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

Construção e aplicação de precondicionador ILUk por blocos com paralelismo em memória compartilhada.

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

Resumo. Apresentamos a implementação paralela, em memória compartilhada, do precondicionador BILUk para matrizes BCSR, através do OpenMP, utilizando a técnica de escalonamento por níveis na construção numérica e na aplicação.

Palavras-chave. paralelismo em memória compartilhada, solver triangular, precondicionadores, núcleos de cálculos intensivos, escalonamento por níveis, matrizes esparsas em bloco

1 Introdução

Discutimos a construção e a aplicação de um precondicionador ILU (fatoração LU incompleta) por blocos, com preenchimento por níveis (BILUk) [5] e com paralelismo de memória compartilhada, utilizando o OpenMP. Baseamo-nos na técnica de escalonamento por níveis (level scheduling) [3], focando na sua implementação em CPU e MIC (Many Integrated Core Architecture) da Intel, cujo nome comercial é Xeon Phi, em particular nesse trabalho testamos o KNC (Knights Corner). Nosso contexto de interesse é a simulação da explotação de petróleo, onde um sistema de equações diferenciais parciais não-lineares é discretizado, dando origem a um sistema de equações algébricas não-lineares, que é resolvido por um método de Newton. Em cada passo desse método, temos que tratar sistemas lineares de grande porte, com matrizes esparsas tendo pequenos blocos densos relativos a cada nó da malha de discretização, através de métodos iterativos de Krylov. Nesse caso, podem ser resolvidos centenas de sistemas lineares que manterão as mesmas esparsidade e estrutura da matriz original, apesar de sofrerem alterações em suas entradas numéricas. Nosso objetivo é apresentar um método que leve em conta essas características e, por isso, possa se concentrar na paralelização da fatoração numérica e da aplicação do precondicionador. Além dessa breve introdução, essa contribuição apresenta, em sua segunda seção, a técnica utilizada para a paralelização do algoritmo; enquanto que na seção 3 mostramos detalhes da implementação do BILUk. A seção 4 trata do solver triangular, a rotina que aplica o precondicionador em cada iteração do solver linear. As seções 5 e 6 apresentam os

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

resultados dos testes e comentários sobre os resultados. Finalmente, na seção 7, indicamos algumas conclusões e trabalhos futuros.

2 Paralelismo - Escalonamento por Níveis

O paralelismo neste precondicionador é obtido ao explorarmos uma possível quantidade reduzida de dependências entre as linhas da matriz, devida à sua esparsidade. Utilizamos o grafo de dependências para subdividir o problema em subgrupos independentes, onde os cálculos podem ser feitos em paralelo. Os subgrupos são chamados de níveis e são numerados de acordo. Cada nível é formado por linhas da matriz que são independentes entre si, e possuem dependências apenas de dados calculados em níveis anteriores. Os níveis devem ser resolvidos em sucessão para obtenção ordenada dos dados necessários para os níveis subsequentes. Esta técnica é conhecida como escalonamento por níveis [3].

A dependência por níveis pode ser igualmente observada e aproveitada na aplicação do precondicionador. Além disso, as dependências para a solução do sistema associado à matriz triangular inferior L, do precondicionador incompleto, possuem as mesmas relações presentes na sua construção.

3 BILUk Setup

A Figura 1 apresenta o esquema de construção do precondicionador. Note que introduzimos o passo de escalonamento por níveis entre as duas fases da construção do ILUk.

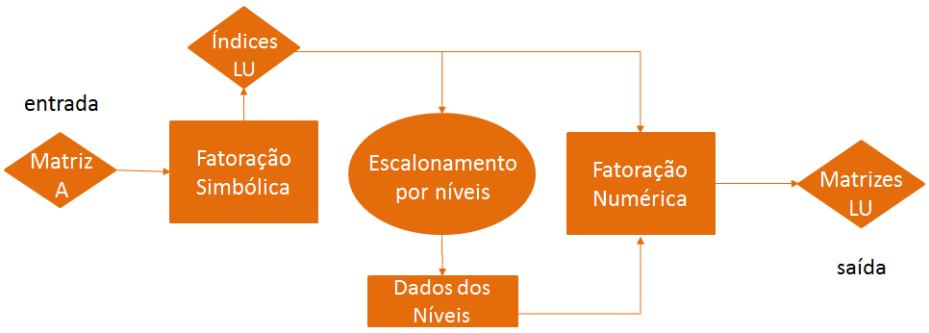

Figura 1: Esquema de construção ILUk paralelo

A fase simbólica irá obter todas as localizações das entradas não nulas resultantes da fatoração LU incompleta, sem, no entanto, calcular o valor numéricos dessas entradas. Isto é feito a partir de um algoritmo de fatoração LU, ignorando porém a produção de novas entradas não nulas cujo nível é maior do que k e sem realizar os cálculos numéricos, ver [5] para detalhes.

