmo de contabilização de k-mers de dados de amostras genômicas e metagenômicas. O algoritmo foi originalmente desenvolvido para permitir a execução em paralelo da contabilização de k-mers em uma arquitetura manycores de uma GPU. Neste trabalho apresentamos o ganho de desempenho ao implementarmos paralelismo para a execução paralela de módulos do algoritmo em múltiplos núcleos de uma CPU, em múltiplas GPU's e em múltiplos nós de processamento de uma cluster híbrida.*

Abstract. *This work presents a study on the performance gain generated with the increase of the level of parallelism of execution of a k-mers accounting algorithm of data of genomic and metagenomic samples. The algorithm was originally developed to allow the parallel execution of k-mer accounting on GPU´s manycores. This work presents the performance gain when implementing parallel execution from algorithm modules on multiple cores of a CPU, in multiple GPUs and in multiple processing nodes of a hybrid cluster.*

1. Introdução

Com o surgimento da tecnologia *NGS (Next Generation Sequencing)* muitos foram os avanços na área da pesquisa genômica. As técnicas de sequenciamentos mais avançados geravam 84 MB de dados por rodada e hoje, com o NGS, são gerados terabytes de dados por rodada. Os avanços na tecnologia de sequenciamento aumentaram a necessidade do desenvolvimento de técnicas para permitir a análise dos dados de forma mais rápida e precisa. Neste trabalho apresentamos otimizações de desempenho de um algoritmo de contabilização de *k-mers* para dados de amostras genômicas e metagenômicas, chamado de "Contabilizador da Frequência de Repetição de *k-mers*"ou **CFRK**.

O algoritmo **CFRK** [Vilasboas et al. 2015] foi desenvolvido para efetuar contabilização da frequência de repetição de *k-mers* de bases de dados de amostras genômicas em ambientes de *GPU's*. O termo *k-mer* é utilizado como referência a todas as possíveis combinações de comprimento k que estão contidas em uma sequência de dados arbitrária. Essa técnica está contida na interseção entre a mineração de dados e a análise combinatória. As aplicações dos *k-mers* são inúmeras. *K-mers* podem ser utilizados para montagem [Garg et al. 2013] de amostras de DNA, RNA ou proteínas, para alinhamento de sequências [Church et al. 2011], ou para análise qualitativa de um sequenciamento [Plaza Onate et al. 2015]. Também podem ser utilizados como entrada para algum método de classificação ou agrupamento [Fiannaca et al. 2015]. Os valores de k utilizados variam de acordo com a análise feita.

A primeira versão do **CFRK** apresentou um ganho de aproximadamente 6 vezes em relação ao **Jellyfish**, considerado o estado da arte para contabilização de *k-mer*, para valores de k menores que 5, porém tinha a limitação de processar apenas arquivos de tamanho menor ou igual à memória principal da *GPU*. A segunda versão do **CFRK** [Vilasboas et al. 2016] implementou uma rotina para dividir os dados de entrada e possibilitou a execução de arquivos de tamanho maior do que a memória principal da *GPU*. Neste trabalho apresentamos a terceira versão do **CFRK**, que introduziu mais três níveis de paralelismo: um primeiro nível para permitir o processamento de parte do algorítimo em múltiplos núcleos do *módulo CPU*, um segundo nível para permitir o processamento de parte do algoritmo em múltiplas *GPU's* e um terceiro nível para permitir a execução de múltiplas instâncias do algoritmo **CFRK** em nós de processamento de uma *cluster*.

Este trabalho analisa o ganho de desempenho do **CFRK** para cada otimização implementada e o ganho de desempenho total do **CFRK** com todas as otimizações. Demonstramos que a adição de mais 3 níveis de paralelismo aumenta significativamente o desempenho do algoritmo.

Este trabalho está dividido da seguinte forma: Seção 2 apresenta os trabalhos relacionados; Seção 3 apresenta a descrição do algoritmo **CFRK** e das otimizações desenvolvidas. Seção 4 apresenta a análise de desempenho. Seção 5 apresenta a descrição do **MCFRK**, resultados e análises. Seção 6 apresenta as conclusões e os trabalhos futuros.

2. Trabalhos Relacionados

O *Jellyfish* [Marçais and Kingsford 2011] é um algoritmo desenvolvido para processamento em memória compartilhada e é considerado o estado da arte entre os algoritmos de contabilização da frequência de repetição de *k-mers*. Ele utiliza várias estruturas *lock-free*, que são estruturas que permitem operações atômicas sem o bloqueio da memória e, por isso, não degradam o desempenho em ambientes multiprocessáveis. O *Jellyfish* apresenta bom desempenho para a contabilização de valores de $10 \leq k \leq 32$ enquanto que o **CFRK** possui melhor desempenho para valores de $k \leq 9$. Por outro lado, o *Jellyfish* foi projetado para processar amostras de genoma, pois ele considera todos os *reads*[1] como sendo pertencente a um mesmo organismo. O **CFRK** foi projetado para permitir a análise de amostras de genomas e de metagenomas, pois considera que cada *read* pode pertencer a um organismo distinto. Isto possibilita uma análise conclusiva para grande classe de pesquisas em bioinformática, conforme o trabalho [Edwards et al. 2012].

