EXECUÇÃO DE SISTEMAS DE EQUAÇÕES LINEARES EM PROCESSADORES MULTI-CORE[1]

Henrique Bespalhuk Facci[2], Ronaldo Augusto de Lara Gonçalves
UEM – Universidade Estadual de Maringá
Departamento de Informática
Avenida Colombo, 5790. Jardim Universitário.
CEP 87020-900 Maringá – PR

hfacci@gmail.com, ronaldo@din.uem.br

Resumo. *Aplicações paralelas estão sendo cada vez mais estudadas devido ao fato de que processadores compostos de vários núcleos de processamento se tornaram comuns. Nesse contexto, o objetivo deste trabalho é o de implementar e analisar algoritmos paralelos que resolvem sistemas de equações lineares. Foram feitos 2 experimentos, com os algoritmos de eliminação de Gauss e relaxamento Gauss-Seidel. Os resultados obtidos permitem averiguar a melhora no desempenho de cada algoritmo sobre a o respectivo algoritmo seqüencial, de acordo com a variação no número de threads, atingindo o pico acima 82% na redução do tempo de processamento em casos aqui relatados.*

1. Introdução

Hoje em dia estamos sempre querendo que nossas aplicações computacionais executem mais rápido, como nas aplicações científicas que demandam confiabilidade e rapidez. Para conseguir isso, avanços no hardware e no projeto de técnicas e ferramentas para a criação de aplicações mais eficientes têm sido investigados. Um dos avanços na área de hardware foi a criação de processadores multi-core, o qual contém 2 ou mais núcleos de processamento. Teoricamente, cada núcleo de um processador multi-core tem a capacidade de sustentar o mesmo desempenho que um processador mono-core da mesma linha. Uma forma de aproveitar este poder computacional em benefício da aplicação é fazer uso da programação paralela [1].

Visto o grande potencial da paralelização de aplicações, muitos estudos têm sido feitos para descobrir o melhor método de maximizar o uso desses processadores. Diferentes técnicas então foram desenvolvidas para analisar o desempenho dessas aplicações paralelas, entre elas estão a simulação, modelagem analítica e experimentação real. A computação paralela está sendo usada para modelar problemas difíceis das diversas áreas de conhecimento, tais como Meio Ambiente, Física,

[1] Projeto apoiado pela Fundação Araucária (PAC - Clusters)

[2] Aluno bolsista PIBIC

Aplicada, Matemática, Biociências, Biotecnologia, Genética, Química, Ciências Moleculares, Geologia, Sismologia; Engenharia Elétrica e Ciência da Computação.

Hoje em dia, até aplicações comerciais estão necessitando do poder da computação paralela. Aplicações com mineração de dados, busca na *web*, processamento de imagens e diagnósticos médicos, necessitam processar grandes quantidades de dados de maneiras sofisticadas para um bom funcionamento. Neste contexto, este artigo apresenta a paralelização de dois métodos matemáticos usados na solução de sistemas de equações lineares, Eliminação de Gauss (não iterativa) e Relaxamento de Gauss-Seidel (iterativa) bem como sua análise de desempenho.

Este artigo está organizado da seguinte forma. A seção 2 explora os conceitos de programação paralela e *multithreaded*. A seção 3 apresenta os algoritmos paralelos. Os experimentos e resultados aparecem na seção 4. A seção 5 mostra as principais conclusões e a bibliografia utilizada aparece na última seção.

2. Modelos de Programação Paralela

Computação paralela pode ser definida como o uso de múltiplos recursos do computador para resolver um problema computacional [1]. Os diferentes modelos de programação paralela são dependentes do modelo de arquitetura da memória utilizado. Esses modelos de arquitetura são a memória compartilhada, memória distribuída e um modelo hibrido de memória compartilhada-distribuída.

A arquitetura memória compartilhada [6] se baseia na existência de vários processadores, provavelmente muito próximos e quase sempre dentro de uma mesma estrutura física de acomodação, usando uma mesma memória global, comum a todos os processadores para comunicação entre os processos, muito embora cada processador possa ter sua memória local privada. Programação *multithreaded* normalmente é a mais comum para se utilizar este tipo de arquitetura..

