Proceeding Series of the Brazilian Society of Applied and Computational Mathematics, v. 6, n. 1, 2018.

Trabalho apresentado no XXXVII CNMAC, S.J. dos Campos - SP, 2017.

Proceeding Series of the Brazilian Society of Computational and Applied Mathematics

Implementação Paralela de Núcleos Computacionais do Solver Orthomin

João P. K. Zanardi, Ítalo C. L. Nievinski[1]
Faculdade de Engenharia Mecânica, PPGEM, UERJ, Rio de Janeiro - RJ
Luiz Mariano Carvalho[2]
Instituto de Matemática e Estatística , UERJ, Rio de Janeiro - RJ

Resumo. Apresentamos uma implementação paralela eficiente em memória compartilhada de alguns núcleos computacionais do solver Orthomin. Os núcleos apresentados são o produto matriz vetor e o processo de ortogonalização de Gram-Schmidt. As implementações foram feitas em C++ utilizando diretivas OpenMP para a paralelização. Além disso, as implementações foram feitas visando as arquiteturas Intel Xeon e os aceleradores Intel Xeon Phi.

Palavras-chave. Produto Matriz-Vetor, Gram-Schmidt, Paralelismo de Grão Fino, Memória Compartilhada, Computação de Alto Desempenho.

1 Introdução

Discutimos a implementação de dois importantes núcleos de Álgebra Linear computacional do solver linear Orthomin [5]: o produto Matriz-Vetor e do método de Gram-Schmidt. O Orthomin é um solver linear baseado em subespaços de Krylov amplamente utilizado na indústria do petróleo e por isso é de extrema importância que sua construção seja eficiente. Além da sua aplicabilidade no Orthomin o produto Matriz-Vetor e o método de Gram-Schmidt são núcleos computacionais com várias aplicações e no atual cenário massivamente paralelo é crucial ter uma boa implementação de ambos em paralelo. As implementações foram feitas C++ com paralelismo de memória compartilhada utilizando OpenMP [1], focando sua implementação nas arquiteturas Intel Xeon e Xeon Phi. O desenvolvimento deste trabalho pode ser descrito a seguir, na seção 2 discutiremos o produto Matriz-Vetor. A seção 3 discutirá a implementação do método de Gram-Schmidt. Resultados dos experimentos numéricos podem ser vistos na seção 4. Por fim, as conclusões se encontram na seção 5.

2 O Produto Matriz-Vetor

O produto Matriz Vetor [3] é uma operação $y = Ax$, onde A é uma matriz esparsa e x é um vetor denso. A multiplicação matriz vetor é um importante núcleo em álgebra linear

[1]jpzanardi@gmail.com, italonievinski@gmail.com
[2]luizmc@ime.uerj.br

DOI: 10.5540/03.2018.006.01.0379

computacional, aparecendo frequentemente em métodos iterativos, tais como métodos de Krylov usados para a resolução de sistemas lineares de grande porte. Portanto, é crucial para a eficiência de um método iterativo uma boa implementação da multiplicação de uma matriz por um vetor.

Uma boa implementação deste núcleo depende do formato de armazenamento utilizado para salvar a matriz. Existem várias maneiras de se armazenar uma matriz, em nosso caso escolhemos o formato *Block Compressed Sparse Row* [3], essa escolha foi baseada na natureza do problema resolvido. Nosso problema é oriundo da simulação de reservatórios de petróleo estamos trabalhando com variáveis distintas em EDPs, após a discretização chegamos a uma matriz resultante em bloco, onde cada bloco tem o tamanho do número de variáveis, sendo assim conveniente utilizar esta estrutura em blocos para o armazenamento da matriz. O código do produto matriz vetor em formato BCRS com tamanho de bloco bs pode ser escrito em C++ como:

```
for( i = 0; i < m ; i++)
    for( j = row[i]; j < row[i+1]; j++)
        for( k = 0; k < bs; k++)
            for( l = 0; l < bs; i++)
                y[i*bs+j]+=A[j*bs*bs+k*bs+l]*x[col[j]+k];
```

A primeira otimização feita no código foi implemetá-lo como uma função *template* em relação ao tamanho do bloco, isto é, o compilador cria, em tempo de compilação, um código otimizado para um determinado tamanho de bloco, deixando o programa mais rápido.

O código foi paralelizado utilizando a seguinte instrução OpenMP

#pragma omp parallel for schedule(static, chunk)

onde *chunk* é a parte inteira da divisão do número de blocos linhas pelo número de *threads*. A instrução **schedule(static, chunk)** significa que cada thread vai computar *chunk* linhas. A escolha do *chunk* é importante para a redução do desbalanceamento de carga, isto é, evitar que algumas *threads* façam muito mais trabalhos que outras.