O **Gerbil** [Erbert et al. 2017] é um algoritmo para contabilização da frequência de repetição de *k-mers* com suporte a múltiplas *GPU's* e que faz uso do disco para melhorar a eficiência em relação à memória. Sua execução consiste de duas fases: distribuição e contabilização. Este algoritmo é projetado para o processamento de genomas. Isto permite que o *Gerbil* efetue menos transferência de dados entre a *GPU* e o *host* e apresente um bom desempenho para valores de $k \leq 32$. Porém, assim como o *Jellyfish*, o **Gerbil** não pode ser utilizado para pesquisas que necessitam gerar um vetor de frequência para cada *read*.

[1] Fragmentos de material genético resultantes do sequenciamento

3. Descrição do algoritmo

CFRK é acrônimo para "Contabilizador da Frequência de Repetição de *k-mers*", foi desenvolvido para *GPU's* da *NVidia*, utilizando a linguagem *CUDA* e os *drivers* disponibilizados pelo fabricante.

Para o desenvolvimento do **CFRK** utilizamos o conceito de programação modular, onde cada módulo é responsável por uma atividade específica na execução do algoritmo. O **CFRK** é composto por dois módulos principais: o módulo que contém as funções que são executadas pela *CPU*, chamado *módulo CPU*, e o módulo que contém as funções que são executadas pela *GPU*, chamado *módulo GPU*. O *módulo CPU* é composto por dois submódulos: o *main* e o *kmer_main*. O submódulo *main* é responsável pela leitura do arquivo de entrada, pela preparação dos *chunks* para serem enviados as *GPU's* e pela gerência da execução dos outros submódulos. O módulo *kmer_main* é responsável pela preparação da *GPU* para a execução e possui as rotinas de gerenciamento e transferência de memória e chamadas aos *kernels*. O *módulo GPU* é composto pelo submódulo *kmer_kernel*, que contém os *kernels* executadas na *GPU*.

3.1. Descrição das funções do *módulo CPU*

Nesta seção descreveremos o funcionamento dos dois submódulos contidos no *módulo CPU*, o submódulo *main* e o submódulo *kmer_main*.

3.1.1. Submódulo *main*

O arquivo resultante do sequenciamento de uma amostra genômica ou metagenômica contém um grande número de fragmentos, denominadas *reads*. Cada *read* possui uma sequência de nucleotídeos que são representados pelos caracteres A, C, G, T ou N. O caractere N é observado no *read* quando o sequenciador não reconhece com precisão o nucleotídeo pertencente àquela posição. Quando o arquivo é lido, é feita uma codificação para transformar os caracteres A, C, G, T ou N para os valores numéricos 0, 1, 2, 3 ou -1, respectivamente. Os *reads* são dispostos em sequência formando uma estrutura de *array* unidimensional contendo todos os nucleotídeos de todos os *reads*, chamado *vetor de reads*. Para identificar o final de cada *read*, é utilizado o valor -1. O valor -1 é utilizado para representar tipos de dados inválidos, ou seja, o *k-mer* que contém o valor -1 não deverá ser processado. Os arquivos gerados pelos sequenciadores de tecnologia *NGS* podem conter uma grande quantidade de informação e a memória global da *GPU* pode não ser suficiente para armazenar toda essa informação. Por isto o *vetor de reads* é subdividido e armazenado em estruturas chamadas *chunks*. Cada *chunk* possui uma quantidade pré-definida de *reads*, bem como as informações da posição inicial e o tamanho de cada *read*. Posteriormente, os *chunks* são enfileirados e serão enviados um a um para o processamento na *GPU*. Com isso pode-se ajustar a quantidade de dados que será enviada para o processamento na *GPU* e, assim, tem-se o controle da quantidade total de memória utilizada para o processamento. O usuário tem liberdade para definir a quantidade de *reads* por *chunk*, sendo que a quantidade máxima varia de acordo com a disponibilidade da memória da placa que está em utilização. Esta função demanda um grande tempo de processamento devido a manipulação de memória. Esta função pertence ao módulo *main*, no qual estão todas as funções que são executadas exclusivamente pela *CPU*.

3.1.2. Submódulo *kmer_main*

Os *chunks* são enviados para o submódulo *kmer_main* através de uma estrutura que contém as sequências a serem processadas o número de sequências, o tamanho de cada sequência e a posição inicial de cada sequência. É neste módulo de que se encontram todas as funções de gerenciamento de memória, funções para alocação e desalocação dos vetores, funções de transferência de memória entre *host* e *device* e as chamadas aos *kernels*. Este módulo foi alvo do estudo publicado em [Vilasboas et al. 2016].

