Proceeding Series of the Brazilian Society of Computational and Applied Mathematics

Técnicas de otimização combinatória aplicadas a um precondicionador paralelo baseado no algoritmo SPIKE

Brenno A. Lugon[1], Leonardo M. Lima[2], Marcelo T. P. Carrion[1]
Lucia Catabriga[1], Maria Cristina Rangel[1], Maria Claudia S. Boeres[1]
Departamento de Informática, UFES, Vitória, ES

Abstract. Neste trabalho, utilizamos o algoritmo paralelo híbrido SPIKE como um precondicionador para um método iterativo não-estacionário combinando as arquiteturas de memória distribuída e compartilhada MPI/OpenMP. A fim de obter um bom precondicionador resolvemos um conjunto de problemas combinatórios como reordenamento, particionamento, *matching* e o problema quadrático da mochila. Apresentamos os resultados avaliando o *speedup* e escalabilidade em sistemas resultantes de formulações de elementos finitos.
Keywords. SPIKE, algoritmo paralelo híbrido, problemas combinatórios

1 Introdução

Com o crescimento na área de programação paralela e evolução das arquiteturas paralelas, uma tendência é o uso de uma abordagem híbrida MPI/OpenMP. Neste paradigma de programação, multicomputadores exploram o paralelismo usando memória distribuída enquanto multiprocessadores garantem as vantagens do uso de memória compartilhada. A Figura 1 exibe o esquema arquitetural dessa abordagem híbrida: diferentes *ranks* (Memória + CPU com 4 *cores*) se comunicam usando MPI via rede, enquanto as *threads* — ou *cores* — em cada *rank* se comunicam usando OpenMP.

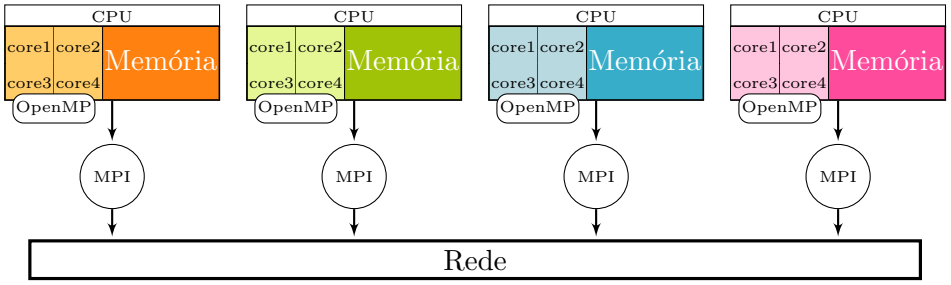

Figura 1: Abordagem híbrida MPI e OpenMP

[1]{blugon,mcarrion,luciac,crangel,boeres}@inf.ufes.br
[2]lmuniz@ifes.edu.br

Para a resolução de sistemas lineares de grande porte, [10] sugere a utilização de métodos iterativos baseados em projeções de subespaços de Krylov por suas boas propriedades numéricas e computacionais. Além disso, o uso de precondicionadores acelera a convergência e consequentemente melhora a eficiência desses métodos na obtenção da solução. Em aplicações paralelas, o algoritmo híbrido SPIKE [8, 9], criado inicialmente para resolver um sistema linear, pode também ser usado como precondicionador para o método iterativo. Ele é dito um método híbrido por ser capaz de tirar vantagem da robustez dos métodos diretos e do baixo custo computacional dos métodos iterativos.

Este artigo, cujo objetivo é a implementação de um método GMRES híbrido precondicionado com SPIKE considerando técnicas de otimização, está organizado da seguinte forma: na seção 2, explicamos o funcionamento do algoritmo SPIKE e as adaptações — e estratégias combinatórias — necessárias para usá-lo como precondicionador; a seção 3 apresenta os resultados obtidos de testes computacionais, com conclusões e trabalhos futuros sendo abordados na seção 4; por fim, seguem as seções de agradecimentos e referências bibliográficas.