A arquitetura de memória distribuída [7] se baseia na existência de vários computadores, provavelmente remotos e dependendo do tipo de aplicação pode ser estruturado fisicamente em espaço laboratorial, empresarial ou de longa distância, conectados entre si e dispostos a executarem paralelamente uma mesma aplicação. Como cada computador tem a sua própria memória local, são necessários mecanismos de troca de mensagem para a comunicação. MPI é um dos padrões de plataformas que fornece suporte a troca de mensagem nesse sistema.

O modelo híbrido de arquitetura [8] tenta aproveitar as vantagens dos modelos anteriores para compensar as desvantagens de cada arquitetura. Normalmente, as memórias são locais com alguma faixa de endereçamento compartilhada a todos os processadores, suportada por *hardware* ou *software* específico. Nos experimentos deste artigo foi usado o modelo de memória compartilhada e estilo de programação *multithreaded*. Para a implementação dos algoritmos foi utilizada a linguagem C juntamente com a biblioteca Pthreads. O compilador usado foi o GCC versão 4.1.2 sem nenhuma otimização.

2.1. Trabalhos Relacionados

No trabalho realizado por [2], desenvolvido na Universidade do Algarve, é proposto um método de paralelizar aplicações matriciais automaticamente usando um sistema chamado SPAM 1.0, que paraleliza um código através de uma linguagem sequencial também proposta, SEQ. Para a utilização do sistema é necessário escrever um código em SEQ e detalhar como os blocos estão conectados.

Na dissertação feita por [3], desenvolvida na Universidade de São Paulo em São Carlos, foi desenvolvido uma biblioteca para facilitar a implementação de códigos paralelos de forma mais amigável ao usuário, deixando o paralelismo da forma mais abstrata possível usando a abstração de um objeto. Usando então esta abordagem para a criação da interface e a biblioteca ScaLAPACK para realizar as operações, foi possível então criar uma interface mais amigável sem ter perdas de desempenho.

O trabalho produzido por [4], desenvolvido na Universidade Federal de Lavras, tem como objetivo paralelizar o método de eliminação de Gauss para um ambiente PVM, *Parallel Virtual Machine*, implementado pela RedHat. O código implementado para os testes teve base em um algoritmo de eliminação de Gauss orientado a linha, ou seja, cada processador cuidava de linhas diferentes. A Paralelização alcançou um máximo de *speedup* de 2,5 em 3 processadores.

Na apostila feita por [5], Desenvolvida na Universidade Estadual de São Paulo, é feita uma relação dos passos que se deve tomar na hora de escrever um código em paralelo. A apostila mostra as representações de uma aplicação, os modelos e as execuções de um código em paralelo.

3. Aplicações Alvo

Nesta seção são apresentados os algoritmos que foram paralelizados, suas fórmulas matemáticas e seus algoritmos na forma paralela usando *threads*.

3.1. Eliminação de Gauss

A Eliminação de Gauss é um algoritmo para resolver problemas de sistemas de equações lineares. Ele resolve os sistemas usando os valores da diagonal principal como pivô e então zera os valores abaixo da mesma coluna, atualizando os valores das outras variáveis das linhas ao fazer isso. Quando o algoritmo termina de processar as variáveis abaixo da diagonal principal, ele então passa a fazer o mesmo processo na parte de cima da diagonal principal, começando desta vez pelo final da diagonal e zerando os valores. A seguir está a fórmula matemática do método.

$$a_{j,k} = a_{j,k} + a_{j-1,k}\left(\frac{-a_{j,i}}{a_{ii}}\right) \quad (1)$$

Onde i = (1…n-1), j = (i…n) e k = (i…n+1), onde n é o número de equações do sistema. As versões seqüencial e paralela usados na implementação desde método são mostradas nos Algoritmos 1 e 2.