3 O Método de Gram-Schmidt

O método de Gram-Schmidt [2] é um processo de ortogonlização de um conjuntos de vetores pertencentes a um espaço vetorial com produto interno, em nosso caso utilizaremos o $\mathbb{R}^n$ com o produto interno usual. O método recebe um conjunto finito de vetores linearmente independentes $S = \{a_1, a_2, \ldots, a_n\}$ e retorna um conjunto $S' = \{q_1, q_2, \ldots, q_n\}$ tal que $\langle q_i, q_j \rangle = 0$ se e somente se $i \neq j$. Dentro do Orthomin o método de Gram-Schmidt é utilizado dentro de uma iteração de Arnoldi, isto é, o Gram-Schmidt para produzir uma sequência de vetores ortonormais $q_1, q_2, q_3, \ldots$, tais que qualquer conjunto de vetores $q_1, q_2, \ldots, q_n$ geram o espaço de Krylov K_n. O algoritmo para o método de Gram-Schmidt dentro do solver Orthomin pode ser escrito como:

DOI: 10.5540/03.2018.006.01.0379

```
1  Iniciando com um vetor ortonormal q_1
2
3  for k = 2:n
4      q_k = A*q_k-1
5      for j = 1:k-1
6          r_jk = <q_j,w>
7      end
8      for j = 1:k-1
9          w = w - r_jk*q_j
10     end
11     r_kk = ||w||
12     q_k = w/r_kk
```

onde A é a matriz do sistema linear.

Em nossa implementação as operações do tipo BLAS1 foram substituídas por operações do tipo BLAS2, mas especificamente o *loop* com produtos internos da linha 5 e o *loop* de *daxpy* da linha 8 foram retirados e em seu lugar fazemos um produto matriz-vetor denso, onde a matriz utilizada é a matriz cujas a colunas são $q_1, \ldots, q_j$. As linhas 11 e 12 também foram trocadas por uma rotina que normaliza os vetores q_i.

O produto matriz-vetor denso foi implementado trabalhando com a blocagem do vetor resultante, aumentando assim a localidade do acesso a este vetor. O tamanho do bloco foi escolhido de tal forma que o bloco coubesse inteiramente na memória *cache*. O código em C++ do produto de uma matriz densa A por um vetor x resultando em y pode ser visto abaixo.

```
1          int jo;
2          static const int JB = 8;
3
4      #pragma omp parallel for lastprivate(jo)
5          for(jo = 0; jo < m - JB; jo += JB)
6                  for(int i = 0; i < n; i++)
7              #pragma simd vectorlength(JB)
8                          for(int ji = 0; ji < JB; ji++)
9                              y[ji + jo] -= A[i*m + ji + jo]*x[i];
10     #pragma omp parallel for
11         for(int i = 0; i < n; i++)
12     #pragma simd vectorlength(JB)
13                 for(int ji = jo; ji < m; ji++)
14                     y[ji] -= A[i*m + ji]*x[i];
```

No código a constante JB é quem define o tamanho do bloco do vetor resultante y, no caso utilizamos 8 para XEon e 256 para Xeon Phi, estes valores são escolhidos pois a memória *cache* do Xeon e Xeon Phi tem capacidade para 8 e 256 *doubles* respectivamente. Os *loops* mais externos são paralelizados com a diretiva **# pragma omp parallel for**. O *loop* da linha 4 utiliza ainda a diretiva **lastprivate(jo)**, esta diretiva faz com que o valor da variável *jo* ao fim do *loop* seja o valor retornado pela última *thread*. A diretiva

DOI: 10.5540/03.2018.006.01.0379

#pragma simd vectorlenght(JB) é utilizada para que o compilador vetorize o último *loop*, assim cada iteração será equivalente a JB iterações escalares.

4 Resultados

Os testes para o produto matriz vetor foram realizados numa máquina com arquitetura *Sandy Bridge*, com 2 sockets e 8 cores em cada um, o acelerador utilizado foi o Knights Corner com 60 cores. Já os testes para o método de Gram-Schmidt foram realizados numa máquina com arquitetura *Ivy Bridge*, com 2 sockets e 10 cores em cada um, o acelerador utilizado também foi o Knights Corner de 60 cores. As especificações completas de cada máquina podem ser vistas na Tabela 1. Em nossos experimentos só utilizamos 1 socket e a performance foi mensurada e comparada com a largura de banda da máquina medida através do *stream bandwidth* da Universidade de Vírginia [4]. Nas tabelas, vamos nos referir ao Xeon como CPU e KNC para o Xeon Phi.

Tabela 1: Especificações da Máquina

	Intel Xeon	Intel Xeon	Intel Xeon Phi
Microarquitetura	Sandy Bridge E5-2670	Ivy Bridge	Kinghts Corner
Frequência	2.6 GHz	3 GHz	1.09GHz
#sockets	2	2	1
#cores	8	10	60
#threads por core	1	1	4
Stream Bandwidth	37 GB/s	40 GB/s	143.4GB/s

A Tabela 2 mostra os dados das matrizes testadas, onde n é o número de blocos linhas, nnz é o número de blocos não nulos e bs é o tamanho do bloco. Todas as matrizes testadas são matrizes reais de simulações de reservatórios de petróleo.

