Um estudo sobre a paralelização do método do Gradiente Conjugado

Ricardo Neves Carvalho[1], Ana Paula Canal[2]

[1]Acadêmico do curso de Ciência da Computação – Centro Universitário Franciscano Rua dos Andradas, 1614 – 97010-032 – Santa Maria – RS – Brasil

[2]Orientadora. Professora do curso de Ciência da Computação – Centro Universitário Franciscano Rua dos Andradas, 1614 – 97010-032 – Santa Maria – RS – Brasil

ricardo.neves@unifra.edu.br, apc@unifra.br

Abstract. *The resolution of linear systems can be done through iterative methods. The Conjugate Gradient method belongs to this class and was used to solve the systems of this paper. This method is best suited for large systems and the associated matrix needs to be square, symmetric, and definite positive so that it can be applied. Due to the large volume of data parallelization is possible, aiming at better performance. In this paper we tried to use OpenCL, which due to installation problems and compatibility did not obtain positive results and OpenMP, that is able to divide the processing between the cores of the processor.*

Resumo. *A resolução de sistemas lineares pode ser feita através de métodos iterativos. O método do Gradiente Conjugado pertence a esta classe e foi usado para a solução dos sistemas deste trabalho. Esse método é mais indicado para grandes sistemas e a matriz associada precisa ser quadrada, simétrica e positiva definida, para que possa ser aplicado. Devido ao grande volume de dados é cabível a paralelização, visando em melhor desempenho. Nesse trabalho tentou-se utilizar OpenCL, que devido a problemas de instalação e compatibilidade não obteve resultados positivos e OpenMP, que conseguir dividir o processamento entre os núcleos do processador.*

Palavras-Chaves – Resolução de Sistemas Lineares; GPU (Unidade de Processamento Gráfico); Programação Paralela.

1. Introdução

Muitos problemas do cotidiano podem ser resolvidos usando equações lineares. Considera-se uma equação linear quando existem *n* incógnitas para a equação, e possuem um formato específico [Lathi 2007]. Estas, por sua vez podem estar juntas em um sistema linear e podem ser encontradas na área da eletrônica, por exemplo. Conforme Rufato (2014), é necessário levar em consideração algumas leis básicas dos circuitos elétricos que podem ser expressas na forma de sistemas lineares.

Para resolução dessas equações, há os métodos indiretos ou iterativos, que permitem obter a solução do sistema com uma dada precisão. São exemplos dessa classe os métodos: Jacobi, Gauss-Seidel, Gradiente Conjugado (GC), entre outros. Em muitos casos a matriz

associada ao sistema possui uma grande quantidade de linhas e colunas, que possibilita sua resolução por meio da implementação destes métodos numéricos.

Para esses casos que o GC pode ser uma forma de resolução, pois é um método numérico que resolve sistemas lineares do tipo $Ax = b$. Sabe-se que para a representação dessas equações são utilizadas matrizes e para resolução com este método, as matrizes devem ser simétricas e positivas definidas [Shewchuk 1994].

Este método possui operações entre matrizes e vetores como soma, cálculo de produto escalar dos vetores e multiplicação entre vetor e matriz, sendo que essa última é a que mais impacta no desempenho computacional do método. A paralelização se torna viável, devido ao fato dessa solução de sistemas lineares ser mais indicada para grandes sistemas, e é nesse sentido que cabe melhorar desempenho [Grisa 2010].

A programação concorrente é uma forma de realizar a execução visando um melhor desempenho. Atualmente os processadores comerciais mais comuns têm sido fabricados com dois, quatro, oito, ou até mais núcleos de processamento, o que possibilitam um maior desempenho na execução das tarefas pela CPU (Unidade Central de Processamento). Por conter essa quantidade de núcleos é praticável a execução de programas em paralelo, que pode ser feita através de linguagens de programação e bibliotecas destinadas a este fim.

No entanto, a paralelização pode não ocorrer apenas na CPU, podendo ser implementada em placas gráficas. Estas, são destinadas especificamente para processamento paralelo de operações de ponto flutuante, o que proporciona uma maior capacidade de processamento [Nvidia 2017]. Porém, um processo não será paralelizado de forma automática, sendo necessário programar para que isso aconteça.

Para que seja possível, a ideia foi usar a plataforma de desenvolvimento OpenCL (linguagem de computação aberta, em uma tradução livre), que permite paralelizar as operações pertencentes do método numérico. Utilizando esta visa-se obter portabilidade, já que é possível programar em arquiteturas heterogêneas, ou seja, em CPU e GPU (Unidade de Processamento Gráfico) [Khronos OpenCL Working Group 2015].

Observando os recursos apresentados pelo OpenCL, juntamente com a pré-disposição do método para a paralelização, este trabalho é um estudo de caso da viabilidade da paralelização do método do GC com a API (Interface de Programação de Aplicações) OpenCL e OpenMP.

1.1. Objetivo geral

Paralelizar o método numérico do Gradiente Conjugado utilizando as APIs OpenCL e OpenMP.

1.2. Objetivos específicos

Para alcançar o objetivo geral, tem-se os seguintes objetivos específicos:

a) Desenvolver um algoritmo sequencial do Gradiente Conjugado;
b) Desenvolver algoritmos paralelos utilizando OpenCL e OpenMP;
c) Analisar o desempenho dos algoritmos paralelos desenvolvidos.

1.3. Justificativa

A necessidade de obter resultados processados computacionalmente de uma grande quantidade de dados em tempo hábil, é realidade de inúmeras organizações atualmente. Nesse sentido torna-se importante usar ferramentas que possam auxiliar nessa tarefa, conforme Hölbig, Mazzanetto e Pavan (2017). As afirmações acima justificam a necessidade de desenvolver programas que possam processar dados da melhor forma possível para o usuário.

Nesse sentido, a paralelização pode possibilitar um melhor desempenho de uma determinada aplicação, e o padrão OpenCL pode auxiliar nisso. Além da capacidade de paralelizar, para Munshi (2011) nenhum outro padrão de programação permite tal portabilidade entre modelos, marcas e linguagens de programação. Ainda conforme Serpa *et al.* (2017) o padrão de programação OpenMP (Multi-Processamento Aberto) é indicado para arquiteturas de memória compartilhada, utilizando diretivas *#pragma*, baseadas em C/C++. Portanto, devido aos processadores atuais possuírem mais de um núcleo de processamento, é viável utilizar esse padrão nessa arquitetura.

Outro aspecto importante, conforme Canal (2000) sobre o GC, justifica-se sua utilização por ser um método que envolve operações básicas entre matrizes e vetores, o que facilita sua implementação. Todavia, o desenvolvimento de um programa paralelo não é trivial, pois utiliza memória compartilhada e alternância de processamento entre a CPU e GPU.

1.4. Organização do texto

O texto está organizado da seguinte forma: na Seção 2, definição do método, OpenCL e OpenMP. Na Seção 3, os trabalhos relacionados. A metodologia, na Seção 4, onde são determinados os passos para a confecção do trabalho. O projeto de paralelização é descrito na Seção 5. A conclusão e os trabalhos futuros são abordados na Seção 6.