Algoritmo 1 Eliminação de Gauss

```
variáveis:
I, J, K, Var, Res, Mul, N;
Início:
Leia(coeficientes);
Leia(resultados);
Para (I = 0; I < N-1; I++)
    Para (J = I+1; J < N; J++)
        Mul = Var[J][I] / Var[I][I];
        Para (K = I; K < N; K++)
            Var[J][K] = Var[J][K] - Var[K][J] * Mul;
        Res[J] = Res[J] – Res[I] * mul;
Para (I = N-1; I > 0; I++)
    Para (J = I-1; J >= 0; J++)
        Res[J-1] = Res[J] – Res[I] * Var[I][J];
        Var[I][J] = 0;

Fim;
```

Algoritmo 2 Eliminação de Gauss Paralela

```
variaveis:
I, J, Var, Res, N, Num_threads, Barreira;
Início:
Leia(coeficientes);
Leia(resultados);
Leia(Num_threads)
CriaThreads(Num_threads);
Barreira = 0;
Cada thread faça:
variaveis:
I, J, K, mul;
    Para (I = 0; I < N-1; I++)
        If (I > Barreira) TravaThread();
        Para (J = I+1+thead_ID; J < N; J += Num_Threads)
            Mul = Var[J][I] / Var[I][I];
            Para (K = I; K < N; K++)
                Var[J][K] = Var[J][K] - Var[K][J] * Mul;
            Res[J] = Res[J] – Res[I] * mul;
        If (I % Num_threads == thread_ID)
            Barreira++;
            LiberaThreads();
EsperaThreads();
Para (I = N-1; I > 0; I++)
    Para (J = I-1; J >= 0; J++)
        Res[J-1] = Res[J] – Res[I] * Var[I][J];
        Var[I][J] = 0;

Fim;
```

O Algoritmo 2 trata apenas da primeira parte do método, que tem uma complexidade $O(n^3)$, de forma paralela, deixando então a segunda parte ser executada de forma sequencial. Ele obtém os dados do sistema de equações (coeficientes e resultados) e os aloca em uma matriz, obtém o número de *threads* e as cria. Cada *thread* se responsabiliza por um subconjunto das equações a processar, associado a seu número de identificação (`thread_ID`). Ao começar uma iteração (`I`), a *thread* verifica se a equação pivô que será usada (`I`) está pronta, para isso ela verifica o valor da variável `barreira`; caso `barreira` < `I`, a equação ainda não foi processada ao ponto de poder ser usada como pivô, então a *thread* espera na barreira até que a *thread* responsável por aquela equação termine de processá-la, caso contrario a *thread* continua a sua iteração normalmente. As variáveis `Num_threads` e `Barreira` são globais, assim como as matrizes `Var` e `Res`.

3.2. Relaxamento Gauss-Seidel

Assim como o Algoritmo 1, o algoritmo de Relaxamento de Gauss-Seidel resolve sistemas de equações lineares. O método usado por este algoritmo é iterativo, executando um procedimento repetitivo até os valores atingidos sejam aceitáveis ou alcance o número máximo de iterações. A seguir está a fórmula matemática do método.

$$x_i^k = \frac{1}{a_{i,i}}\left[b_i - \sum_{j=1}^{i-1} a_{i,j} a_j^k - \sum_{j=i+1}^{n} a_{i,j} x_j^{k-1}\right] \quad (2)$$

Com i = (0...n), onde n é o tamanho da matriz, e k é o número da iteração atual.

O sistema exige que a matriz de coeficientes e resultados satisfaça algumas restrições para que ele possa ser convergente. A matriz precisa ter uma diagonal dominante, ser simétrica ou positiva definida. O método de solução consiste em, a partir de valores anteriores das variáveis, usá-los para calcular os valores atualizados das variáveis, os quais tentam se aproximar (convergente) dos valores-solução, mas pode se afastar (não convergente). Inicialmente usa-se um valor inicial (estimativas iniciais) para os valores para as variáveis. Uma iteração é o tempo de processar a matriz toda e fazer uma nova tentativa de aproximação para todas as variáveis. Quando uma iteração acaba, a próxima inicia com os valores atualizados. Com o passar das iterações, os variáveis passam a sofrer menos variação, até o momento onde os valores não mudam mais ou estão abaixo de um limite.

A seguir estão os Algoritmos 3 e 4, sequencial e paralelo respectivamente, usados na implementação desde método. O Algoritmo 4 procede similarmente ao Algoritmo 3 no início. Após todos os dados serem lidos, o algoritmo utiliza um valor inicial para as variáveis, no caso dos experimentos deste trabalho, 0 para todas, e cria a quantidade de *threads* requisitadas para trabalhar assim como no Algoritmo 2. Como não existe qualquer tipo de sincronização, as variáveis podem estar com valores diferentes em relação aos valores do método sequencial, ainda que estejam no mesmo momento da execução, porém, por causa do método ser iterativo, esta diferença se resolve com o avanço das iterações.