Durante uma simulação onde o padrão de esparsidade da matriz é mantido constante durante vários passos das iterações não-lineares, essa fatoração simbólica será realizada apenas uma vez durante todos esses passos, logo a otimização desta etapa produzirá poucos

DOI: 10.5540/03.2018.006.01.0343

ganhos. Como os casos de interesse, em especial em simulações da explotação de reservatórios de petróleo, possuem esta característica, sendo assim a otimização deste passo não foi o foco desse trabalho. Foi desenvolvido, porém, uma versão paralela para a fatoração simbólica com k = 1.

Após a fase simbólica, como pode ser acompanhado na Figura 1, é preciso realizar o escalonamento por níveis. Iremos extrair os níveis, a partir da esparsidade de L já obtida, para só então realizarmos a fase numérica em paralelo. O Algoritmo 1 apresenta a fase de construção dos níveis.

Algoritmo 1: Construção dos Níveis

```
1 nv = 0;
2 linhas_restantes <- linhas de L
3 while(linhas_restantes != vazio)
4    for(cada linha i em linhas_restantes)
5       if(nao existe nao nulo (i,j), j > i, em linhas_restantes)
6          associa a linha i com o nivel nv;
7          remove a linha i de linhas_restantes;
8    nv++;
```

Uma vez que o escalonamento por níveis é calculado a partir da fatoração simbólica, não precisaremos calculá-lo novamente enquanto não houver mudança na fatoração simbólica. No caso de interesse, este passo também será realizado poucas vezes durante uma simulação, e sua otimização apresenta pouca relevância.

A fase numérica paralela da construção do precondicionador é descrita no Algoritmo 2. As linhas 8 e 10 são os núcleos de processamento do algoritmo, realizando os produtos de blocos densos $A = B * C$ e $A = A + B * C$. Para tanto, utilizamos uma biblioteca com código aberto, especializada no produto de matrizes pequenas e otimizada para arquitetura Intel (X86) [1].

A linha 12 realiza um inversão do pivô que será armazenado na diagonal da matriz U. O pivô também é um bloco denso e, para sua inversão, utilizamos uma eliminação gaussiana com pivotamento parcial.

Os laços nas linhas de cada nível são paralelizados via OpenMP, como pode ser visto na linha 2 do Algoritmo 2. Quanto maior o número de linhas por nível e menor a quantidade de níveis, mais eficiente se torna a paralelização do trabalho.

Algoritmo 2: ILUK Numérico

```
1  for(cada nivel)
2     #pragma omp parallel for
3        for(cada linha no nivel)
4           inicializa vetor de trabalho
5           insere zeros em L e U
6           copia os elementos de A para L e U
7           for(cada elemento antes da diagonal na linha)
8              Multiplica o elemento pelo pivo (invertido)
9              for(cada elemento na linha do pivo)
10                Combinacao numerica das linhas (DGEMMs)
11       reinicia o vetor de trabalho
```

DOI: 10.5540/03.2018.006.01.0343

```
12 ||       inverte o elemento da diagonal de U
```

4 Aplicação - Solver Triangular

A aplicação do precondicionador é feita a cada iteração no método de Krylov através de $v = U\backslash(L\backslash r)$[3]. Serão realizados então um solver para frente e um solver para trás, observando que L é triangular unitária (apenas entradas iguais a um, na diagonal principal).

Os solvers triangulares também serão paralelizados utilizando o esquema de escalonamento por níveis como é proposto em [2]. Observe o uso dos níveis no Algoritmo 3, que mostra $x = U\backslash b$. Note que, para a triangular unitária, o passo da linha 7 do Algoritmo 3 não é realizado.

Algoritmo 3: Solver Triangular

```
1 || // Resolve U x = b
2 || for(cada nivel)
3 ||    #pragma omp parallel for
4 ||       for (cada linha i de n a 1)
5 ||          for (cada bloco elemento j nao nulo da linha i)
6 ||             x[i] -= U[i,j]*x[j];
7 ||       x[i] = U[i,i]*x[i];
```

As linhas 6 e 7 são pequenos produtos matriz × vetor densos, tratam-se dos núcleos de aplicação do precondicionador. Para otimizar estas rotinas, utilizamos vetores auxiliares e laços de tamanho fixo de acordo com o tamanho do bloco através do uso de templates do C++ [4]. Para melhorar a performance, após o cálculo do BILUk, as linhas das matrizes L e U são reordenadas de acordo com os níveis, melhorando o padrão de acesso ao dado durante os solvers triangulares na aplicação do precondicionador.