2 O algoritmo SPIKE

O algoritmo paralelo híbrido SPIKE tem como objetivo encontrar a solução de um sistema linear da forma $Ax = f$ usando computação paralela. Considere $A = \{a_{ij}\}_{n\times n}$ como sendo uma matriz esparsa com estrutura de banda, i.e., com os elementos não-nulos dispostos próximos à diagonal principal. Considere também A como sendo diagonal dominante, ou seja, $\sum_{i\neq j} |a_{ij}| \leq |a_{ii}|, \forall i$. O algoritmo SPIKE se baseia na decomposição $A = D \times S$, onde D é uma matriz bloco diagonal e $S = D^{-1}A$ é chamada de matriz de *spikes*. Portanto, resolver o sistema $Ax = f$ recai em resolver os sistemas $Dg = f$ e $Sx = g$. A Figura 2 mostra como é feita essa decomposição para $p = 3$ processadores.

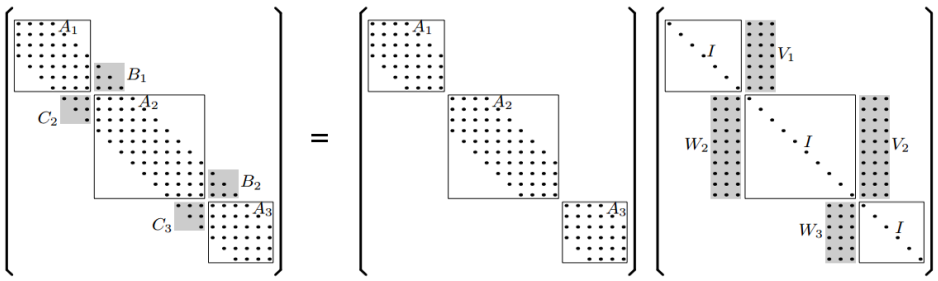

Figura 2: Decomposição $A = D \times S$

As matrizes $\mathbf{B}_i$ e $\mathbf{C}_{i+1}$, $i = 1, ..., p-1$, são chamadas de blocos de acoplamento e possuem dimensão $\mathbf{k} \times \mathbf{k}$, onde $\mathbf{k}$ é a largura de banda de A. As matrizes $\mathbf{V}_i$ e $\mathbf{W}_{i+1}$, $i = 1, ..., p-1$, são chamadas de *spikes* e formam a matriz de *spikes* S. De forma geral, as etapas do algoritmo SPIKE são:

1. Particionar o sistema original em vários *ranks* (ou processadores).
2. Aplicar a fatoração **LU** em cada bloco diagonal A_i.

3. Construir a matriz de *spikes* S resolvendo:

$$\begin{aligned} \mathbf{L}_1\mathbf{U}_1[\mathbf{V}_1, \mathbf{g}_1] &= \left[\begin{pmatrix} 0 \\ \mathbf{B}_1 \end{pmatrix}, \mathbf{f}_1\right] \\ \mathbf{L}_i\mathbf{U}_i[\mathbf{W}_i, \mathbf{V}_i, \mathbf{g}_i] &= \left[\begin{pmatrix} \mathbf{C}_i \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \mathbf{B}_i \end{pmatrix}, \mathbf{f}_i\right] \\ \mathbf{L}_p\mathbf{U}_p[\mathbf{W}_p, \mathbf{g}_p] &= \left[\begin{pmatrix} \mathbf{C}_p \\ 0 \end{pmatrix}, \mathbf{f}_p\right] , \end{aligned} \tag{1}$$

onde $1 < i < p$ sendo p o número de processadores.

4. Montar e resolver o sistema reduzido $Sz = g$:

$$\begin{bmatrix} I & \mathbf{V}_1^{(b)} & & \\ \mathbf{W}_i^{(t)} & I & & \mathbf{V}_i^{(t)} \\ \mathbf{W}_i^{(b)} & & I & \mathbf{V}_i^{(b)} \\ & & \mathbf{W}_p^{(t)} & I \end{bmatrix} \begin{bmatrix} z_1^{(b)} \\ z_i^{(t)} \\ z_i^{(b)} \\ z_p^{(t)} \end{bmatrix} = \begin{bmatrix} g_1^{(b)} \\ g_i^{(t)} \\ g_i^{(b)} \\ g_p^{(t)} \end{bmatrix} , \tag{2}$$