2. Referencial Teórico

Nesta seção são descritos o método do Gradiente Conjugado e as bibliotecas OpenCL e OpenMP.

2.1. Gradiente Conjugado

Sistemas lineares devem estar expressos na forma conforme a Figura 1.

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \ldots + a_{2n}x_n = b_2 \\ \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot \\ \cdot \quad \cdot \quad \cdot \quad \cdot \quad \cdot \\ a_{m1}x_1 + a_{m2}x_2 + a_{m3}x_3 + \ldots + a_{2n}x_n = b_m \end{cases}$$

Figura 1. Sistema Linear.

Considera-se uma equação linear quando existem *n* incógnitas para a equação e representadas nesta forma [Lathi 2007]. Sistemas lineares são compostos por várias equações e são aplicados de vários modos, na programação linear, previsão meteorológica, modelagem do clima, computação científica, entre outros [Saad 2003]. Todos os exemplos supracitados podem gerar grandes sistemas lineares, com muitas incógnitas e valores nulos.

Para a solução destes, há os métodos iterativos que partem de uma solução inicial arbitrária e seu cálculo segue gradualmente até que o critério de parada seja alcançado, geralmente ligado ao limite de tolerância do erro. Dentro desta classe, existem os métodos não estacionários, apresentam hereditariedade entre as iterações e para a paralelização o Gradiente Conjugado é um exemplo dessa categoria. Uma vantagem desses é não utilizar operações elementares entre as linhas e colunas, o que evita a troca de elementos da matriz.

Uma matriz é dita esparsa quando a maioria dos seus elementos são nulos. Dessa forma, pode ser estruturada ou não estruturada [Saad 2003]. As estruturadas, possuem um padrão regular dos seus elementos não-nulos, frequentemente ocorrem próximos a diagonal principal, chamada de matriz banda. Já as desestruturadas não possuem padrão para a ocorrência dos seus elementos. Na Figura 2, pode-se ver a diferença visual de ambas, sendo que os quadrados pretos de cada representam os elementos não-nulos.

(a) estruturada (b) desestruturada

Figura 2. Exemplo de matrizes esparsas.

Em termos computacionais a esparsidade da matriz pode ser vista como problema, já que ela precisará ser carregada na memória principal do computador. Deve-se encontrar uma maneira de economizar espaço de memória e minimizar as operações com elementos nulos presentes. Existem maneiras de armazenar matrizes visando a sua otimização, são elas: formato diagonal, formato *Matrix Market* conforme previsto por NIST (2007), entre outras.

Com relação ao Gradiente Conjugado, seu estudo vem desde a década de 50 motivado pela tentativa de encontrar uma melhor solução de sistemas provenientes da discretização de equações diferenciais parciais e viabilizados pela entrada dos computadores nos cálculos científicos [Cunha 2003]. Para que o método possa ser aplicado, há algumas condições necessárias como:

- $A_{n \times m}$ *se, somente se,* $n = m$, o que implica em uma matriz quadrada;
- $A^t = A$, sua matriz transposta ser igual a sua original (matriz simétrica);
- A matriz A ser positiva e definida, $x^tAx > 0$ para todo $x \neq 0$.

Sendo: A é a matriz dos coeficientes, x é o vetor de incógnitas, b é o vetor dos termos independentes e t é a transposição entre linhas e colunas. Uma matriz é positiva definida quando os determinantes, das submatrizes quadradas (dentro da matriz A) são positivos.

O método parte do princípio que o gradiente formado é um campo vetorial, que aponta para a direção mais crescente da função $F(x)$ (função quadrática) [Shewchuk 1994]. A resolução do sistema linear consiste na base da parabolóide, onde o gradiente é zero. Por tanto, é preciso seguir a direção contrária ao crescimento da função, assim tentando chegar na base da parabolóide (a solução), como visto na Figura 3.

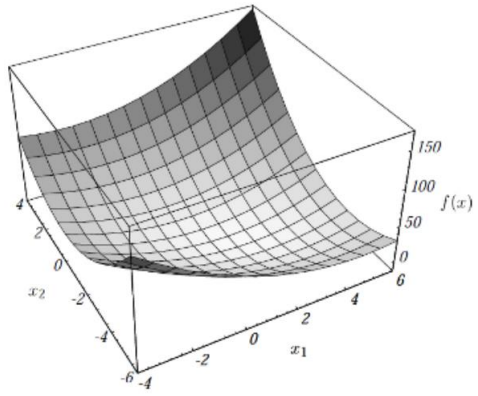

Figura 3. Exemplo de uma parabolóide.

Conforme Cunha (2003), é preciso calcular a direção e o tamanho do passo a ser dado. Para isso, é necessário calcular o resíduo, que indica o quão distante se está da solução do sistema. Para que isso seja possível, é necessária a atualização do resíduo a cada iteração realizada. O início do método geralmente é dado por, $x^0 = 0$, quando não há uma boa estimativa inicial para x. A partir dessa formação, a direção de busca é definida, para que uma direção já pesquisada não seja novamente executada, até encontrar a solução desejada [Cunha 2003].

Com relação ao critério de parada, é definido um número máximo de iterações, juntamente com a norma do resíduo, e termina quando a norma do resíduo atual for maior que o erro multiplicado pela norma do resíduo da iteração 0, representado pela fórmula $||r^i|| > \varepsilon ||r^0||$. O valor ideal para ε é 10^{-6}, que também foi adotado neste trabalho [Shewchuk 2010]. A Figura 4 apresenta o algoritmo do GC em pseudocódigo, que descreve as operações que devem ser realizadas.

```
1   i = 0
2   r = b − Ax
3   d = r
4   δnovo = rr
5   δ0 = δnovo
6   Enquanto i < imax e δnovo > ε²δ0
7       q = Ad
8       α = δnovo / dq
9       x = x + αq
10      Se i é divisível por 50
11          r = b − Ax
12      senão
13          r = r − αq
14      δvelho = δnovo
15      δnovo = rr
16      β = δnovo / δvelho
17      d = r + βd
18      i = i + 1
```

Figura 4. Descrição do algoritmo sequencial do Gradiente Conjugado. [Shewchuk 1994]

Na Figura 4, a variável i armazena a quantidade de iterações do método; i_{max} é o número máximo de iterações; A é a matriz esparsa associada ao sistema linear empregado; b é um vetor de termos independentes; r é o vetor de resíduo que deve ser novamente calculado a cada iteração; d é a direção de pesquisa para encontrar o resultado; δ vetor que armazena a norma do resíduo; α é o tamanho do passo, dado pelo método para encontrar a solução; β é a razão entre os resíduos novo e velho; x é a solução do algoritmo.