3.1.3. Otimização no *módulo CPU*

Foi feito um estudo do comportamento das funções executadas pela *CPU* durante a execução do **CFRK**. Foi utilizada a ferramenta *VTune* disponibilizada pela *Intel* em seu pacote *Intel Parallel Studio*. Observamos nestes resultados que a função de leitura de arquivo e a função responsável pela seleção dos *chunks* compunham os *hotspots*. O VTune nos permitiu visualizar as instruções que mais demandaram processamento dentro destes *hotspots* e com base nestas informações foi selecionado um laço de repetição para a paralelização via diretivas do *OpenMP*.

O laço de repetição selecionado foi o que mais demandou processamento na função de seleção dos *chunks*. A operação realizada por este laço de repetição consiste em uma cópia de memória entre dados lidos do arquivo de entrada e a estrutura que armazenará os *chunks* para o processamento. A diretiva inserida foi a *#pragma omp parallel for*. Essa diretiva cria uma região paralela na qual as iterações do laço de repetição são divididas para serem executadas em paralelo pelos núcleos da *CPU*.

3.2. Descrição das funções do *módulo GPU*

O processo de contabilização da frequência de repetição de *k-mers* é dividido em duas fases. A primeira fase é a conversão dos valores da base 4 para a base 10 e a segunda fase é a contabilização da frequência de repetição de *k-mers*.

A primeira fase do processamento consiste em converter os valores numéricos dos *k-mers*, que estão na base 4, para a base 10. Essa conversão é feita para facilitar o mapeamento entre o valor da combinação do *k-mer* e a posição do contador no *vetor de frequência* referente àquela combinação, dado que o valor do *k-mer* na base 10 será exatamente o valor da posição a qual deverá ser feita a contabilização da repetição. O resultado desta conversão é armazenado em um vetor denominado *vetor de conversão*.

Na segunda fase é calculada da frequência de repetição de cada *k-mer*. É alocado um vetor chamado *vetor de frequência*. Este vetor irá armazenar o resultado do cálculo da frequência de repetição. Para cada *read* no *vetor de reads* será necessário alocar um *vetor de frequência* e cada posição do *vetor de frequência* será um contador correspondente ao valor do *k-mer* no *vetor de conversão*. Então o *vetor de conversão* é percorrido e é realizada a operação $+1$ no *vetor de frequência* na posição correspondente ao valor do *k-mer*.

3.2.1. Otimização no *módulo GPU*

A *GPU* foi utilizada para acelerar o processamento da conversão dos valores dos *k-mers* da base 4 para a base 10 e também a contabilização da frequência de repetição dos *k-mers*. Essas funções foram selecionadas para este tipo de abordagem pelo fato de se enquadrarem no paradigma *Single Instruction Multiple Data (SIMD)*, ou seja, vários dados sendo operados por uma instrução. Estas otimizações estão descritas no trabalho [Vilasboas et al. 2016].

Para este trabalho foi feita a alteração do tipo do dado utilizado no *vetor de reads*. Até a segunda versão do **CFRK** o *vetor de reads* era armazenado como *int* (inteiro), que ocupa 4 *bytes* na memória. A proposta foi alterar de *int* para o tipo *char* (caractere), que ocupa 1 *byte* na memória. Essa alteração foi possível porque a linguagem *CUDA* é uma extensão da linguagem *C* e esta usa o tipo do dado apenas como referência ao espaço de alocação em memória permitindo fazer cálculos com variáveis de qualquer tipo primitivo.

Essa alteração foi feita no *kernel ComputeIndex* do submódulo *kmer_kernel* que pertence ao *módulo GPU*.

3.3. Suporte a execução em múltiplas *GPU's*

De forma a possibilitar que a execução do **CFRK** em múltiplas *GPU's*, foi desenvolvida uma função no submódulo *main* que reconhece o número de *GPU's* disponíveis através da função *cudaGetDeviceCount* da biblioteca padrão do *CUDA* e utiliza a biblioteca *pthreds* para criar *threads* do módulo *kmer_main* para serem executadas em paralelo em cada *GPU*. A função desenvolvida, após gerar as *threads*, irá designar um grupo de *chunks* para serem processados em cada *thread* e a seguir enviar as *threads* para serem executadas em cada *GPU*. Após a execução os vetores de frequência gerados por cada *GPU* são coletados pelo módulo *main* e armazenados no arquivo de saída e as *threads* são destruídas.

4. Resultados e análises

Nesta seção apresentaremos os resultados referentes as otimizações implementadas nos dois módulos principais e seus respectivos resultados.