onde $\mathbf{V}_i^{(b)}$,$\mathbf{W}_i^{(b)}$ e $\mathbf{V}_i^{(t)}$,$\mathbf{W}_i^{(t)}$ são a parte inferior e superior de tamanho $\mathbf{k} \times \mathbf{k}$ das matrizes $\mathbf{V}_i$ e $\mathbf{W}_i$, respectivamente. Da mesma forma, $z_i^{(b)}$, $g_i^{(b)}$ e $z_i^{(t)}$, $g_i^{(t)}$ são a parte inferior e superior de tamanho $\mathbf{k}$ dos vetores z_i e g_i, respectivamente.

5. Uma vez que o sistema S foi resolvido, a solução final pode ser obtida efetuando:

$$\begin{aligned} x_1 &= g_1 - V_1 z_2 \\ x_i &= g_i - W_i z_{i-1} - V_i z_{i+1} \qquad \text{para } i = 2, ..., p-1 \\ x_p &= g_p - W_p z_{p-1} \ . \end{aligned} \tag{3}$$

Com o objetivo de melhorar a eficiência — reduzindo os cálculos computacionais envolvidos na decomposição $\mathbf{LU}$ —, podemos calcular apenas as partes superiores e inferiores de tamanho $\mathbf{k} \times \mathbf{k}$ das matrizes $\mathbf{W}_i$, $\mathbf{V}_i$ e do vetor g_i. Assim, a solução final é calculada por:

$$\begin{aligned} \mathbf{L}_1\mathbf{U}_1 x_1 &= f_1 - B_1 z_2 \\ \mathbf{L}_i\mathbf{U}_i x_i &= f_i - C_i z_{i-1} - B_i z_{i+1} \qquad \text{para } i = 2, ..., p-1 \\ \mathbf{L}_p\mathbf{U}_p x_p &= f_p - C_p z_{p-1} \ . \end{aligned} \tag{4}$$

2.1 O algoritmo SPIKE como precondicionador

O algoritmo SPIKE pode ser facilmente usado como um precondicionador dentro de um método iterativo não-estacionário como o GMRES ou BiCGStab [10]. Nesse caso, a matriz precondicionadora é dada por $M = \tilde{D}\tilde{S}$ da decomposição do SPIKE. A mudança com relação ao algoritmo original se dá na montagem do sistema reduzido $Sz = g$ na etapa 4, como mostra a equação (5). Agora, podemos desconsiderar as matrizes $\mathbf{V}_i^{(t)}$ e $\mathbf{W}_i^{(b)}$ — e também $\mathbf{k}$ pode ser menor que a largura de banda de A — já que o precondicionador representa apenas uma aproximação $\bar{x}$ da solução final. O Algoritmo 1 mostra o laço principal do método iterativo com precondicionador SPIKE.

$$\begin{bmatrix} I & \mathbf{V}_i^{(b)} \\ \mathbf{W}_{i+1}^{(t)} & I \end{bmatrix} \begin{bmatrix} z_i^{(b)} \\ z_{i+1}^{(t)} \end{bmatrix} = \begin{bmatrix} g_i^{(b)} \\ g_{i+1}^{(t)} \end{bmatrix} \tag{5}$$

DOI: 10.5540/03.2015.003.02.0024

Algoritmo 1: Laço principal do método iterativo com precondicionador SPIKE

1 $z = Av$ `/* Realizar o produto matriz-vetor em paralelo */`

2 $M\bar{x} = z$ `/* Resolver a etapa do precondicionador, com` $M = \tilde{D}\tilde{S}$ `*/`

Neste trabalho, as etapas de cálculo dos fatores **LU** foram feitas usando o software PARDISO [2, 12, 13] de resolução de sistemas lineares via métodos diretos usando memória compartilhada. De maneira eficiente, o PARDISO é capaz de resolver sistemas com múltiplos vetores de termos independentes, além de calcular somente as partes inferiores e superiores da solução de um sistema. Chamada de PSPIKE [3], a junção PARDISO + SPIKE é bastante eficiente na exploração do paralelismo MPI/OpenMP.