5 Descrição dos testes

Foram realizados dois grupos de experimentos: o primeiro de performance isolada do solver triangular, o segundo de performance isolada do BILUK numérico. Em todos os casos foram realizadas 10 rodadas do teste, e os valores apresentados são suas médias aritméticas.

Foram utilizados três ambientes computacionais diferentes: Xeon Sandy Bridge de 2.6 GHz com 8 cores e largura de banda de 37.05 GB/s; Xeon Ivy Bridge de 3.0 GHz com 10 cores e largura de banda de 40 GB/s; e KNC de 1.09 GHz e 60 cores e largura de banda de 143 GB/s.

O primeiro grupo de experimentos foi realizado no primeiro e terceiro ambientes, enquanto o segundo grupo foi realizado no segundo e terceiro ambientes.

[3] Utilizamos a notação do Matlab/Octave, onde \ representa a aplicação de um solver direto.

[4] Templates é uma funcionalidade de C++ permitindo funções operarem com tipos genéricos, assim podemos utilizar diferentes tamanhos constantes de blocos, sem precisar reescrever o código para cada um, permitindo que o compilador realize otimizações em tempo de compilação.

DOI: 10.5540/03.2018.006.01.0343

As matrizes quadradas utilizadas são provenientes de simulações da explotação de reservatórios de petróleo. Algumas de suas características podem ser vistas na Tabela 1, onde "Ordem" é o número de linhas-bloco da matriz, cada bloco sendo uma matriz quadrada de ordem quatro, "BlcsNN" é o total de blocos não nulos de matriz e "ILU(1) BlcsNN" é a quantidade de blocos não-nulos da fatoração incompleta LU de nível 1, contando os fatores L e U.

Tabela 1: Detalhes das matrizes

Nome	**Ordem**	**BlcsNN**	**ILU(1) BlcsNN**
Mat1	66.077	449.376	824.313
Mat2	182.558	1.223.695	2.224.768
Mat3	408.865	2.731.921	4.975.059
Mat4	617.459	4.379.139	8.257.861

6 Resultados

6.1 BILUk Numérico

A Tabela 2 apresenta os resultados de tempo em segundos, **t(s)**, e gigaflops por segundo, **Gf/s**, do BILUk no segundo ambiente para um, dois, quatro e dez cores (uma thread por core). Todos os testes foram realizados com $k = 1$ (BILU1).

Tabela 2: BILUk na Ivy Bridge

	1 core		**2 cores**		**4 cores**		**10 cores**	
Nome	**t(s)**	**Gf/s**	**t(s)**	**Gf/s**	**t(s)**	**Gf/s**	**t(s)**	**Gf/s**
Mat1	0,09	2,8	0,05	5,2	0,03	9,2	0,01	17,5
Mat2	0,23	2,7	0,12	5,1	0,06	9,6	0,03	19,9
Mat3	0,56	2,5	0,29	4,7	0,16	8,8	0,08	18,3
Mat4	1,19	2,4	0,67	4,2	0,39	7,3	0,22	13,1

A Tabela 2 mostra uma redução de 5,4 a 7,4 vezes no tempo de construção do BILUk de 1 para 10 cores, representando um aumento de 2,7Gf/s para 19,9Gf/s para o caso mais positivo. Apesar da performance da maior matriz ser a menos intensa, o resultado dos demais testes não indicam que o aumento do tamanho da matriz reduza a escalabilidade do código, pois há um aumento dessa propriedade para as matrizes Mat1, Mat2 e Mat3. Considerando a escalabilidade ideal, que seria o aumento de dez vezes da performance de um para dez cores, o BILUk apresentou um potencial de 74% da escalabilidade perfeita. No entanto o BILUk é um código limitado pela largura de banda, que tem um aumento em proporção inferior ao número de cores utilizados. Portanto podemos concluir que a técnica de escalonamento por níveis produz boa escalabilidade para esta quantidade limitada de cores no BILUk.

DOI: 10.5540/03.2018.006.01.0343

Os resultados para o terceiro ambiente estão na Tabela 3 e mostram resultados inferiores aos atingidos na Ivy Bridge.