Para os experimentos foram utilizando os nós computacionais *GPU's* do supercomputador SDumont, onde cada nó possui a configuração: 2 x *CPU Intel Xeon E5-2695v2 Ivy Bridge, 2,4GHZ*, 24 núcleos (12 por *CPU*), totalizando de 1.296 núcleos, *64GB DDR3 RAM*, 2 x *Nvidia K40* (dispositivo *GPU*), sistema operacional *Red Hat Enterprise Linux Server release 6.4*, *kernel* versão *2.6.32-504.12.2.el6.x86_64*, *CUDA 8.0*, *Intel Parallel Studio XE 2016*, compilador *icpc* com flags as *flags -qopenmp e -Ofast*, *VTune* e *OpenMP 4.5*. Este supercomputador está alocado no Laboratório Nacional de Computação Científica (LNCC) e para este trabalho foram utilizados os nós *B715* com placas aceleradoras *GPU Nvidia K40*.

O arquivo de entrada utilizado foi uma amostra real de metagenoma com 8.7GB de dados cuja identificação no banco de dados *SRA* do *NCBI* (*National Center for Biotechnology Information*) é *SRX2021688*. Este arquivo foi escolhido por representar um típico arquivo de tamanho médio utilizado para pesquisas de metagenoma.

4.1. Resultado das otimizações no *módulo CPU*

Utilizamos o *VTune* para gerar o perfil de execução do **CFRK** paralelizado com a diretiva *OpenMP*. Dado que o valor de k não é utilizado pelas rotinas dos submódulos da *CPU*, os resultados apresentados são relativos a um valor fixo de k. Escolhemos o valor $k = 4$ que é utilizado para análise de genoma conforme o trabalho [Edwards et al. 2012]. Denominamos a versão do **CFRK** que possui os submódulos do *módulo CPU* paralelizados através do *OpenMP* de **CFRK_OMP**. A Tabela 1 apresenta na primeira coluna o nome das funções das duas versões do algoritmo **CFRK** sendo a primeira linha referente a versão sem a diretiva *OpenMP*, **CFRK**, e a segunda linha referente a versão com a diretiva *OpenMP*, **CFRK_OMP**. A segunda coluna apresenta o tempo de execução em segundos de cada uma das funções. A terceira coluna apresenta o ganho em vezes da função com a diretiva *OpenMP* em relação a mesma função sem a diretiva. Nossos expe-

Função	Tempo (segundos)	Ganho (vezes)
SelectChunk (CFRK)	10. 427	—
SelectChunk (CFRK_OMP)	0.303	34.41

Tabela 1: Resultado obtido através do VTune para o CFRK e CFRK_OMP

rimentos demonstram que a paralelização ao nível dos múltiplos núcleos da *CPU* através da inserção de diretivas do *OpenMP* em um laço de repetição da função de cópia de dados do arquivo de entrada para uma estrutura de processamento gerou um ganho de 34 vezes para 12 núcleos em relação a execução com apenas um núcleo. Este ganho mostra que o ambiente *multicore*, além de aumentar a taxa de execução em paralela do algoritmo também ajuda a diminuir o tempo de acesso aos dados na memória *cache*, gerando um ganho "supralinear"para a função *SelectChunk*.

4.2. Resultado das otimizações no *módulo GPU*

O submódulo *kmer_kernel*, que executa na *GPU*, possui o *kernel ComputeIndex* que executa uma conversão dos valores dos *k-mers* da base 4 para a base 10 antes de realizar a contabilização da frequência de repetição dos *k-mers*. De forma a diminuir o espaço ocupado pelos dados na *cache* da *GPU*, realizamos a alteração do tipo do dado utilizado no *vetor de reads*. Originalmente este vetor era armazenado como *int* (inteiro), ocupando 4 *bytes* por elemento na memória. Este foi alterado para o tipo *char* (caractere), que ocupa 1 *byte* por elemento na memória. Essa alteração foi possível porque a linguagem *CUDA* é uma extensão da linguagem *C* e esta usa o tipo do dado apenas como referência ao espaço de alocação em memória permitindo fazer cálculos com variáveis de qualquer tipo primitivo.

Os resultados apresentados nesta seção são da execução do *kernel ComputeIndex* e foram obtidos através do *nvprof*. As Tabelas 2, 3, 4 e 5 apresentam os resultados da execução do *kernel ComputeIndex* gerados pelo perfilador *nvprof* para os valores de $k = \{2, 3, 4\}$ respectivamente. A primeira coluna apresenta o nome das versões do **CFRK** onde **CFRK_INT** se refere a versão original com o *vetor de reads* armazenado como inteiro e **CFRK_CHAR** se refere a nova versão com o *vetor de reads* armazenado como caractere. A segunda coluna apresenta a Ocupação que é a relação entre a média de *warps* ativos por ciclo e a quantidade máxima de *warps* suportada pela placa. A terceira coluna

apresenta o IPC (*Instructions per cycle*) que é a quantidade de operações executadas por ciclo. A quarta coluna apresenta o valor médio de acessos repetidos à memória devido ao *cache miss*. A quinta coluna apresenta as Instruções por *warp* que é a média de instruções executadas por *warp*. A sexta coluna apresenta *Warps* por ciclo que é a quantidade média de *warps* ativos por ciclo. Ao observar as Tabelas 2, 3, 4 e 5 notamos uma grande