O uso do PARDISO não é a única maneira de se obter somente as partes inferiores e superiores das matrizes $\mathbf{W}_i$, $\mathbf{V}_i$ e do vetor g_i. Uma outra abordagem é o uso da estratégia **LU**/**UL**, onde se obtém $\mathbf{V}_i^b$ usando a fatoração **LU** e $\mathbf{W}_i^t$ através da fatoração **UL**. Dessa forma, reduzem-se os custos computacionais pois não é necessário calcular todos os coeficientes de cada fatoração [5].

2.2 Estratégias combinatórias

Para uma matriz A qualquer ser resolvida pelo SPIKE, é necessário transformá-la — através de permutações de linhas e colunas — em uma matriz com estrutura de banda, tal que suas entradas sejam cobertas pelas blocos diagonais A_i e pelos blocos de acoplamento B_i e C_{i+1}. Um bom precondicionador M deve contemplar nesses blocos os elementos não-nulos de maior valor absoluto. De forma geral, quatro estratégias de otimização combinatória são aplicadas na matriz, visando satisfazer quatro objetivos principais:

(i) Tornar a matriz original em uma matriz com estrutura de banda;

(ii) Obter os blocos diagonais;

(iii) Garantir a não-singularidade de cada bloco diagonal;

(iv) Mover elementos mais significativos para os blocos de acoplamento.

Cada objetivo é modelado como um problema combinatório, como mostrado a seguir.

2.2.1 Reordenamento

O reordenamento de matrizes esparsas é um problema bastante estudado e, no contexto deste trabalho, seu objetivo é transformar uma matriz geral em uma matriz com estrutura de banda, aproximando os elementos não-nulos da diagonal principal. Ele pode ser visto como um problema de rerrotulação de grafos, para o qual adotamos o algoritmo *Reverse Cuthill-McKee* (RCM) [1] pelos bons resultados presentes na literatura.

2.2.2 Particionamento

Nesse problema, desejamos atribuir linhas/colunas consecutivas da matriz a diferentes processadores, i.e., determinar as matrizes A_i, equilibrando o número de elementos não-nulos em cada um deles. Esse particionamento é modelado como um problema específico, chamado de *Chains-on-chains partitioning* (CCP) [7]. O CCP pode ser resolvido em tempo polinomial, e o algoritmo *MinMax* [4] é empregado para obter a solução exata.

DOI: 10.5540/03.2015.003.02.0024

2.2.3 *Matching*

O problema de estabelecer que os blocos diagonais A_i sejam não-singulares e, portanto, possam ser resolvidos utilizando a fatoração **LU** em cada processador, pode ser interpretado como um problema de *matching* valorado em grafos. O objetivo é permutar linhas no bloco de modo que elementos de grande valor absoluto fiquem na diagonal. Isso equivale a obter, no grafo bipartido que representa A_i, um *matching* perfeito de valor máximo, com o *Successive Shortest Path algorithm* [6] sendo usado para essa tarefa.

2.2.4 Problema quadrático da mochila

Consiste em obter os blocos de acoplamento B_i e C_{i+1} de tamanho $\mathbf{k}\times\mathbf{k}$, através de uma permutação simétrica (para não alterar os efeitos da etapa anterior) da matriz. De modo geral, permutam-se linhas e colunas de forma a concentrar elementos mais significativos em módulo — localizados nas submatrizes definidas entre os blocos diagonais A_i — dentro dos blocos de acoplamento. A heurística *DeMin* proposta por [11] é aplicada para realizar tal reordenamento.

3 Testes computacionais

Foi realizado um conjunto de testes com matrizes derivadas de problemas de elementos finitos, como mostra a Tabela 1. Os testes foram executados no *cluster* Enterprise 3 do laboratório LCAD na Ufes. Ele possui 28 máquinas quad-core Intel 2 Q6600 (116 *cores*) com 2.4GHz de frequência de *clock*, 4MB L2 e 4GB of DRAM, interconectadas por um *switch* 48-Port 4200G 3COM Gigabit Ethernet. Os códigos em C foram compilados com gcc 4.4.7, usando Open MPI 1.6.2, no sistema operacional CentOS 6.5. A versão da biblioteca PARDISO utilizada foi `pardiso500-MPI-GNU450-X86-64`.

