Reescrita de código e utilização de Paralelismo de Dados para a busca de Small RNAs (sRNAs)

[1]André G D Almeida, [1]Edison L S Machado, [3]Sergio Y Fujii, [1]Julia Pacholock, [2]Maria B R Steffens, [1]Lucas F Oliveira

[1]Setor de Educação Profissional e Tecnológica – Universidade Federal do Paraná (UFPR) – Curitiba – PR – Brasil

[2]Setor de Bioquímica e Biologia Molecular – Universidade Federal do Paraná (UFPR), Caixa Postal 19046 – CEP 81531-990 – Curitiba – PR – Brasil

[3]Departamento de Informática – Universidade Federal do Paraná (UFPR), Caixa Postal 19801 – CEP 81531-980 – Curitiba – PR – Brasil

gehringandre@gmail.com, edisonluizsm@gmail.com, sfujii@gmail.com, juujiepvs@gmail.com, steffens@ufpr.br, lferrari@ufpr.br

Abstract. *Today computers have increasingly more processing power, allowing older programs have to be readjusted in order to make better use of computer resources. This paper presents an refactoring from sRNAScanner that performs studies areas intergenic sRNAs in bacterial genomes. To refactoring the code, high-performance libraries and manual code tunning are use. The results of the execution times getting above the expected 156.8 times faster in the best case and 108.7 times in the worst case. These results encourage future adaptation of the code to run on GPUs.*

Resumo. *Atualmente os computadores possuem cada vez mais poder de processamento, permitindo que programas antigos tenham que ser readaptados, afim de fazer melhor uso dos recursos do computador. Neste artigo apresentamos a otimização de uma ferramenta que realiza estudos de áreas intergênicas de sRNAs em genomas bacterianos. Para a reescrita do código, foram utilizadas bibliotecas de alto desempenho e otimização manual do código. Os resultados dos tempos de execução superou o esperado ficando 156,8 vezes mais rápido no melhor caso e 108,7 vezes no pior caso. Esses resultados encorajam uma futura adaptação do código para ser executado em GPUs.*

1. Introdução

O poder de processamento dos computadores atuais está cada vez maior. Com a mudança de paradigma na indústria de processadores é comum encontrar computadores com mais de um núcleo e muita memória disponível, algo impensável até alguns anos atrás. A industria dos computadores foi muito afetada com a recente descoberta de que a velocidade do clock não poderia mais aumentar. A solução encontrada foi partir para as arquiteturas multi-core, onde diversos processadores se comunicam uns com os outros

em um ou vários núcleos. A arquitetura multi-core foi a solução e hoje em dia está presente na maioria dos processadores [Schmidt 2010].

Antigamente o custo e limitação da memória RAM forçavam os softwares fazerem muito uso do disco rígido. Hoje, porém, esse tipo de estratégia se tornou o maior entrave de qualquer aplicação, pois o disco não possui velocidade compatível com os demais componentes. Dessa maneira, as aplicações que necessitam de muito poder de processamento devem ser otimizadas para aproveitar o hardware ao máximo [M. Cunha, A. Coutinho, and J. Telles (2001)].

A programação paralela ajuda a desenvolver aplicações usando o hardware disponível de forma mais eficiente. O princípio do multi-threading envolve a criação de vária threads, ou seja, vários fluxos de execução simultâneos [B. Nichols, D. Buttlar, J. P. Farrell, (1996)]. Nesse caso, parte do código é dividido entre várias threads ou dos dados são divididos e processados pelas threads. Existem alguns recursos para controlar o início, fim e o sincronismo das mesmas, evitando que threads diferentes tentem modificar o mesmo valor ao mesmo tempo [B. Nichols, D. Buttlar, J. P. Farrell, (1996)].

O software otimizado neste trabalho foi o sRNAScanner [Sridhar 2010]. O código original do software, escrito em C++ estruturado, utiliza cinco arquivos de entrada: um arquivo com o genoma bacteriano, outro arquivo que contêm as informações das proteínas contidas no genoma e três arquivos, cada uma com uma matriz base de probabilidade, usado para encontrar as sequencias de sRNAs dentro do genoma. O Software utiliza muitos arquivos temporários durante o processamento, quarenta e quatro no total, o que acarreta um desempenho ruim e um tempo elevado de execução e, atrelado a isso, a conversão dos números lidos como caracteres para valores inteiros que precisam ser analisados pelo resto do programa. Alguns laços de repetição que faziam excessos de comparações e uma grande quantidade de cálculos complexos para o processador. O programa possui quatro arquivos de saída, contendo as informações dos sRNAs encontrados e suas respectivas sequências ([Shridar 2010] e [Shridar 2013]).