Versão	Ocupação	IPC	Cache miss	Inst. por Warp	Warps por ciclo
CFRK_INT	87.64%	2.80	0.00599	82.92	7.11
CFRK_CHAR	92.26%	4.07	0.00122	202.89	15.79

Tabela 2: Dados do perfilador para $k = 2$

Versão	Ocupação	IPC	Cache miss	Inst. por Warp	Warps por ciclo
CFRK_INT	89.30%	2.80	0.00956	103.90	6.68
CFRK_CHAR	93.96%	4.07	0.00175	282.84	17.72

Tabela 3: Dados do perfilador para $k = 3$

Versão	Ocupação	IPC	Cache miss	Inst. por Warp	Warps por ciclo
CFRK_INT	91.06%	2.77	0.01193	124.87	6.27
CFRK_CHAR	94.87%	3.94	0.00205	362.78	18.03

Tabela 4: Dados do perfilador para $k = 4$

Versão	Ocupação	IPC	Cache miss	Inst. por Warp	Warps por ciclo
CFRK_INT	91.58%	2.80	0.01362	145.85	7.55
CFRK_CHAR	94.55%	4.03	0.00224	442.73	17.93

Tabela 5: Dados do perfilador para $k = 5$

diminuição da taxa de *Cache miss* e um consequente aumento do respectivo IPC da versão anterior, **CFRK_INT**, em relação a versão atual, **CFRK_CHAR**. Note que na versão **CFRK_INT** o valor de *cache miss* aumenta com proporção média de 3×10^{-3} e a versão **CFRK_CHAR** aumenta com proporção média de 3×10^{-4}. Por consequência deste fato todas as outras taxas aumentam. A Ocupação aumentou 4% em média, o valor do IPC teve um aumento de aproximadamente 1.4 vezes, o número de Instruções por Warp teve um aumento de 3 vezes em média, o número de Warps por ciclo teve um aumento de 2.5 vezes em média.

Isto se deve pela diminuição da ocupação da memória. Ao trocar o tipo de dado inteiro, 4 *bytes*, para caractere, 1 *byte*, foi possível alocar 4 vezes mais dados nas memórias *cache* da *GPU*. Isso fez com que as requisições à memória global diminuíssem, aumentando a performance deste *kernel*.

4.3. Avaliação de desempenho do CFRK com 3 níveis de paralelismo

A Figura 1 apresenta o tempo de execução do **CFRK** da versão sem as otimizações apresentadas neste trabalho, identificada no gráfico como *orig*, em relação a versão com todas as otimizações apresentadas neste trabalho, identificada no gráfico como **CFRK_OMP_MG**. O eixo x apresenta o tempo de execução em segundos e o eixo y apresenta os valores de k. A Tabela 6 apresenta a redução do tempo obtido pelos aprimo-

Figura 1: Tempo de execução do CFRK e do CFRK_OMP_MG

ramentos feitos na versão inicial do **CFRK**. A primeira linha apresenta os valores de k, a segunda linha apresenta o tempo em segundos e a terceira linha apresenta o ganho em porcentagem. Observando a Figura 1 e a Tabela 6 vemos que o tempo de processamento

k	2	3	4	5
Tempo (segundos)	90.19	81.16	84.06	88.53
Ganho (%)	39.92	35.40	34.99	29.58

Tabela 6: Tempo de processamento reduzido entre o CFRK_OMP_MG em relação ao CFRK

foi reduzido em mais de 80 segundos em todos os casos. A maior redução no tempo foi de 39.92% para k = 2, que executa menos operações de transferência de dados entre a *GPU* e o *host*. Conforme análise apresentada em trabalhos anteriores [Vilasboas et al. 2016], a medida que o valor de k aumenta, a complexidade do tempo de execução do *kernel* de contabilização de frequência de *k-mer* aumenta. Por outro lado, a medida que aumenta o valor de k aumenta o tamanho do vetor de contabilização de *k-mers* e consequentemente o tempo com a transferência de dados entre a *GPU* e o *host*, justificando a diminuição do ganho.

5. MCFRK

O **MCFRK** [Coelho et al. 2016] foi desenvolvido para possibilitar a execução de forma eficiente do **CFRK** com arquivos que possuem tamanho maior ou igual a memoria principal da máquina em execução. **MCFRK** é o acrônimo de **M**PI - **C**ontabilizador da

Frequência de **R**epetição de **K**-mers. Este algoritmo é uma extensão do **CFRK** para permitir a execução em ambientes de memoria distribuída, ou *cluster*, composta por diversos nós computacionais com placas aceleradoras *GPU's*. Para isso, foi utilizado o padrão de troca de mensagens *MPI*. A versão do **MCFRK** apresentada nesta seção integra todas as otimizações relatadas nas seções anteriores.