Algoritmo 3 Relaxamento Gauss-Seidel

```
variaveis:
I, J, K, Var, Res, Xs, Aux, N;
Início:
Leia(coeficientes);
Leia(resultados);
Leia(MaxIterações);
SuposuçãoInicial(Xs);
Para (I = 0; I < MaxIterações; I++)
    Para (J = 0; J < N; J++)
        Aux = 0;
        Para (K = 0; K < J; K++)
            Aux += Xs[K]*Var[J][K]
        Para (K = J+1; K < N; K++)
            Aux += Xs[K]*Var[J][K]
        Xs[J] = (Res[J] - Aux)/Var[J][J];
Fim;
```

Algoritmo 4 Relaxamento Gauss-Seidel Paralelo

```
variaveis:
Var, Res, Xs, N, Num_threads, MaxIterações;
Início:
Leia(coeficientes);
Leia(resultados);
Leia(MaxIterações);
Leia(Num_threads);
CriaThreads(Num_threads);
SuposuçãoInicial(Xs);

Cada thread faça:
variaveis:
I, J, K, Aux;
    Para (I = 0; I < MaxIterações; I++)
        Para (J = Thread_ID; J < N; J += Num_threads)
            Aux = 0;
            Para (K = 0; K < J; K++)
                Aux += Xs[K]*Var[J][K]
            Para (K = J+1; K < N; K++)
                Aux += Xs[K]*Var[J][K]
            Xs[J] = (Res[J] - Aux)/Var[J][J];

EsperaThreads();
Fim;
```

4. Experimentos

Essa seção descreve os experimentos realizados com os algoritmos apresentados na seção anterior e mostra os resultados em quadros e gráficos. Os códigos dos

experimentos foram testados em um ambiente linux, sobre uma plataforma de memória compartilhada contendo 2 processadores quad-core *hyperthreading*, totalizando o poder de 8 *cores* duplos, o que chamamos aqui de 16 *slots* de processamento, o que permite a execução de até 16 *threads* simultaneamente. Convém ressaltar que, na tecnologia *hyperthreading*, os dois *slots* de um mesmo *core* contém recursos compartilhados e recursos separados por *thread* e a execução simultânea não é completamente paralela. Cada *slot* tem uma cache de 12288 KB e uma performace de 2394.171 MHz. Existe um total de 8 GB de memória RAM disponível neste ambiente.

4.1. Experimento 1 – Eliminação de Gauss

Para este experimento, os Algoritmos 1 e 2 foram executados sobre os mesmos sistemas de equações. Os resultados obtidos em termos de tempo de execução em segundos em função do número de *threads* utilizadas são mostrados no Quadro 1 e plotados no gráfico da Figura 1. O Quadro 1 também mostra o ganho percentual do método paralelo sobre o seqüencial em termos de redução do tempo.

Num_Threads	1	2	4	8	16	32	64
Sequencial	56,599	56,599	56,599	56,599	56,599	56,599	56,599
Paralelo	55,563	42,068	23,144	12,487	11,518	13,702	15,489
Ganho	1,8%	25,7%	59,1%	77,9%	79,6%	75,8%	72,6%