Tabela 3: BILUk no KNC

	1 core 4 threads		**15 cores 60 threads**		**30 cores 120 threads**		**60 cores 240 threads**	
Nome	**t(s)**	**Gf/s**	**t(s)**	**Gf/s**	**t(s)**	**Gf/s**	**t(s)**	**Gf/s**
Mat1	0,48	0,5	0,06	4,5	0,04	5,9	0,04	6,7
Mat2	1,21	0,5	0,10	6,4	0,08	7,9	0,13	4,8
Mat3	2,90	0,5	0,27	5,1	0,25	6,2	0,31	4,5
Mat4	5,60	0,5	0,85	3,3	0,74	3,8	0,90	3,2

O KNC, veja a Tabela 3, mostra um speedup de 8,8 à 12,7 vezes de 1 para 15 cores, atingindo 6,35Gf/s e um potencial de 84% da escalabilidade ideal no melhor caso. No entanto a melhoria de performance de 15 para 30 cores não ultrapassa 1,3 vezes, o que representaria 8% da escalabilidade ideal. Analisando de 30 para 60 cores notamos uma perda de performance para os casos testados, com exceção do caso Mat1. Estes resultados sugerem que o nível de paralelismo obtido com o escalonamento por níveis no BILUk não é o suficiente para alimentar o número de threads do KNC.

6.2 Solver Triangular

A Tabela 4 apresenta os resultados do solver triangular no primeiro e terceiro ambientes. Apresentamos a largura de banda atingida (LarBnd), o percentual desta em relação à largura de banda disponível (LarBnd %) e o total de operações de ponto flutuante realizadas por segundo, tomando como unidade de medida um gigaflop por segundo (Gf/s). O solver triangular é uma rotina sabidamente limitada pela largura de banda, no entanto a performance não ultrapassa 66% da largura disponível na máquina, para 8 threads, no Sandy Bridge. Esta performance sugere que a abordagem de escalonamento por níveis gera resultados positivos mas ainda não é suficiente para garantir um paralelismo adequado para as matrizes de interesse. Testes preliminares mostraram um potencial superior nestas mesmas matrizes utilizando as técnicas descritas em [4].

Os testes no terceiro ambiente com 240 threads (60 cores, quatro threads por core) são ainda menos encorajadores, não ultrapassando 21% da largura de banda, pois o KNC exige um paralelismo mais intenso do que a técnica de escalonamento por níveis oferece para os problemas analisados.

7 Conclusões

Os resultados apresentados mostram que a técnica de escalonamento por níveis para a construção do precondicionador BILUk produz um potencial de escalabilidade adequado para CPU, de acordo com os resultados realizados no ambiente com Ivy Bridge. No entanto, no KNC a performance da construção do precondicionador com as técnicas apre-

DOI: 10.5540/03.2018.006.01.0343

Tabela 4: Solver Triangular, Sandy Bridge 8 threads, KNC 240 threads

	Sandy Bridge			**KNC**		
Nome	**LarBnd**	**LarBnd %**	**Gf/s**	**LarBnd**	**LarBnd %**	**Gf/s**
Mat1	23,1	62%	5,6	10,7	7%	2,6
Mat2	24,3	65%	5,9	24,1	17%	5,8
Mat3	23,2	62%	5,6	29,9	21%	7,2
Mat4	24,8	66%	6,0	30,6	21%	7,3

sentadas se mostrou ineficiente devido à limitação de trabalho paralelo disponível nos problemas de interesse.

Quanto à aplicação do precondicionador, apesar de resultados positivos na CPU, ainda é possível observar uma margem considerável para a performance do solver triangular alcançar o seu limite da largura de banda. Assim como na construção, a aplicação do precondicionador no KNC se mostrou ineficiente e com baixa escalabilidade. Apesar de atingir uma largura de banda superior aos casos em CPU, o valor ainda é muito baixo, visto que uma comparação justa sugere a performance de 2 sockets de CPU contra uma placa KNC, e que apenas 21% da largura de banda disponível foi atingida.

Devemos, no futuro, investigar se o balanceamento de carga e outras técnicas alternativas de paralelização do algoritmo poderão alcançar performances mais satisfatórias em uma arquitetura como KNC, que exige um grau elevado de paralelismo para tanto. Para alcançar uma performance otimizada utilizando OpenMP, iremos considerar a abordagem apresentada em [4] nos próximos passos da pesquisa.

Referências

[1] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. LIBXSMM: accelerating small matrix multiplications by runtime code generation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 84. IEEE Press, 2016.

[2] M. Naumov. Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the gpu. *NVIDIA Corp., Westford, MA, USA, Tech. Rep. NVR-2011*, 1, 2011.

[3] M. Naumov. Parallel incomplete-LU and Cholesky factorization in the preconditioned iterative methods on the GPU. Technical report, NVIDIA, NVR-2012, 2012.

[4] J. Park, M. Smelyanskiy, N. Sundaram, and P. Dubey. Sparsifying synchronization for high-performance shared-memory sparse triangular solver. In *Supercomputing*, pages 124–140. Springer, 2014.

[5] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

DOI: 10.5540/03.2018.006.01.0343