matriz	dimensão (n)	não-nulos (nnz)	área de aplicação
mesh_1	1.043.474	7.294.192	Elementos Finitos 2D
cube_1	83.987	1.277.909	Elementos Finitos 3D

Tabela 1: Características das matrizes-teste

		GMRES	P_SGMRES, $\mathbf{k}=50$			P_SGMRES, $\mathbf{k}=100$		
			OpenMP			OpenMP		
	MPI		1	2	4	1	2	4
mesh_1	2	2503.06	123.68	94.91	89.05	132.53	103.54	95.65
	4	1427.71	89.74	70.60	65.86	101.44	80.24	75.55
	8	865.57	50.05	39.65	37.18	58.21	47.31	44.27
	16	541.26	27.49	25.86	20.65	38.22	35.47	30.40
cube_1	2	2714.71	38.53	24.07	19.50	47.74	29.20	22.88
	4	1945.78	13.14	8.92	7.79	18.15	12.11	10.22
	8	1832.29	5.79	4.38	4.23	8.47	6.58	5.91
	16	1538.70	3.64	3.01	2.85	5.46	4.93	4.42

Tabela 2: Diferentes tempos (em segundos) para resolver cada matriz-teste

DOI: 10.5540/03.2015.003.02.0024

Os tempos computacionais, em segundos, de testes preliminares para resolução dos sistemas oriundos das matrizes-teste são exibidos na Tabela 2. A coluna GMRES mostra o caso não-precondicionado, enquanto as colunas rotuladas com P$_S$GMRES, $\mathbf{k} = 50$ e P$_S$GMRES, $\mathbf{k} = 100$ se referem aos testes precondicionados com SPIKE, sendo $\mathbf{k}$ o tamanho dos blocos de acoplamento arbitrado em cada caso. Para cada matriz, as diferentes linhas indicam quantidades distintas de *ranks* MPI (descritas na coluna MPI) utilizados na parte de computação distribuída, enquanto as colunas identificadas com 1, 2 e 4 consideram o correspondente número de *threads* OpenMP usadas pelo PARDISO. Em ambas as matrizes, provou-se ser mais vantajoso combinar o maior número de *ranks* ao maior número de *threads*, ou seja, 16 *ranks* MPI e 4 *threads* OpenMP.

A Figura 3 apresenta os gráficos dos *speedups* referentes aos testes realizados com as matrizes-teste somente considerando $\mathbf{k} = 50$ (que obteve melhores resultados). O *speedup* alcançado por mesh_1 é bem parecido ao serem consideradas 1, 2 ou 4 *threads* OpenMP, o que sugere uma certa escalabilidade. Já o gráfico da matriz cube_1 apresenta alguns pontos de *speedup* superlinear, que possivelmente não seria sustentado ao se empregar mais *ranks* MPI como a própria curva sugere. O tamanho de cube_1 e sua forma, ou seja, menor e mais densa, podem haver influenciado a hierarquia da memória *cache*.