Tabela 2: Matrizes Testadas

Matriz	n	nnz	bs
1	66077	449376	4
2	182558	1223695	4
3	408865	2731921	4
4	617459	4379139	4

A Tabela 3 mostra o percentual de largura de banda alcançado pelos dois núcleos no Xeon e Xeon Phi, como pode ser visto o produto matriz-vetor alcançou perto de 90% da largura de banda tanto em Xeon e Xeon Phi exceto para a matriz 1 em Xeon Phi. Já o Gram-Schmidt teve resultados próximos a 90% da largura de banda no Xeon e os resultados variaram da ordem 70 à 90% no Xeon Phi. A Figura 1 mostra o percentual de tempo gasto por cada núcleo na resolução completa dos sistema linear utilizando o Orthomin em Xeon e Xeon Phi, onde `Precond` se refere a construção do precondicionador, `Aplic. Precond` é o tempo gasto com a aplicação do precondicionador, `Matriz-Vetor`

DOI: 10.5540/03.2018.006.01.0379

e `Gram-Schmidt` são os percentuais dos tempos gastos com o produto Matriz-Vetor e o método de Gram-Schmidt, e `Outros` é o tempo gasto com partes pequenas do solver, como produtos internos, normas, *daxpy*, etc. Por esta figura, fica clara a importância do Gram-Schmidt dentro do solver Orthomin, uma vez que ele representa a maior fatia de tempo no Xeon e a segunda maior no Xeon Phi. O produto matriz vetor também apresenta uma boa fração do tempo, sendo inclusive o segundo núcleo mais demorado para o Xeon.

Tabela 3: Resultados

Matriz	Xeon	Xeon Phi	Xeon	Xeon Phi
1	94%	76%	91%	60%
2	92%	87%	89%	78%
3	87%	90%	88%	85%
4	87%	84%	88%	88%

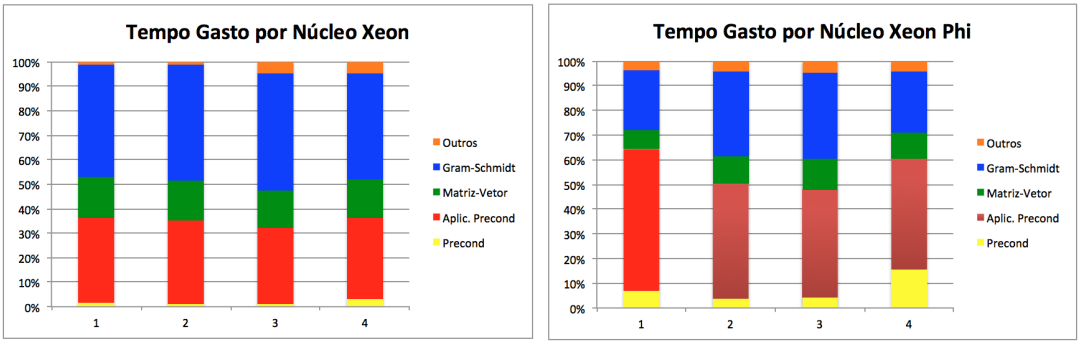

Figura 1: Percentual de Tempo Gasto por Núcleo do Orthomin.

5 Conclusão

O produto Matriz-Vetor atingiu praticamente 90% da largura de banda, tanto para Xeon quanto para Xeon Phi, na maioria dos casos testados. Este resultado é muito bom uma vez que tanto o produto Matriz-Vetor quanto o Gram-Schmidt tem a performance limitados pela largura de banda. Além disso, o uso de funções *templates* deixam o código mais simples, legível e rápido. A versão clássica blocada e com BLAS2 do método de Gram-Schmidt apresentou resultados bons para o Xeon e razoáveis para o KNC, sendo eficiente o uso do Xeon Phi apenas para as maiores matrizes. O Gram-Schmidt revelou-se o núcleo computacional que consome a maior parte do solver Orthomin no Xeon e o segundo maior Xeon Phi, sendo assim de suma importância ter uma implementação eficiente deste núcleo. O produto Matriz-Vetor consome uma boa porção do tempo do Orthomin, chegando a casa dos 30% no Xeon e 15% em Xeon Phi. Vale destacar que existem precondicionadores que necessitam apenas do produto Matriz-Vetor em sua aplicação, sendo assim o percentual gasto por este núcleo dentro do solver pode ser ainda maior.

DOI: 10.5540/03.2018.006.01.0379

Referências

[1] R.Chandra. *Parallel Programing in OpenMP*. Morgan Kaufman, 2001.

[2] C. Van Loan, G. Golub. *Matrix Computations*. Johns Hopkins studies in mathematical sciences, 1996.

[3] X.Liu, M. Smeliansky, E. Chow, P. Dubey. Efficient Sparse Matrix-Vector Multiplication on x86-Based Many-Core Processors, *Proceedings of the 27th international ACM conference on International conference on supercomputing*, 273-282, 2013.

[4] J.McCaplin. Virginia Stream Benchmark,*https://www.cs.virginia.edu/stream/*.

[5] P. Vinsome. Orthomin, an iterative method for solving sparse sets of simultaneous linear equations, *SPE Symposium on Numerical Simulation of Reservoir Performance. Society of Petroleum Engineers* , 1976.

DOI: 10.5540/03.2018.006.01.0379