O **MCFRK** decompõe o arquivo original em arquivos menores para serem processados nos múltiplos nós de processamento da *cluster*. Foram adicionadas ao submódulo *main* do *módulo CPU* funções da biblioteca *MPI* para enviar, criar e gerenciar a execução de processos **CFRK** nos nós de processamento. A execução do **MCFRK** é dividida em duas fases: o pré-processamento e o processamento. Na fase de pré-processamento é realizada a divisão do arquivo. Esta divisão é realizada através de uma aplicação desenvolvida na linguagem C, chamada de split, que realiza o particionamento do arquivo em arquivos menores com o mesmo número de *reads*. Na fase de processamento e após a inicialização dos processos *MPI*, é fornecido a localização dos respectivos arquivos de entrada e de saída. Cada processo *MPI* realiza a execução de uma instância do **CFRK** e ao final do processamento os resultados do arquivo de saída são armazenados em disco.

5.1. Resultados e análise

Primeiramente vamos apresentar o resultado de cada uma das otimizações separadamente e posteriormente apresentar uma análise mais detalhada do **MCFRK**.

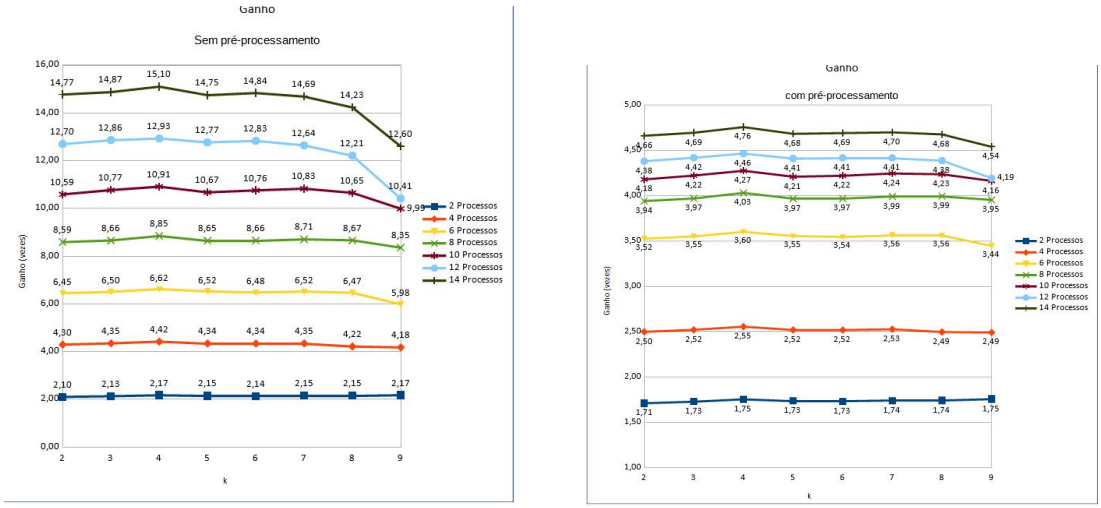
Para os experimentos apresentados na Tabela 7 foram utilizados 8 nós do supercomputador SDumont, onde cada nó possui as configurações apresentadas na Seção 4. O arquivo de entrada utilizado foi uma amostra real de metagenoma com 8.7GB de dados cuja identificação no banco de dados *SRA* do *NCBI* (*National Center for Biotechnology Information*) é *SRX2021688*. A Tabela 7 apresenta o tempo de execução e o ganho para cada otimização aplicada ao **MCFRK**. A primeira coluna apresenta a descrição da otimização; a segunda coluna apresenta os tempos de execução de cada versão com 1 processo *MPI*; a terceira coluna apresenta os tempos de execução de cada versão com 8 processos *MPI* sem a contabilização da operação de *split*; a quarta coluna apresenta os tempos de execução de cada versão com 8 processos *MPI* com a contabilização da operação de *split*; a quinta coluna apresenta o ganho de performance de cada versão sem a contabilização da operação de *split*; a sexta coluna apresenta o ganho de performance de cada versão sem a contabilização da operação de *split*.