2.1.1. Armazenamento da matriz esparsa

O armazenamento da matriz esparsa pode-se dar no formato diagonal, onde são necessárias duas estruturas, uma matriz M e um vetor Dif. Cada coluna da matriz M, armazena uma diagonal da matriz A (que é esparsa) e o vetor Dif é responsável por armazenar a diferença de cada diagonal a diagonal principal. A Figura 5 ilustra como isso ocorre, conforme descrito por Canal (2000).

$$A = \begin{pmatrix} 3 & 1 & 0 & 0 & 0 \\ 1 & 3 & 1 & 0 & 0 \\ 0 & 1 & 3 & 1 & 0 \\ 0 & 0 & 1 & 3 & 1 \\ 0 & 0 & 0 & 1 & 3 \end{pmatrix}_{5\times5} \Leftrightarrow M = \begin{pmatrix} 0 & 3 & 1 \\ 1 & 3 & 1 \\ 1 & 3 & 1 \\ 1 & 3 & 1 \\ 1 & 3 & 0 \end{pmatrix}_{5\times3} \quad Dif = \begin{pmatrix} -1 & 0 & 1 \end{pmatrix}_{1\times3}$$

Figura 5. Exemplificação de armazenamento utilizando formato diagonal.

Este foi o formato adotado para o armazenamento da matriz esparsa, com o intuito de minimizar seu espaço na memória e visando um melhor desempenho do método do GC, analisando o tempo de processamento até que se chegue na resolução desejada.

2.2. OpenCL

O termo GPU foi cunhado por meados de 1990, quando seu processamento gráfico e a possibilidade de programação das placas gráficas foram intensificados, segundo Figueirêdo Júnior (2015). Desde então a aceitação do mercado para as placas só aumentou, tendo em vista que quanto maior desempenho, melhor para o usuário final.

Com o passar do tempo o número de processadores e a quantidade de memória disponível aumentou, com isso a possibilidade de processar dados não gráficos, e assim surge a nomenclatura GPGPU (Unidades de Processamento Gráfico de Propósito Geral). A GPU possui núcleos (ou *cores*) de processamento. Entretanto, a CPU possui pequenos *cores* otimizados para processamento sequencial, já a GPU é projetada especificamente para processamento paralelo de operações de ponto flutuante, o que proporciona uma maior capacidade de processamento [Nvidia 2017], como mostrado na Figura 6.

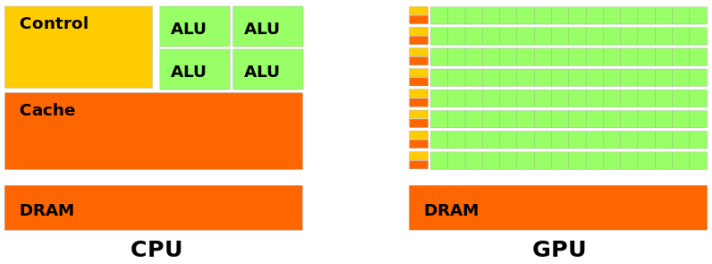

Figura 6. Ilustração da divisão interna das unidades de processamento.

O OpenCL é uma API de programação paralela para ambientes heterogêneos, desenvolvido pela Khronos Group, que como seu nome já sugere é uma linguagem de computação aberta, o que permite que seu código-fonte possa ser executado em diferentes unidades de processamento e diferentes sistemas operacionais.

É afirmado por Pereira, Souza e Moreno (2013) "Programas únicos escritos em OpenCL podem ser executados em uma ampla gama de sistemas, a partir de telefones celulares, computadores portáteis até nós de supercomputadores". Nota-se que esta afirmação é uma das vantagens de sua utilização.

A arquitetura básica desta plataforma pode ser descrita utilizando quatro modelos: de plataforma, de execução, de memória e de programação. O modelo de plataforma, de maneira simples é composto por um *host* conectado a pelo menos um dispositivo OpenCL (CPU ou GPU) que são divididos em Unidades de Computação (UCs), e dentro desses subdivididos em Elementos de Processamento (PEs), a Figura 7 mostra essas divisões.

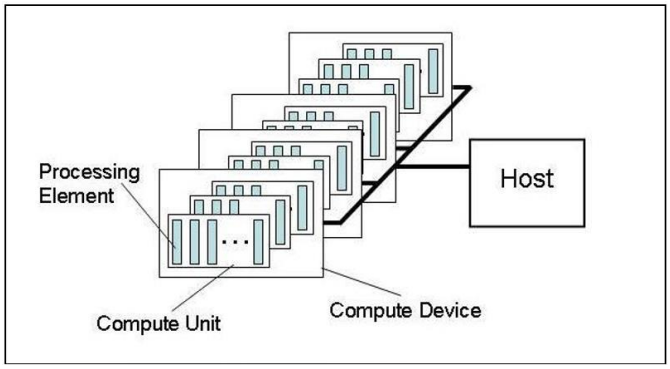

Figura 7. Modelo de plataforma [Khronos OpenCL Woking Group 2015].

O modelo de execução ocorre em duas etapas: a primeira chamada *kernels*, onde as funções são executadas nos dispositivos. A outra, chamada de programa de *host*, fica encarregada de executar as partes sequencias do programa e gerencia a execução da etapa anterior [Khronos OpenCL Woking Group 2015].

Com relação ao modelo de memória, existem cinco tipos de regiões, sendo elas: memória de *host* que é visível apenas para o *host* que está manipulando a variável; memória global, permite leitura e escrita para todos os elementos presentes; memória constante, é permitida apenas a leitura para os elementos; memória local, permite acesso de leitura e escrita, apenas para os elementos do mesmo *kernel* e a memória privada, que é visível apenas dentro do elemento. Na Figura 8 a ilustração os tipos de memória.

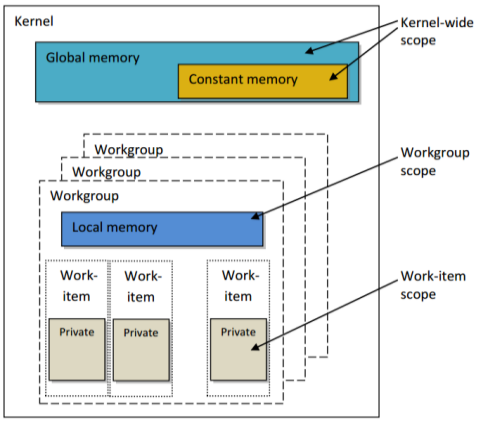

Figura 8. Modelo de memória [Gaster et al. 2012].

Por fim, o modelo de programação suporta as linguagens C/C++, Java e Python, sendo possível realizar paralelismo de dados e tarefas, de forma separada, ou híbrida.

2.3. OpenMP

Esta API é uma especificação aberta para multiprocessamento que atua em ambientes de memória compartilhada [LLNL 2017]. Nesse ambiente, existem vários núcleos de processamento e uma memória compartilhada entre eles, como visto na Figura 9. Possui suporte pelos sistemas operacionais Linux/Unix e Windows e possui diretivas para compiladores das linguagens Fortran, C/C++ e uma implementação para Java utilizando JOMP.

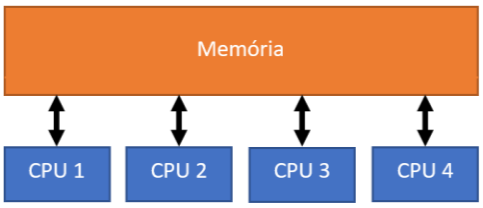

Figura 9. Modelo de memória compartilhada.