Quadro 1: Desempenho do método de eliminação de Gauss

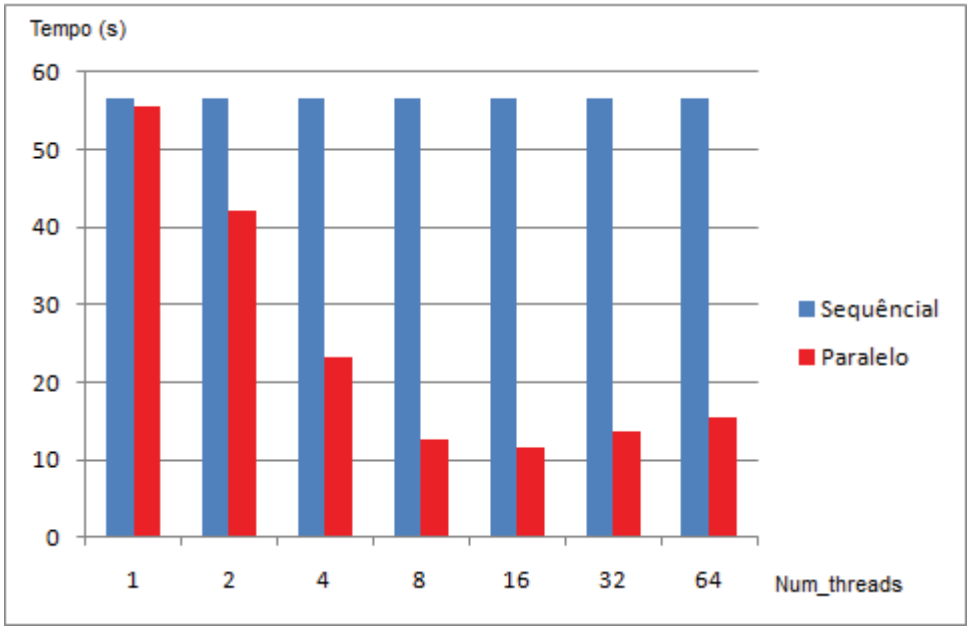

Figura 1. Desempenho do método de eliminação de Gauss

Como se pode observar no Quadro 1, o ganho de desempenho atinge o máximo de 79,6% quando se usa 16 *threads*, que é o máximo de *slots* da arquitetura. Percebe-se que o ganho aumenta muito pouco a partir de 8 threads, pois este número já preenche

todos os *cores* da arquitetura. Quando se executa mais de 8 *threads*, os *cores* passam a compartilhar duas ou mais *threads* cada um. Ainda com 16 *threads*, cada *core* pode executar duas *threads* em *slots* diferentes por meio da capacidade *hyperthreading*, mas a partir deste ponto o ganho reduz, pois as *threads* passam a sofrer maior conflito.

4.2. Experimento 2 – Relaxamento de Gauss-Seidel

Para este experimento, os Algoritmos 3 e 4 foram executados sobre os mesmos sistemas de equações. Ambos os códigos foram implementados modificando-se os algoritmos anteriores. Experimentamos 3 sistemas com 1000 equações de 1000 incógnitas. Para uma melhor comparação dos resultados, os códigos não param de executar quando o sistema converge, mas sim quando terminam todas as iterações pedidas. Os resultados obtidos em termos de tempo de execução em segundos em função do número de *threads* utilizadas são mostrados no Quadro 2 e plotados no gráfico da Figura 2. O Quadro 2 também mostra o ganho percentual do método paralelo sobre o sequencial em termos de redução do tempo.

Num_Threads	1	2	4	8	16	32	64
Sequencial	131,34	131,34	131,34	131,34	131,34	131,34	131,34
Paralelo	130,57	76,96	42,51	29,51	22,88	24,78	24,36
Ganho	0,6%	41,4%	67,6%	77,5%	82,6%	81,1%	81,5%