2. Materiais e Métodos

Visando melhorar o desempenho do programa sRNAScanner utilizando da melhor maneira os recursos disponíveis o processo de reescrita do código fonte foi efetuado em três etapas distintas.

A primeira etapa teve foco em uma análise parcial do código, para compreender o funcionamento do programa e o seu papel na área de biologia, além de identificar funções que pudessem ser paralelizadas no futuro. Uma das dificuldades ocorreu logo no começo, durante a análise do código. Muitas variáveis eram globais, utilizadas por várias funções, dificultando a identificação das suas reais necessidades, além de não possuírem nomes de fácil identificação e/ou interpretação. A verificação das funções mais custosas computacionalmente foi feita com a utilização do programa *gprof* que fornece informações sobre o percentual de tempo gasto pelas funções. A figura 1 mostra um exemplo de saída do *gprof* para o software sRNAScanner original. Neste caso podemos perceber que a função ***another()*** ocupa 94,64% do tempo de processamento e foi uma das primeiras a ser alterada.

Figura 1. Resultado apresentado pelo programa *gprof* para o desempenho do software original

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
94.64          54.25    54.25   132992   407.90   407.90  another(int, float, int, int, char)
 2.77          55.84     1.59                             slide_m3(void*)
 1.03    56.43     0.59    18473    31.94    31.94  fun(int, ..., char, float, float)
 0.84    56.91     0.48     4314   111.28   111.28  range123(char)
 0.31    57.09     0.18                             slide_m1(void*)
 0.21    57.21     0.12                             slide_m2(void*)
 0.12    57.28     0.07                             p_seq()
 0.02    57.29     0.01                             linear_genome(char*)
 0.02    57.30     0.01                             unique(char, char*, int, float)
 0.02    57.31     0.01                             some123(char)
 0.02    57.32     0.01                             masking1(char)
 0.02    57.33     0.01                             diff_c_cc(char, int, int)
 0.00    57.33     0.00        1     0.00     0.00  _GLOBAL__sub_I_m1
 0.00    57.33     0.00        1     0.00     0.00  log_score_m1(char*)
 0.00    57.33     0.00        1     0.00     0.00  log_score_m2(char*)
 0.00    57.33     0.00        1     0.00     0.00  log_score_m3(char*)
```

Na segunda parte do trabalho foi desenvolvida a reescrita do código na sua primeira versão. De acordo com a hierarquia de memória, foram substituídos os arquivos textos temporários originais por estruturas de dados, como, por exemplo, listas encadeadas e vetores. Foram utilizadas estruturas de dados (*vector e linked list*) pertencentes a biblioteca STL (*Standard Template Library*) da linguagem C++. Durante a reescrita, foram paralelizadas todas as funções que foram identificadas como prováveis candidatas na primeira etapa e foi utilizada a biblioteca *pthreads*. As funções que foram paralelizadas são as responsáveis por ler os cinco arquivos de entrada do programa, escrever os quatro arquivos de saída e as funções da região crítica do programa, as que mais demoram para serem executadas.

Para cada caso foi feito um estudo baseado em testes a partir do qual se determinou se era vantajoso ou não o uso de *threads* naquele determinado trecho do código. O processo de reescrita foi desenvolvido função por função, sempre comparando os resultados obtidos com os do software original. Os resultados biológicos obtidos pela primeira versão reescrita, foram os mesmos obtidos pelo programa original.

Na terceira etapa foi feita uma análise mais minuciosa do código e uma segunda versão da reescrita foi desenvolvida. Nesta versão foram reescritos laços de repetição que tinham excesso de comparações, e a simplificação de cálculos complexos para o processador, que em grande quantidade, diminuem significativamente o tempo de execução. Nessa parte mais delicada do projeto foram detectados erros no programa original. Por exemplo, durante a leitura dos arquivos de entrada, em todas as iterações do loop, as informações eram colocadas em um vetor temporário, essas informações eram convertidas e atribuídas a outras variáveis permanentes, porém as variáveis temporárias não eram zeradas a cada iteração. Como resultado ocorria um vazamento de dados nas iterações subsequentes do *loop*, pois a cada iteração do *loop*, as variáveis temporárias continham informações das iterações anteriores. A figura 2 mostra parte da fução do programa original, onde ocorre o vazamento de dados do vetor.

Para corrigir esses erros, a cada iteração dos *loops*, as variáveis temporárias foram zeradas antes de serem utilizadas novamente. Após corrigido esses erros de vazamento de dados, os resultados do programa mudaram significativamente. Além dos resultados biológicos que a primeira versão reescrita apresentava, o programa passou a

apresentar muito mais resultados, que antes não eram encontrados. A medida que a reescrita era feita, testes eram realizados para garantir que os resultados estavam corretos.