São providas diversas diretivas de compilação, biblioteca de rotinas e variáveis de ambiente, as quais são inseridas diretamente em códigos sequenciais [Gressler e Cera 2014]. Sua implementação é feita pela OpenMP e atualmente está na versão 4.5 de novembro de 2015. Esta API usa modelo *fork/join* (mestre/escravo), onde há um fluxo de execução principal e novos *threads* são disparadas para dividir o trabalho e no fim uma seção paralela, é feito um *join*, ou seja, a sincronização entre os *threads*.

É importante citar algumas características que o diferenciam de outros padrões. Não é necessário definir o número de *threads* a serem criados, por padrão serão criados a mesma quantidade de núcleos do processador, podendo ser definido um número diferente. Além disso, não é possível ver como cada *thread* é criado e inicializado. Assim como, a divisão do trabalho não é explicita, o que facilita sua programação.

3. Trabalhos Relacionados

Durante a pesquisa foram encontrados alguns trabalhos semelhantes e definidos como correlatos.

Foi desenvolvido por Grisa (2010) o trabalho intitulado de: GPU computing: Implementação do método do Gradiente Conjugado utilizando CUDA. Consistem em uma aplicação paralela utilizando CUDA, em uma arquitetura Nvidia Tesla e matrizes simétricas positivas e definidas, encontradas no repositório *Matrix Market*. Houve uma redução do tempo em até 50% quando utilizado o algoritmo em CUDA paralelizado, se comparado com o sequencial. Foi testado em uma placa gráfica Nvidia GeForce 250, com 1GB de memória DDR3. A grande diferença entre o presente trabalho e este é a API utilizada, e por consequência a arquitetura.

Outro trabalho selecionado foi Avaliação do Desempenho de Duas Versões do Algoritmo do Gradiente Conjugado Paralelizado em Cluster de PCs, desenvolvido por Galante, Balbinot e Martinotto (2002) tem como objetivo avaliar o desempenho do algoritmo clássico de Shewchuk (1994) e outra versão de Chronopoulos (1989). Para isso, foi utilizada a biblioteca Pthreads e o resultado final foi uma superioridade de 2% do algoritmo clássico ao de Chronopoulos (1989), que foi atribuída a um cálculo de produto escalar adicional presente neste. A maior consideração do trabalho é a comparação estabelecida entre os dois algoritmos, mostrando que o de Shewchuk é superior dentro do ambiente em que foi testado.

O terceiro, é o trabalho de Bueno (2013), intitulado de: Resolução de sistemas de equações lineares de grande porte em clusters multi-GPU utilizando o método do gradiente conjugado em OpenCL™, que procura explorar o potencial computacional de múltiplas GPUs, para solucionar os sistemas lineares. Da mesma forma, estabelece uma comparação entre CUDA e OpenCL testado em dois ambientes, utilizando placas gráficas fabricadas pela Nvidia. Ambos ambientes são *clusters* de placas de vídeo interligadas entre si. O primeiro chamado de MultiGPU 1, composto por três placas gráficas Tesla C1060, com 30 Unidades de Computação (UC), cada uma. E o MultiGPU 2 constituído por duas placas gráficas Tesla C2075, com 14 UCs cada.

Notou-se que CUDA obteve um melhor desempenho, no ambiente de teste, concluindo que é uma boa escolha para desenvolvimento de aplicações paralelas, desde que exista *hardware* compatível. Com relação a paralelização do método houve uma redução do tempo de execução de 63% no ambiente MultiCPU 1 e de 41% no MultiGPU 2. No comparativo entre as APIs, nota-se uma superioridade de CUDA nos testes feitos, quanto ao OpenCL. Apesar disso, é necessário atentar o propósito de desenvolvimento de cada uma delas, uma é exclusiva para placas Nvidia, a outra é mais voltada para ambientes heterogêneos, onde podem haver CPU e GPU.

Na Seção 4, serão abordados maiores detalhes o ambiente de teste e como se dará o desenvolvimento do trabalho.

4. Metodologia

De acordo com as definições propostas por Gil (2002), este trabalho foi definido como uma pesquisa exploratória com revisão bibliográfica, pois teve o intuito de pesquisar sobre o

método do GC e de como a paralelização deste pode contribuir para um melhor desempenho do mesmo. Nesse sentido, o objeto de estudo foi o padrão OpenCL, o que caracteriza um estudo de caso do mesmo, explanando melhor suas especificidades.

Para que esta pesquisa fosse possível, a linguagem utilizada como padrão nos manuais publicados pelas marcas que possuem compatibilidade com o OpenCL é C/C++. Assim, esta foi a linguagem adotada para o desenvolvimento da aplicação paralela, com o objetivo de utilizar a GPU para a resolução dos sistemas lineares.

Sobre o algoritmo sequencial que foi desenvolvido, conforme visto na Figura 4, na Seção 2.1, tem-se os seguintes passos:

1. Inicializar a variável *i* para o número de iterações realizadas;
2. Fazer a aproximação inicial do primeiro resíduo;
3. Igualar os vetores *d* e r_0 (para não perder o valor de r_0);
4. Calcular o produto escalar de *r* e r^t, *e armazenar em* δ;
5. Definir δ_0 como δ_{novo};

Início do laço de repetição: enquanto o número máximo de iterações não for atingido e a norma do resíduo for maior que uma fração de primeiro resíduo multiplicado pelo erro.

6. Multiplicação entre a matriz esparsa *A* e o vetor *d*, e armazenar o resultado em *q*;
7. Definir o tamanho do passo a ser dado, pela fórmula $\alpha = \frac{\delta_{novo}}{d^t q}$;
8. Definir a aproximação de *x*, por $x = x + \alpha q$;

A cada 50 iterações será feito uma atualização do resíduo. Se for a iteração 50:

9. Calcular o novo resíduo através da subtração dos vetores, pela fórmula $r = r - Ax$;

Se não

9. Calcular o novo resíduo através da subtração dos vetores, pela fórmula $r = r - \alpha q$;
10. Igualar δ_{velho} como $\delta_{novo;}$
11. Calcular o δ_{novo} pelo produto escalar entre de *r* e r^t;
12. Calcular β para auxiliar no cálculo da nova direção de pesquisa, feita pela razão entre δ_{velho} e δ_{novo};
13. Calcular a nova direção de pesquisa; pela multiplicação escalar de β e *d*, e seu resultado somado ao vetor *r*;
14. Incrementar *i* com + 1;
15. Repetir os passos 6 ao 15, até satisfazer o critério de parada definido no laço de repetição.

Fim do método, sistema linear resolvido.

Conforme visto nos trabalhos de Shewchuk (1994), a definição do valor cinquenta para a atualização do resíduo é dada pelos autores que realizaram o estudo da paralelização desse método. A utilização desse valor é devido ao fato da atualização comum, pela fórmula *r*

$= r - Ax$, ser mais custosa gerando uma multiplicação a mais entre matriz e vetor. Por isso, se faz sua aproximação, pela fórmula $r = r - \alpha q$, e a cinquenta iterações é realizado o cálculo para a correção dessa aproximação feita.