Versão	1 processo	8 processos sem *split*	8 processos com *split*	Ganho sem *split*	Ganho com *split*
Sem otimização	160.39	20.4	70.545	7.86	2.27
Otimização módulo CPU	151.18	19.22	69.365	7.87	2.18
Otimização módulo GPU	155.97	19.99	70.135	7.80	2.22
Todas as otimizações	144.64	18.07	68.215	8,00	2.12

Tabela 7: Tempo de execução e ganho de cada otimização aplicada ao MCFRK

Para os experimentos apresentados nas Figuras 2a e 2b foram utilizados 14 nós do supercomputador SDumont, onde cada nó possui as configurações apresentadas na Seção 4. O arquivo de entrada utilizado foi uma amostra real de metagenoma com 15GB de dados cuja identificação no banco de dados *SRA* do *NCBI* (*National Center for Biotechnology Information*) é *SRX1142487*.

A Figura 2a apresenta o ganho obtido pela execução do **MCFRK** com múltiplos processos em relação a execução dele próprio com um processo sem incluir o tempo da fase de pré-processamento. A linha em azul escuro apresenta o ganho com 2 processos, a linha em laranja apresenta o ganho com 4 processos, a linha em amarelo apresenta o ganho com 6 processos, a linha em verde apresenta o ganho com 8 processos, a linha em vinho apresenta o ganho com 10 processos, a linha em azul claro apresenta o ganho com 12 processos, a linha em verde escuro apresenta o ganho com 14 processos. A Figura 2b apresenta o ganho obtido pela execução do **MCFRK** com múltiplos processos em relação a execução dele próprio com um processo incluindo o tempo de pré-processamento. A linha em azul escuro apresenta o ganho com 2 processos, a linha em laranja apresenta o ganho com 4 processos, a linha em amarelo apresenta o ganho com 6 processos, a linha em verde apresenta o ganho com 8 processos, a linha em vinho apresenta o ganho com 10 processos, a linha em azul claro apresenta o ganho com 12 processos, a linha em verde escuro apresenta o ganho com 14 processos. A diferença entre os ganhos de desempenho nas



(a) Ganho do **MCFRK** sem o pré-processamento

(b) Ganho do **MCFRK** com o pré-processamento

Figura 2: Ganho do MCFRK

Figuras 2a e 2b, sem e com a operação split, mostram o impacto no desempenho do sistema causado pelas operações em disco. Conforme explicado anteriormente, a operação split é executada na fase de pré-processamento onde é feito o particionamento do arquivo de entrada em arquivos menores para serem enviados para os nós de processamento da *cluster*. Isto constata o gargalo existente nos algoritmos que realizam muitas operações de leitura e escrita em disco.

Ao compararmos as Figuras 2a e 2b podemos observar que a medida em que aumentamos o número de processos, aumentamos o grau de execução em paralelo do algoritmo ao mesmo tempo em que diminuímos o tamanho do arquivo a ser processado em cada nó. Isto gera a diminuição no tempo de processamento por nó apresentada na figura 2a, porém o tempo de execução da operação split permanece o mesmo, aumentado a proporção do tempo de execução da operação split em relação as outras operações o que explica a diminuição do speed-up apresentada na figura 2b.

Demostramos que o **CFRK**, ao acrescentar o terceiro nível de paralelismo através da execução em uma *cluster* híbrida composta por nós de processamento com duas *GPU's* cada, é capaz de apresentar um ganho "supralinear"para a maioria dos valores de k. O mesmo experimento mostra que ao contabilizar os gastos com o pré-processamento, a otimização em três níveis diminui e passa para um ganho de até 3.84 vezes com 14 nós de processamento, conforme a Figura 2b. Observamos que a medida em que aumentamos os nós de processamento, aumentamos o grau de execução em paralelo do algoritmo ao mesmo tempo em que diminuímos o tamanho do arquivo a ser processado em cada nó, ou seja, diminuímos a taxa de "cache miss"gerando um ganho "supralinear"no algoritmo **MCFRK**, conforme a Figura 2a. Por outro lado o tempo de execução da operação split se manteve constante, aumentado a proporção do tempo de execução da operação split em relação as outras operações e causando a diminuição no ganho total, conforme a Figura 2b.

Por final, podemos observar que o ganho de desempenho obtido com as otimizações implementadas neste trabalho possibilitou o aumento na eficiência do algoritmo **MCFRK** que passou a apresentar bom desempenho para valores de k até 9, conforme podemos observar nas Figuras 2a e 2b.

6. Conclusão e trabalhos futuros

Neste trabalho apresentamos otimizações desenvolvidas no algoritmo **CFRK**, originalmente desenvolvido para implementar paralelismo ao nível de *threads* em uma *GPU*, para permitir a execução com mais 3 níveis de paralelismo. Em um primeiro nível, utilizamos as diretivas do padrão *OpenMP* para permitir o processamento de módulos do algoritmo em múltiplos núcleos da *CPU*. Em um segundo nível, utilizamos as bibliotecas do modelo de programação *pthread* para permitir o processamento de mais de uma instância do *módulo GPU* em múltiplas *GPU's*. Por final, em um terceiro nível, utilizamos as bibliotecas do modelo de programação *MPI* para permitir a execução de múltiplos processos do algoritmo **CFRK** em nós de processamento de uma *cluster*.