```
while((fp.get(ch))&&(ch!=EOF))
        {
                while(ch!='.')
                        { //Lê o primeiro número e copia o num para o vetor temporário a.
                                a[j++]=ch;
                                fp.get(ch);
                        }
                x=atoi(a); //Converte o número do vetor a para int e atribui à variável permanente x
                j=0;
                fp.get(ch);//Pula dois caracteres do arquivo
                fp.get(ch);
                while(ch!='\t')
                        { //Lê o segundo número e copia para o vetor temporário a
                                a[j++]=ch;
                                fp.get(ch);
                        }
                y=atoi(a); //Converte o número do vetor a para int e atribui à variável permanente y
                j=0;
                fp.get(ch);
                while(ch!='\n')
                        {//Vai até o final da linha
                                fp.get(ch);
                        }
                f1 << x << "\n" << y << "\n"; //salva os valores de x e y no arquivo temp/ptt1.txt
                f1 << flush;
        }
```

Figura 2. Parte da função onde ocorre o vazamento de dados do vetor.

3. Resultados

Para calcular o tempo de execução dos programas desenvolvidos e do original, duas chamadas para a função *time()* foram inseridas no código. Uma no início e outra no final da execução. Ao final da execução do programa o tempo em segundos é impresso na tela. O ganho foi calculado utilizando a divisão do tempo de execução do programa original pelo tempo de execução das versões reescritas do código. A estação de trabalho utilizada nos testes possui as seguintes configurações: processador AMD Phenom II x6 3,2GHz 1090T Black Edition; 16GB de memória RAM DDR3; Sistema operacional Debian 6.0 GNU/Linux codinome Wheezy.

Assim, com todas as modificações mostradas na seção 2, a execução do programa, utilizando o genoma da bactéria Chlamydia trachomatis D-UW-3-Cx/ AE001273, ficou 28,88 vezes mais rápido na versão 1.0, e 156,852 vezes mais rápido na versão 2.0. A Tabela 1 mostra os resultados de outros três genomas com seus tamanhos em megabytes, seus respectivos tempos em segundos e os ganhos obtidos para cada genoma em cada versão. Nestes resultados foram utilizadas 6 *threads* para os testes, pois a máquina utilizada possui seis núcleos de processamento.

Tabela 1. Resultados dos tempos gastos em segundos para processar os genomas testados

Genomas/Tamanho	Original	Reescrita versão 1.0		Reescrita versão 2.0	
	Tempo	Tempo	Ganho	Tempo	Ganho
Chlamydia trachomatis D-UW-3-Cx/AE001273 (1.1 MB)	319,508	11,063	28,880	2,037	156,852
Streptococcus pneumoniae TIGR4/AE005672 (2.1 MB)	1.540,131	67,233	22,907	11,775	130,796
Helicobacter Pylori 26695/AE000511 (1.7 MB)	1.604,028	56,207	28,537	9,945	161,289
Clostridium tetani E88/AE015927 (2.8 MB)	7.956,030	338,257	23,520	60,118	132,340
Escherichia coli K12-MG1655/U00096 (4.5 MB)	2.060,265	107,106	19,235	18,855	109,268
Salmonella typhimurium LT2/AE006468 (4.7 MB)	2.225,475	120,123	18,526	21,185	105,049
Bacillus anthracis Ames AE016879 (5.1 MB)	11.348,865	575,457	19,721	104,347	108,760

Na tabela 2 é mostrado a quantidade de sRNAs encontrados pelo programa original e pela nossa versão 2.0, visto que a versão 1.0 encontra a mesma quantidade do programa original. Além da quantidade de genomas encontrados a tabela mostra o tempo em segundos necessários para encontrar cada um dos sRNAs pelos dois programas.

Tabela 2. Resultados da quantidade de sRNAs encontrados e a quantidade de tempo em segundos para encontrar cada um deles.

Genomas/Tamanho	Original		Reescrita versão 2.0	
	sRNAs	Tempo (s)	sRNAs	Tempo (s)
Chlamydia trachomatis D-UW-3-Cx/AE001273 (1.1 MB)	10	31,950	153	0,013