Para os testes, que foram necessários para que a execução da aplicação, foi adotado o repositório *online Matrix Market*, pois segundo NIST (2007) este é um repositório de dados de teste para estudos comparativos de algoritmos para álgebra linear numérica, com quase 500 matrizes esparsas. Entre as matrizes disponíveis, estão as que se enquadram muito bem para o método, que foram as: BCSSTK18, BCSSTK27, BCSSTM25, BCSSTM24, BCSSTM26, NOS7.

Para os testes de desempenho foram desenvolvido um programa sequencial do GC em C e a tentativa de uma versão equivalente utilizando OpenCL. Ao longo deste trabalho também foi desenvolvida uma outra versão em paralelo utilizando OpenMP, tendo em vista que esta é uma API aberta de programação e é capaz de paralelizar dados entre os núcleos do processador. Para a comparação entre as versões foram observados o tempo de execução de cada uma delas, e o número de iterações para chegar ao resultado final.

Para os testes foi utilizado um *notebook* com processador Intel Core i7-4510U (4MB de cache e 2 GHz com 4 núcleos de processamento), uma placa gráfica AMD Radeon R7 M260 (*clock* de 980 MHz e 2 GB em DDR3) e 16GB de memória RAM. O sistema operacional de teste foi o Microsoft Windows 10.

5. Desenvolvimento do algoritmo paralelo

O objetivo deste trabalho foi de paralelizar os cálculos matemáticos entre matrizes e vetores. Tendo em vista que a paralelização destas regiões do código, podem refletir num ganho de desempenho, pois há um grande conjunto de dados sendo trabalhado nestes.

Portanto, a paralelização deste trabalho se dá nas seguintes equações/operações, descritas anteriormente no algoritmo sequencial:

- a multiplicação entre a matriz esparsa A e o vetor x, em $r = b - Ax$;
- a subtração dos vetores b e Ax;
- a atribuição do vetor r, ao vetor d, em $d = r$;
- o produto escalar entre o vetor r com ele mesmo, em $\delta_{novo} = rr$;
- a multiplicação entre a matriz A e o vetor d, em $q = Ad$;
- a multiplicação e soma dos vetores, em $x = x + \alpha q$;
- a multiplicação e subtração entre os vetores, em $r = r - \alpha q$;
- e a multiplicação e soma dos vetores, $d = r + \beta d$.

Incialmente foi programada e testada a versão sequencial do algoritmo, que obteve os seguintes resultados, vistos no Quadro 1. Para o armazenamento das matrizes foi utilizado o formato diagonal, de acordo com a especificidade prevista na seção 2.1.1.

Quadro 1. Resultado dos testes de tempo de execução no algoritmo sequencial.

Nome da matriz	Dimensão	Número de iterações	Tempo médio (s)	Estrutura
BCSSTK18	11948	-	-	Esparsa Diagonal
BCSSTK27	1224	50000	389477,00	Esparsa Diagonal
BCSSTM24	3562	27	1557,00	Esparsa Diagonal
BCSSTM25	15439	102	111,45	Esparsa Banda Diagonal
BCSSTM26	1922	94	3136,00	Esparsa Banda Diagonal
NOS7	729	28502	146,09	Esparsa Bloco Diagonal

No repositório *Matrix Market*, haviam outras matrizes que poderiam ser solucionadas com este método, no entanto não possuíam grandes dimensões, o que não tornaria viável a paralelização destes dados. A matriz BCSSTK18 não conseguiu ser solucionada em tempo tolerável de processamento. Por outro lado, a matriz BCSSTK27 não foi resolvida em menos de 50000 iterações. Como ambas as matrizes pertencem ao mesmo tipo de problema matemático do repositório, compreende-se que estas não são bons exemplos de matrizes que podem ser resolvidas com o método do GC.

Com relação a versão paralela, primeiramente foi definido como os dados iriam ser armazenados entre as memórias disponíveis, dessa forma as operações matemáticas propostas pelo GC, pretendiam ser divididas entre os kernels. A memória global ficaria responsável em armazenar os dados da matriz e dos vetores quando os cálculos já estiverem prontos, e para que cada parcela de dados seja dividida entre as unidades de computação (UCs) pretende-se utilizar a memória local e privada.

Houveram alguns problemas durante a confecção desta versão. Primeiramente um problema de compatibilidade entre o sistema operacional, que seria o Ubuntu 16.04 LTS, devido a decisão da AMD tirar o suporte para o sistema restou apenas os *drivers* de código aberto da empresa, que possuíam certa instabilidade, já relatada por outros usuários. Sua instalação foi concluída, porém devido a falta de manuais mais detalhados explicando seu funcionamento, algum pacote pode ter deixado de ser instalado, assim dificultado a programação com OpenCL na placa gráfica.

Tendo em vista este cenário e prevendo um possível problema de compilação neste sistema operacional, utilizou-se também o CMake que é uma aplicação com linguagem própria que pode compilar e compatibilizar códigos escritos em *makefile* (Ubuntu) em projetos do Visual Studio (Windows). Junto a isso foi necessário instalar alguns pacotes para o desenvolvimento de aplicações OpenCL que são: *pocl-opencl-icd*, *libpocl1*, *libpocl1-common*, *pocl-opencl-icd*, *libpocl-dev*, *ocl-icd-opencl-dev*, *ocl-icd-libopencl1*, *opencl-*

headers e *amd-opencl-icd*. Entre estes, maioria deles possuem código aberto, com exceção do último, que é a implementação da AMD para a linguagem, que é necessário ser instalado, tendo em vista o ambiente de teste.

Além disso foi necessário implementar um pequeno algoritmo em Phyton para limpar os arquivos residuais que eram criados a cada execução de qualquer trecho de código paralelo. Também foi necessário buscar projetos externos que pudessem listar os componentes suportados pelas implementações do OpenCL e que pudessem mostrar os erros que poderiam vir a ocorrer durante a execução do código.

Porém, a programação deste se mostrou mais trabalhosa do que se imaginava a princípio. Primeiramente, foi realizada apenas uma soma de vetores com números randômicos, que obteve sucesso, a partir disso foi sendo acrescido ao código OpenCL os trechos de programação já previstos no algoritmo sequencial, no entanto não se conseguiu chegar ao final do código, pois era mostrado um erro de compilação, *CL_OUT_OF_RESOURCES*. Este, notifica que não há recursos de *host* suficientes para alocar a implementação OpenCL. Pode estar ligado a um erro de *hardware* e ao contexto de programação definido, neste caso, a variável **source*.