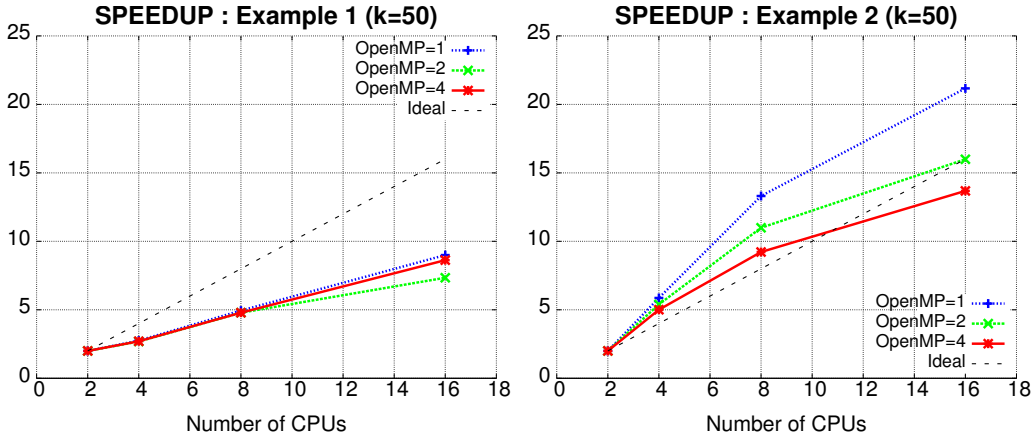

Figura 3: *Speedup* das matrizes mesh_1 *(à esquerda)* e cube_1 *(à direita)*

4 Conclusões e trabalhos futuros

O desempenho do precondicionador SPIKE depende fortemente da qualidade de transformação da matriz original A através das operações combinatórias apresentadas, e especialmente da escolha do tamanho $\mathbf{k}$. Quando $\mathbf{k}$ é maior, o número de iterações do método precondicionado tende a diminuir, uma vez que mais elementos da matriz transformada estão sendo considerados. Entretanto, essa diminuição raramente conduz a uma redução do tempo computacional total, pois as operações de decomposição $\mathbf{LU}$ e comunicações MPI relacionadas aos blocos $\mathbf{V}_i$ e $\mathbf{W}_i$ ficam mais custosas.

Como trabalhos futuros, propõe-se uma implementação própria das funções da biblioteca PARDISO inseridas no contexto do algoritmo SPIKE. Além disso, pretende-se utilizar esse precondicionador em plataformas de computação de larga escala e aplicar uma abordagem paralela às técnicas combinatórias aqui empregadas na forma sequencial.

DOI: 10.5540/03.2015.003.02.0024

5 Agradecimentos

Os autores agradecem às agências de fomento FAPES, CAPES e CNPq o apoio recebido através de bolsas de estudo e suporte financeiro na realização de projetos de pesquisa.

Referências

[1] E. Cuthill and J. McKee, Reducing the bandwidth of sparse symmetric matrices, Proceedings of the 1969 24th National Conference, 157–172, (1969).

[2] A. Kuzmin, M. Luisier and O. Schenk, Fast methods for computing selected elements of the greens function in massively parallel nanoelectronic device simulations, Lecture Notes in Computer Science, vol. 8097, 533–544, (2013).

[3] M. Manguoglu, A. Sameh and O. Schenk, Pspike: A parallel hybrid sparse linear system solver, Lecture Notes in Computer Science, vol. 5704, 797–808, (2009).

[4] F. Manne and T. Sørevik, Optimal partitioning of sequences, Journal of Algorithms, vol. 19, 235–249, (1995).

[5] K. Mendiratta, A banded spike algorithm and solver for shared memory architectures, Master's thesis, (2011).

[6] J. B. Orlin and Y. Lee, Quickmatch-a very fast algorithm for the assignment problem, Working papers 3547-93., MIT, Sloan School of Management, (1993).

[7] A. Pinar and C. Aykanat, Fast optimal load balancing algorithms for 1d partitioning, Journal of Parallel and Distributed Computing, vol. 64, 974–996, (2004).

[8] E. Polizzi and A. Sameh, A parallel hybrid banded system solver: The spike algorithm, Parallel Computing, vol. 32, 177–194, (2006).

[9] E. Polizzi and A. Sameh, SPIKE: A parallel environment for solving banded linear systems, Computers & Fluids, vol. 36, 113–120, (2007).

[10] Y. Saad, Iterative Methods for Sparse Linear Systems, SIAM, Philadelphia, 2nd ed., (2003).

[11] M. Sathe, O. Schenk, B. Uçar and A. Sameh, A Scalable Hybrid Linear Solver Based on Combinatorial Algorithms, Combinatorial Scientific Computing, Chapman-Hall/CRC Computational Science, 95–128, (2012).

[12] O. Schenk, M. Bollhöfer and R. A. Römer, On large-scale diagonalization techniques for the anderson model of localization, SIAM Review, vol. 50, 91–112, (2008).

[13] O. Schenk, A. Wächter and M. Hagemann, Matching-based preprocessing algorithms to the solution of saddle-point problems in large-scale nonconvex interior-point optimization, Computational Optimization and Applications, vol. 36, 321–341, (2007).