Streptococcus pneumoniae TIGR4/AE005672 (2.1 MB)	100	15,401	579	0,020

Helicobacter Pylori 26695/AE000511 (1.7 MB)	35	45,829	737	0,013
Clostridium tetani E88/AE015927 (2.8 MB)	163	48,810	1801	0,033
Escherichia coli K12-MG1655/U00096 (4.5 MB)	82	25,125	351	0,053
Salmonella typhimurium LT2/AE006468 (4.7 MB)	74	30,073	428	0,049
Bacillus anthracis Ames AE016879 (5.1 MB)	255	44,505	1872	0,055

4. Discussão

Analisando a tabela 1, podemos perceber a diferença no tempo de execução das três versões. É possível perceber que o tamanho do genoma interfere na velocidade do programa. O maior entrave do código original era a leitura/escrita em arquivos e a conversão de tipos de dados, para cada caractere de todos arquivos era feita conversão, tanto na leitura como na escrita do arquivo. No caso de teste da bacteria *Bacillus anthracis Ames AE016879*, todos os arquivos, que incluem os de entrada , saída e os temporários, chegam a ocupar 100 MB no disco, de acordo com o comando linux *du -h* Outros problemas de laços e comparações não ajudavam na rápida execução do programa. Os resultados da tabela 1 mostram que escrever o código corretamente levando em conta a utilização adequada da hierarquia de memória do computador resulta em tempos de execução menores. A utilização de bibliotecas como a STL e a *pthreads* auxiliaram na melhora do tempo de execução sem a necessidade de implementações mais complexas.

Alguns outros erros que prejudicavam os resultados biológicos foram encontrados e corrigidos durante o processo, como, por exemplo, um vazamento de dados armazenados em um vetor e um erro na leitura de um arquivo que gerava uma iteração extra e desnecessária de um loop. O erro de vazamento de dados acabava se propagando e comprometendo os resultados do programa.

A tabela 2 mostra a quantidade de genomas encontrados após a correção dos código. Pode-se perceber que o número de sRNAs encontrados é o no mínimo 7 vezes e no máximo 21 vezes maior do que no programa original e que não existe uma relação diretamente proporcional entre o tamanho do genoma e o número de sRNAs encontrados. Mostrando que um erro numérico simples pode comprometer todo o resultado final. A mesma tabela mostra também o tempo que os programa gastam para encontrar um candidato a sRNA, no caso da reescrita de código o tempo necessário para encontrar um sRNA ficou na casa dos centésimos de segundos em todos os casos testados. Já o programa original precisa de no mínimo 31,950 segundos no melhor caso e 44,505 segundos no pior caso para encontrar um candidato a sRNA. A quantidade de memória RAM utilizada durante a execução do genoma da bacteria *Bacilus anthracis Ames AE016879* é de aproximadamente 20 MB.

Como foi mostrado no tópico 2 (Materiais e Métodos), a primeira versão da reescrita apresenta os mesmos resultados que o programa original, mas o número de resultados aumenta exponencialmente na segunda versão, como pode ser análisado na tabela 2. Estes novos resultados estão sendo analisados para que a afirmação de que novas áreas contendo sRNAs podem ser encontradas com a nossa versão do programa. A análise está sendo feita por um especialista na área biológica.

5. Conclusões