Na Figura 9, é mostrado um trecho do código que se encontra o problema.

```
62    const char *source =
63        "__kernel void GC( \
64        __global const int** A, \
65        __global const int* b, \
66        __global int* r, \
67        __global int* d, \
68        __global int* q, \
69        __global int* x) \
70    { \
71    //INÍCIO DO MÉTODO - variável da quantidade de iterações, começa em zero\
72    cont = 0;\
73    \
74    //cálculo do resíduo (r = b - Ax)\
75    for(i=0; i<nroCol; i++){\
76        r[i] = 0;\
77        for(j=0; j<nroCol; j++) r[i] += A[i][j] * x[j];\
78    }\
79    for(i=0; i<nroCol; i++) r[i] = b[i] - r[i];\
80        int id = get_global_id(0); \
81        c[id] = a[id] - b[id]; \
82    }"
83        ;
84
85    //Obtenção de identificadores de plataforma e dispositivo. Será solicitada uma GPU.
86    clGetPlatformIDs(1,  & platformId, NULL);
87    clGetDeviceIDs(platformId, CL_DEVICE_TYPE_GPU, 1,  & deviceId, NULL);
```

Figura 9. Trecho de código em OpenCL.

Acredita-se que o problema apresentado no código pode estar atrelado às dificuldades de instalação dos *drivers* da AMD para o sistema operacional. Visto este fato, foram realizadas duas frentes de pesquisa, uma buscando resolver os problemas já enfrentados durante o desenvolvimento e outra pesquisando mais informações e detalhes de uma possível reinstalação.

Feita essa pesquisa, não foram encontrados muitos artigos, manuais ou vídeo aulas para desenvolvimento em sistemas operacionais da família Windows. A comunidade científica e comercial que trabalha com o OpenCL dá mais foco para sistemas Linux, porém não há detalhes relacionados a erros e problemas de instalação, foram encontrados mais tutoriais e vídeo aulas explicando de maneira geral como a API funcionava, mas não sua instalação de fato, e os que mostravam não estavam previstos para placas gráficas AMD.

Foi desenvolvido também o algoritmo em OpenMP, paralelizando os dados em CPU. Para isso, instalou-se o compilador MinGW (uma versão do GCC para Windows) e a IDE (Ambiente de Desenvolvimento Integrado) Code::Blocks. E, para a compilação do algoritmo, foi importado o arquivo *libgomp-1.dll* (presente no MinGW) para a IDE.

As principais diretivas utilizadas nesta implementação foram *#pragma omp fo*r e *#pragma omp parallel*, sendo que a primeira define a divisão das iterações de um laço de repetição entre os *threads* da seção paralela, e a outra define uma seção paralela e quais variáveis serão compartilhadas ou privadas entre os *threads* (processos). Os dados obtidos por meio desta implementação podem ser vistos na Quadro 2.

Quadro 2. Resultado dos testes de tempo de execução no algoritmo paralelo em OpenMP

Nome da matriz	Dimensão	Número de iterações	Tempo médio (s) Sequencial	Tempo médio (s) Paralelo
BCSSTK18	11948	-	-	-
BCSSTK27	1224	50000	389477,00	253,68
BCSSTM24	3562	27	1557,00	1,12
BCSSTM25	15439	102	111,45	78,33
BCSSTM26	1922	94	3136,00	1,35
NOS 7	729	28502	146,09	47,00

Para esta versão foram utilizados 4 *threads,* já que o processador da máquina de testes possui 4 núcleos de processamento, sendo assim cada *thread* processada por um núcleo. Com relação ao armazenamento na memória, nessa implementação os vetores e a matrizes esparsa foram definidos como compartilhados entre os núcleos e as variáveis auxiliares iterativas como privadas. Também foram definidos pontos de sincronização, através da diretiva *#pragma omp barrier*, sendo utilizados logo depois da seção paralela.

Com relação aos testes com as matrizes, a BCSSTK18 e BCSSTK27 prosseguiram com os problemas visto na versão sequencial, sustentando a justificativa que não são ideais para este método. Também notou-se um gargalo de execução no cálculo do primeiro resíduo em algumas das matrizes, no caso, as que possuem maior dimensão. O número de iterações se permaneceu igual a versão sequencial, para maioria dos casos, exceto BCSSTK18.

Já os tempos de processamento houve em média 1205,71s de ganho de processamento entre as matrizes que conseguiram atingir sua solução em menos de 50000 iterações. Isso comprova que a paralelização destes dados, mesmo que em CPU, resulta em um melhor desempenho das mesmas. Também foi testado o tempo de processamento entre as iterações de cada uma das matrizes, todas conseguiram resolver os cálculos em menos de 1 segundo, com exceção da BCSSTK18 que em algumas iterações levava mais de 5 segundos para a calcular. Na seção de Apêndices se encontram os algoritmos sequencial e paralelo.

6. Conclusões e Trabalhos Futuros

Inicialmente o estudo começou com um entendimento prévio das condições de uso, e como são realizadas as operações do Gradiente Conjugado, proposto por Shewchuk (1994). A partir da definição do método numérico, e da sua indicação de uso, estabeleceu-se que a paralelização dos dados da matriz esparsa associada seria o objetivo. Então, foram encontrados mais trabalhos relacionados utilizando CUDA, OpenMP, OpenACC e OpenCL.

O repositório *Matrix Market* possui uma grande quantidade de matrizes que podem ser solucionadas com estre método. No entanto, algumas possuem pequenas dimensões, o que não seria interessante para a paralelização, já que o intuito é dividir uma grande massa de dados. Portanto apenas matrizes que apresentaram tempos de solução altos no algoritmo sequencial foram comparadas com o algoritmo paralelo em OpenMP. Entre as matrizes comparadas, nota-se que há ganho de desempenho, já que os tempos de processamento foram menores na versão paralela.

Devido aos problemas e falta de material sobre OpenCL e suas dependências aos produtos da AMD, não se conseguiu chegar em um resultado positivo para a paralelização do método do Gradiente Conjugado. Dessa maneira, a principal contribuição deste trabalho é a sua revisão bibliográfica da API que visava ser implementada. Além disso, explica de que modo tentou-se fazer e por qual motivo não deu certo, sendo assim, não podendo ser aplicado para futuras implementações.

Almeja-se comparar de fato o que este trabalho se propôs numa próxima versão, já que é possível paralelizar. Para os trabalhos futuros sugere-se testar o algoritmo com outros formatos de armazenamento, dado que podem influenciar no tempo de execução e no número de iterações até que a solução do sistema seja encontrada. Também, testar em outras APIs, assim podem comparar melhor o desempenho que cada uma obtém para este algoritmo.

Referências

Bueno, Andre Luis Cavalcanti (2013). "Resolução de sistemas de equações lineares de grande porte em clusters multi-GPU utilizando o método do gradiente conjugado em OpenCL™", PUC-Rio, Rio de Janeiro, Brasil. Dissertação de mestrado.

Canal, Ana Paula (2000). "Paralelização de Métodos de Resolução de Sistemas Lineares Esparsos com o DECK em um Cluster de PCs", Universidade Federal do Rio Grande do Sul, Porto Alegre, Brasil. Dissertação de mestrado.

Chronopoulos, A. T.; Gear, C. W. (1989) "S-step Iterative Methods Symmetric Linear System. Journal of Computational And. Applied Mathematics", Vol 25.

Cunha, Maria Cristina (2003). "Métodos Numéricos", 2ª edição, Editora Pioneira, 2003.