Nossos experimentos demonstram que a paralelização ao nível dos múltiplos núcleos da *CPU*, para um nó de 12 núcleos, gerou um ganho de 34 vezes em relação a execução com apenas um núcleo. Este ganho mostra que a paralelização no ambiente *multicore*, além de aumentar a taxa de execução em paralela do algoritmo, diminui o tempo de acesso aos dados na memória, gerando um ganho "supralinear"para a função *SelectChunk*. Demonstramos também que ao acrescentarmos o segundo nível de paralelismo através da execução do algoritmo CFRK em duas *GPU's*, o tempo de execução diminuiu de 39% ao invés de 50% devido a latência de transferência de dados entre a *GPU* e a *CPU*. Demostramos que o **CFRK**, ao acrescentar o terceiro nível de paralelismo através da execução em uma *cluster* híbrida composta por nós de processamento com duas *GPU's* cada, também apresenta um ganho "supralinear"para a maioria dos valores de k devido ao aumento do grau de execução em paralelo do algoritmos e à diminuição do tamanho do arquivo a ser processado em cada nó e consequente diminuição da taxa de "cache miss". Por final, o experimento mostra que ao contabilizar os gastos com o pré-processamento, a otimização em três níveis tem um ganho muito menor, em até 4.78 vezes para 14 nós. Isto se dá porque apesar de aumentarmos o grau de execução em paralelo e diminuirmos a taxa de "cache miss", à medida em que aumentamos o número de nós de processamento, a latência da operação de transferência de dados se mantém cons-

tante, aumentando proporcionalmente o tempo de execução em transferência de dados em relação ao tempo de execução de processamento e causando uma diminuição no ganho total, conforme apresentado na Figura 2b.

Como trabalho futuro pretendemos desenvolver otimizações no **CFRK** de forma a diminuir o gargalo gerado pelas transferências de dados de entrada e saída. Pretendemos também avaliar o desempenho do **CFRK** em arquiteturas que apresentam um canal de alto desempenho para transferência de dados entre o host e a GPU tais como o nvlink da NVIDIA. Como trabalhos futuros também pretendemos acoplar o algoritmo **CFRK** a ferramentas para a análise de amostras de material genético para ser utilizado pelos usuários do portal de bioinformática do LNCC.

Referências

Church, P. C., Goscinski, A., Holt, K., Inouye, M., Ghoting, A., Makarychev, K., and Reumann, M. (2011). Design of multiple sequence alignment algorithms on parallel, distributed memory supercomputers. In *2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 924–927. IEEE.

Coelho, M., Vilasboas, F., and Osthoff, C. (2016). Desenvolvimento de uma versão paralela híbrida para a contabilização da frequência de repetição de k-mers. *ERAD-RJ Escola Regional de Alto Desempenho do Rio de Janeiro*.

Edwards, R. A., Olson, R., Disz, T., Pusch, G. D., Vonstein, V., Stevens, R., and Overbeek, R. (2012). Real time metagenomics: using k-mers to annotate metagenomes. *Bioinformatics*, 28(24):3316–3317.

Erbert, M., Rechner, S., and Müller-Hannemann, M. (2017). Gerbil: a fast and memory-efficient k-mer counter with gpu-support. *Algorithms for Molecular Biology*, 12(1):9.

Fiannaca, A., La Rosa, M., Rizzo, R., and Urso, A. (2015). A k-mer-based barcode DNA classification methodology based on spectral representation and a neural gas network. *Artificial intelligence in medicine*, 64(3):173–84.

Garg, A., Jain, A., and Paul, K. (2013). GGAKE: GPU Based Genome Assembly Using K-Mer Extension. In *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pages 1105–1112. IEEE.

Marçais, G. and Kingsford, C. (2011). A fast, lock-free approach for efficient parallel counting ofoccurrences of k-mers. *Bioinformatics (Oxford, England)*, 27(6):764–70.

Plaza Onate, F., Batto, J.-M., Juste, C., Fadlallah, J., Fougeroux, C., Gouas, D., Pons, N., Kennedy, S., Levenez, F., Dore, J., Ehrlich, S. D., Gorochov, G., and Larsen, M. (2015). Quality control of microbiota metagenomics by k-mer analysis. *BMC genomics*, 16(1):183.

Vilasboas, F., Osthoff, C., Trelles, O., and Vasconcelos, A. T. (2015). Desenvolvimento de um algoritmo paralelo para contabilização da repetição de k-mers. *3ª Conferência Ibero Americana de Computação Aplicada 2015*.

Vilasboas, F., Osthoff, C., Trelles, O., and Vasconcelos, A. T. (2016). Otimização de um algoritmo paralelo para contabilização da repetição de k-mers. *II Escola Regional de Computação de Alto Desempenho do Rio de Janeiro*.