Neste artigo apresentamos duas reescritas de código para o programa sRNAScanner para otimizar a busca de sRNAs. O software original é *open source*, voltado para fins de pesquisa, e está sob a licença GNU GPL. A segunda versão reescrita do código está disponibilizado gratuitamente para o público interessado no link https://dl.dropboxusercontent.com/u/50877409/reescrita_codigo.cpp. A pesquisa se baseou na proposta do trabalho de Fujii (2011), que aplicou técnicas de processamento de alto desempenho no mesmo software. Porém, neste trabalho foi feito uma reescrita total do código e somente algumas ideias foram aproveitadas e não partes do código. O principal resultado obtido no final foi o ganho no tempo de execução do programa e a correção de erros que prejudicavam os resultados. A otimização serviu para adequer o programa aos processadores multi-core e utilizar corretamente as memórias RAM e cache dos processadores. O software agora aproveita melhor o poder de processamento disponível na máquina. Também há mais trechos onde o multi-processamento é utilizado (*pthreads*) diminuindo assim o tempo em que o núcleos da CPU ficam ociosos.

A melhora no tempo de execução superou o esperado e mais do que comprovou que o acesso ao disco rígido e conversão de tipos de dados são, realmente, os maiores problemas nos códigos.

Como trabalhos futuros novos testes com os genomas testados por Shridar (2010) serão feitos e os resultados validados por um especialista na área biológica, pois se ficar comprovado que o programa original não retornava todos os possíveis sRNAs será uma grande contribuição para a área de bioinformática. Um outro trabalho já em desenvolvimento é reescrever algumas funções que necessitam de um grande poder de processamento para serem executadas em placas gráficas com tecnologia GPGPU (*General Purpose Graphics Processing Unit*), utilizando-se das tecnologias Nvidia CUDA e/ou OpenCL. Como o código é composto por muitas estruturas de repetição, há alguns trechos que poderão ser paralelizados de forma eficiente. Neste trabalho, por enquanto com foco na arquitetura CUDA foi considerara a utilização da biblioteca Thrust, potencializando a utilização de memória na GPU, considerando que muitos trechos do código utilizam os mesmos valores e estruturas de dados para realizar operações com poucas diferenças entre si, que são realizadas dentro de estruturas de repetição. O código tambêm utiliza coleção de dados como vetores e matrizes, que poderão ser indexados e utilizados mais eficientemente com as funcionalidades disponíveis em CUDA. Os resultados da reescrita das funções que foram consideradas aptas à paralelização em CUDA serão analizados para concluir quais as verdadeiras vantagens da paralelização. Essas vantagens dependem de uma série de fatores,

considerando o tempo de processamento e o custo computacional da movimentação entre memória como medidas de desempenho.

Referências Bibliográficas

B. Schmidt (2010), Bioinformatics: High Performance Parallel Computer Architectures (Embedded Multi-Core System). CRC Press.

G. Ananth; G. Anshul; K. George, Introduction to Parallel Computing – Pearson Education, Janeiro 2003

J. Sridhar, S. R. Narmada, R. Sabarinathan, H. -Y. Ou, Z. Deng, K. Sekar, Z. A. Rafi, and K. Rajakumar (2010), "sRNAscanner: A Computational Toll for Intergenic Small RNA Detection in Bacterial Genomes," PLoS ONE, vol. 5, no. 8, pp. E11 970+, Aug.

D. Patterson and J. Hennessy (2009), Computer organization and design: the hardware/software interface, ser. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Morgan Kaufmann, 2009.

M. Cunha, A. Coutinho, and J. Telles (2001), "High performance computing applied to boundary elements: Potential problens," Cilamce, vol. 27, pp. 182-188, Jan.

J. Sridhar, Z. A. Rafil, and K. Rajakumar, (2013), "srnascanner 1.0 user guide," http://bicmku.in:8081/, access in 2013, January.

B. Nichols, D. Buttlar, J. P. Farrell, (1996) "Pthreads programming – a POSIX standard for better multiprocessing". O'Reilly.

Fujii, Sérgio Yoshimitsu, "Utilização de técnicas de otimização de desempenho em Bioinfotmática: estudo de caso: sRNAscanner" Dissertação de pós-graduação em Bioinformática, Setor de educação profissional e tecnológica, Universidade Federal do Paraná, 88f. Curitiba, 2012.