Figueirêdo Júnior, Marco Antônio Caldas de (2015). "MASA-OpenCL: Comparação Paralela de Sequências Biológicas Longas em GPU", http://repositorio.unb.br/bitstream/10482/19439/1/2015_MarcoAnt%C3%B4nioCaldasdeFigueir%C3%AAdoJ%C3%BAnior.pdf, acesso em abril de 2017.

Galante, Guilherme; Balbinot, Jeysonn I. e Martinotto, André L. (2002) "Avaliação do Desempenho de Duas Versões do Algoritmo do Gradiente Conjugado Paralelizado em

Cluster de PC", III Workshop em Sistemas Computacionais de Alto Desempenho - 2002 - Vitória, ES, http://www.lbd.dcc.ufmg.br/colecoes/wscad/2002/0028.pdf, acesso em abril de 2017.

Gaster, Benedict et al. (2012). "Heterogeneous Computing with OpenCL". Advcanced Micro Devices, Publicado por Elsevier Inc.

Gil, A. C. (2012). "Como elaborar projetos de pesquisa". 4ª ed. São Paulo: Atlas. v. 1. 171p.

Gressler, Henrique de Oliveira; Cera, Márcia Cristina (2014). "O Impacto da paralelização com OpenMP no desempenho e na qualidade das soluções de um algoritmo genético". Revista Brasileira de Computação Aplicada, v. 6, n. 2.

Grisa, Maurício (2010). "GPU computing: Implementação do método do Gradiente Conjugado utilizando CUDA", Universidade de Caxias do Sul, Caixas do Sul, Brasil. Trabalho de Conclusão de Curso.

Hölbig, Carlos Amaral; Mazzanetto, Angela; Pavan, Willingthon (2017) "Computação Paralela com a Linguagem R: técnicas, ferramentas e aplicações". XVII Escola Regional de Alto Desempenho – ERAD 2017, Ijuí, Brasil.

Khronos OpenCL Working Group (2015) "The OpenCL Specification – version 2.1", The Khronos Group Inc.

Lathi, B. P. (2007) "Sinais e Sistemas Lineares", 2ª Edição, Editora Bookman, 2007.

LLNL. (2017) "OpenMP Tutorials". https://computing.llnl.gov/tutorials/openMP/, acesso em novembro de 2017.

Munshi *et al.* (2012) "OpenCL Programming Guide", Publicado por Pearson Education, Estados Unidos da América.

Nist (2007) "Matrix Market", http://math.nist.gov/MatrixMarket/, acesso em abril de 2017.

Nvidia (2017). "CUDA Programação Paralela Facilitada.", http://www.nvidia.com.br/object/cuda_home_new_br.html, acesso em abril de 2017.

Rufato, Sônia Aparecida Carreira (2014). "Sistemas lineares, aplicações e uma sequência didática", Instituto de Ciências e Matemáticas e de Computação – ICMC-USP, São Carlos. Dissertação de mestrado.

Saad, Yousef (2003). "Iterative Methods for Sparse Linear Systems", Publicado por Society for Industrial and Applied Mathematics, 2ª Edição.

Serpa, Matheus S. *et al.* (2017) "Intel Modern Code: Programação Vetorial e Paralela em Arquiteturas Intel Xeon e Xeon Phi" . XVII Escola Regional de Alto Desempenho – ERAD 2017, Ijuí, Brasil.

Shewchuk, Richard Jonathan (1994). "An Introduction to the Conjugate Gradiente Method Without the Agonizing Pain", Carnegie Mellon University, Pittsburgh. https://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf, acesso em abril de 2017.

Apêndices

A – Algoritmo do Gradiente Conjugado sequencial em linguagem C.

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<string.h>

#define E 0.000001 //erro
#define imax 50000 //número máximo de iterações

double cortaPalavra(char *str){
int i;
double aux;

for(i=0; i<strlen(str); i++) if(str[i] == 'e') str[i] = '\0';
aux = atof(str);

return aux;
}

int main(){
//variáveis do método
int nroLin, nroCol, nroNaoNulos;
double **A, *b, *x, *r, *d, *q, perNovo, perVelho, per0, alpha, betha,
elemento=0; //per = produto escalar do resíduo;
int cont; //variáveis de controle de iterações

//variáveis do arquivo
FILE *arq = fopen("bcsstm25.mtx", "r"); //abre o arquivo no modo
leitura
char *result, Linha[48], numero[20];
int linha=0, coluna=0;
int linhaArq=0;

//variáveis auxiliares
int i, j;
double temp=0;

//variável de controle de tempo
clock_t Ticks[2];
Ticks[0] = clock();

while(!feof(arq)){
fgets(Linha, 100, arq);
if(linhaArq == 1){
sscanf(Linha, "%d %d %d", &nroLin, &nroCol, &nroNaoNulos);

//alocação matriz A
A = (double**) malloc(nroLin *sizeof(double*));
for(i=0; i<nroLin; i++){
A[i] = (double*) malloc(nroCol * sizeof(double));
for(j=0; j<nroCol; j++) A[i][j] = 0;
}
//alocação do vetor b
```

```
b = (double*) malloc(nroCol * sizeof(double));
//alocação do vetor x
x = (double*) malloc(nroCol * sizeof(double));
//alocação do vetor r
r = (double*) malloc(nroCol * sizeof(double));
//alocação do vetor d
d = (double*) malloc(nroCol * sizeof(double));
//alocação do vetor q
q = (double*) malloc(nroCol * sizeof(double));
}else if(linhaArq > 1){
sscanf(Linha, "%d %d %s", &linha, &coluna, numero);

linha--;
coluna--;
elemento = cortaPalavra(numero);
A[linha][coluna] = elemento;
}

linhaArq++;
}
fclose(arq);

//definição dos valores do vetor b
for(i=0; i<nroCol; i++) b[i] = 1;
//definição dos valores do vetor x
for(i=0; i<nroCol; i++) x[i] = 0;

//INÍCIO DO MÉTODO - variável da quantidade de iterações, começa em
zero
cont = 0;

//cálculo do resíduo (r = b - Ax)
for(i=0; i<nroCol; i++){
r[i] = 0;
for(j=0; j<nroCol; j++) r[i] += A[i][j] * x[j];
}
for(i=0; i<nroCol; i++) r[i] = b[i] - r[i];

//primeira direção
for(i=0; i<nroCol; i++) d[i] = r[i];

//primeiro valor de perNovo
for(i=0; i<nroCol; i++) perNovo += r[i]*r[i];

per0 = perNovo;

temp = (E*E)*per0; //erro

while((cont< imax) && (perNovo>temp)){
//cálculo do vetor da matriz esparsa pelo resíduo
for(i=0; i<nroCol; i++){
q[i] = 0;
for(j=0; j<nroCol; j++) q[i] += A[i][j] * r[j];
}

temp=0;
```

```
//tamanho do passo (alpha = (perNovo / qd))
for(i=0; i<nroCol; i++) temp += d[i]*q[i];
alpha = perNovo/temp;

//aproximação de x (x = x + alpha*q)
for(i=0; i<nroCol; i++) x[i] = x[i] + (alpha*q[i]);

//teste de aproximação de resíduo
if(cont%50 == 0){
for(i=0; i<nroCol; i++){
r[i] = 0;
for(j=0; j<nroCol; j++) r[i] += A[i][j] * x[j];
}

for(i=0; i<nroCol; i++) r[i] = b[i] - r[i];
}else{
for(i=0; i<nroCol; i++) r[i] = r[i] - (alpha*q[i]);
}

//perVelho = perNovo
perVelho = perNovo;

//perNovo = rr
for(i=0; i<nroCol; i++) perNovo += r[i]*r[i];

//razão entre o produto escalar dos resíduos
betha = perNovo/perVelho;

//nova direção de pesquisa
for(i=0; i<nroCol; i++) d[i] = r[i] + (betha*d[i]);

cont++;
}

//desalocar ponteiros utilizados
free(A);
free(b);
free(x);
free(r);
free(d);
free(q);

//calculo do tempo decorrido da aplicação
Ticks[1]=clock();
double tempo = (((Ticks[1]-Ticks[0])*1000.0)/CLOCKS_PER_SEC);

printf("Vetor de resultados:\n");
for(i=0; i<nroCol; i++) printf("x[%d]: %f\n", i, x[i]);
printf("\n\nNumero de iteracoes: %d\n", cont);
printf("Tempo de execucao: %g ms.\n", tempo);

return 0;
}
```

B – Algoritmo do Gradiente Conjugado paralelo em OpenMP com linguagem C.

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<string.h>
#include<omp.h>

#define E 0.000001 //erro
#define imax 50000 //número máximo de iterações

double cortaPalavra(char *str){
int i;
double aux;

for(i=0; i<strlen(str); i++) if(str[i] == 'e') str[i] = '\0';
aux = atof(str);

return aux;
}

int main(){
//variáveis do método
int nroLin, nroCol, nroNaoNulos;
double **A, *b, *x, *r, *d, *q, perNovo, perVelho, per0, alpha, betha,
  elemento=0; //per = produto escalar do resíduo;
int cont; //variáveis de controle de iterações

//variáveis do arquivo
FILE *arq = fopen("bcsstk18.mtx", "r"); //abre o arquivo no modo
  leitura
char *result, Linha[48], numero[20];
int linha=0, coluna=0;
int linhaArq=0;

//variáveis auxiliares
int i, j;
double temp=0;

//variável de controle de tempo
clock_t Ticks[2];
Ticks[0] = clock();

while(!feof(arq)){
fgets(Linha, 100, arq);
if(linhaArq == 1){
sscanf(Linha, "%d %d %d", &nroLin, &nroCol, &nroNaoNulos);

//alocação matriz A
A = (double**) malloc(nroLin *sizeof(double*));
for(i=0; i<nroLin; i++){
A[i] = (double*) malloc(nroCol * sizeof(double));
for(j=0; j<nroCol; j++) A[i][j] = 0;
}
//alocação do vetor b
b = (double*) malloc(nroCol * sizeof(double));
```

```
//alocação do vetor x
x = (double*) malloc(nroCol * sizeof(double));
//alocação do vetor r
r = (double*) malloc(nroCol * sizeof(double));
//alocação do vetor d
d = (double*) malloc(nroCol * sizeof(double));
//alocação do vetor q
q = (double*) malloc(nroCol * sizeof(double));
}else if(linhaArq > 1){
sscanf(Linha, "%d %d %s", &linha, &coluna, numero);

linha--;
coluna--;
elemento = cortaPalavra(numero);

A[linha][coluna] = elemento;
}

linhaArq++;
}
fclose(arq);

//definição dos valores do vetor b
#pragma omp for
for(i=0; i<nroCol; i++) b[i] = 1;
#pragma omp barrier
//definição dos valores do vetor x
#pragma omp for
for(i=0; i<nroCol; i++) x[i] = 0;
#pragma omp barrier

//INÍCIO DO MÉTODO - variável da quantidade de iterações, começa em
  zero
cont = 0;

//cálculo do resíduo (r = b - Ax)
#pragma omp parallel shared(A, r, b, x) private(i, j)
{
#pragma omp for
for(i=0; i<nroCol; i++){
r[i] = 0;
for(j=0; j<nroCol; j++) r[i] += A[i][j] * x[j];
}
#pragma omp barrier
}
#pragma omp for
for(i=0; i<nroCol; i++) r[i] = b[i] - r[i];
#pragma omp barrier

//primeira direção
#pragma omp for
for(i=0; i<nroCol; i++) d[i] = r[i];
#pragma omp barrier

//primeiro valor de perNovo
#pragma omp for
```

```
for(i=0; i<nroCol; i++) perNovo += r[i]*r[i];
#pragma omp barrier

per0 = perNovo;

temp = (E*E)*per0; //erro

while((cont< imax) && (perNovo>temp)){
//cálculo do vetor da matriz esparsa pelo resíduo
#pragma omp parallel shared(A, r, q) private(i, j)
{
#pragma omp for
for(i=0; i<nroCol; i++){
q[i] = 0;
for(j=0; j<nroCol; j++) q[i] += A[i][j] * r[j];
}
#pragma omp barrier
}

temp=0;
//tamanho do passo (alpha = (perNovo / qd))
#pragma omp for
for(i=0; i<nroCol; i++) temp += d[i]*q[i];
#pragma omp barrier
alpha = perNovo/temp;

//aproximação de x (x = x + alpha*q)
#pragma omp for
for(i=0; i<nroCol; i++) x[i] = x[i] + (alpha*q[i]);
#pragma omp barrier

//teste de aproximação de resíduo
if(cont%50 == 0){
#pragma omp parallel shared(A, r, x) private(i, j)
{
#pragma omp for
for(i=0; i<nroCol; i++){
r[i] = 0;
for(j=0; j<nroCol; j++) r[i] += A[i][j] * x[j];
}
#pragma omp barrier
}
#pragma omp for
for(i=0; i<nroCol; i++) r[i] = b[i] - r[i];
#pragma omp barrier
}else{
#pragma omp for
for(i=0; i<nroCol; i++) r[i] = r[i] - (alpha*q[i]);
#pragma omp barrier
}

//perVelho = perNovo
perVelho = perNovo;

//perNovo = rr
#pragma omp for
```

```
for(i=0; i<nroCol; i++) perNovo += r[i]*r[i];
#pragma omp barrier

//razão entre o produto escalar dos resíduos
betha = perNovo/perVelho;

//nova direção de pesquisa
#pragma omp for
for(i=0; i<nroCol; i++) d[i] = r[i] + (betha*d[i]);
#pragma omp barrier

cont++;
}
free(A);
free(b);
free(x);
free(r);
free(d);
free(q);

Ticks[1]=clock();
double tempo = (((Ticks[1]-Ticks[0])*1000.0)/CLOCKS_PER_SEC);

printf("Vetor de resultados:\n");
for(i=0; i<nroCol; i++) printf("x[%d]: %f\n", i, x[i]);
printf("\n\nNumero de iteracoes: %d\n", cont);
printf("Tempo de execucao: %g ms.\n", tempo);

return 0;
}
```