Quadro 2: Desempenho do método de relaxamento de Gauss-Seidel

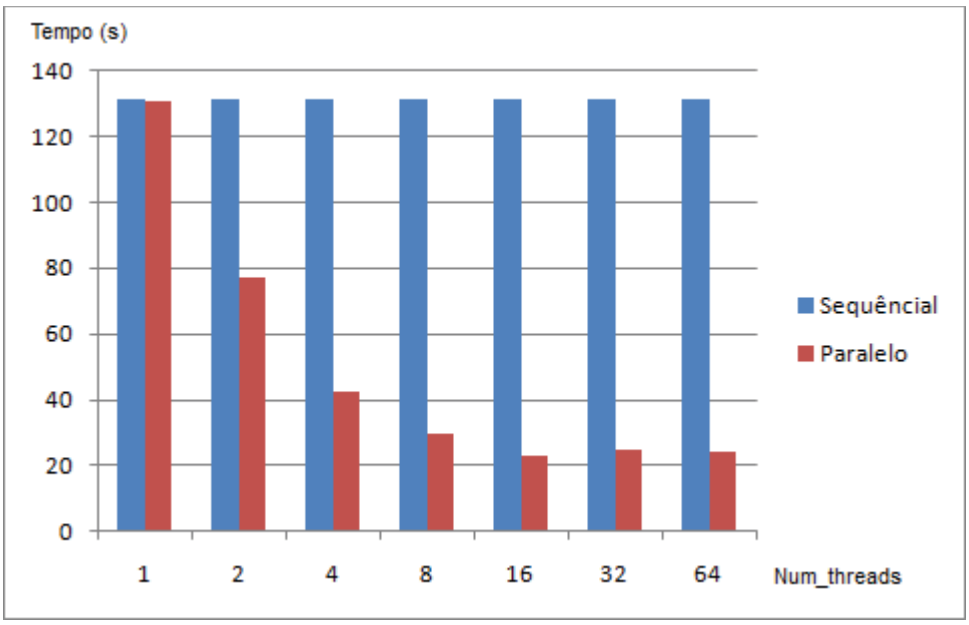

Figura 2. Desempenho do método de relaxamento de Gauss-Seidel

Como se pode observar no Quadro 2, o ganho de desempenho atinge o máximo de 82,6% quando se usa 16 *threads*, que é o máximo de *slots* da arquitetura, ultrapassando o valor equivalente alcançado pelo método de eliminação em 3%.

Percebe-se que o ganho aumenta muito pouco a partir de 8 *threads*, pois este número já preenche todos os *cores* da arquitetura, perfazendo um comportamento similar àquele obtido pelo algoritmo de eliminação e as justificativas são as mesmas para os desempenhos, quando se usa mais de 8 ou 16 *threads* e *hyperthreading*.

5. Conclusão

O processamento matricial pode ser muito lento se executado sequencialmente. Quando usamos as técnicas de programação paralela para melhorar o desempenho, o quanto podemos melhorar depende muito de como a aplicação é paralelizada. Sistemas de equações lineares usam processamento matricial e se encaixam nessa categoria. Nesse trabalho avaliamos a paralelização de algoritmos que solucionam sistemas de equações lineares, usando dois métodos conhecidos: Eliminação de Gauss (não iterativo) e Relaxamento de Gauss-Sedel (iterativo). Em todos os experimentos foi possível melhorar o desempenho das aplicações usando técnicas de paralelismo, onde ambos os códigos tiveram um ganho próximo de 80%, sendo que a solução iterativa se mostrou mais eficiente.

Referências Bibliográficas

[1] Blaise Barney, Lawrence Livermore National Laboratory, Introduction to Parallel Computing, LTRAIN: EC3500, última atualização: 12/10/2010

[2] H. A. Amadeu, A. E. Ruano, Paralelização Automática de Algoritmos Matriciais. Dissertação de Doutorado. Doutoramento em Engenharia Eletrônica e Computação na Especialidade de Sistemas de Controlo. Universidade do Algarve – Faro 2003.

[3] F. A. Rodrigues, Técnicas de Orientação a Objetos para Computação Científica Paralela. Dissertação de Mestrado. Física Aplicada. Universidade de São Paulo – São Carlos 2004.

[4] J. A. Monte-Mor, C. R. Lara e J. O. de Albuquerque. Método Paralelo de Eliminação de Gauss no Ambiente PVM. Revista Eletrônica de Iniciação Científica da Sociedade Brasileira de Computação. ISSN 1519-8219. Ano II, Vol. II, Num. I, SBC, mar-2002.

[5] A. Goldman, Modelos para Computação Paralela. Apostila Técnica. Universidade de São Paulo (IME - USP). Curso de curta duração ministrado/Especialização – São Paulo 2003.

[6] Grama, A., Gupta A., Karypis, G. e Kumar, V., Introduction to Parallel Computing, Addison Wesley, 2ª edição, Janeiro 2003.

[7] Dongarra, J. e Dunigan, T., "Message-passing performance of various computers", Concurrency: Practice and Experience, Vol. 9, No. 10, 1997, pp. 915-926.

[8] Patterson, D. A. e Hennessy, J. L., Organização e Projeto de Computadores: A Interface Hardware/Software, Editora Campus, Rio de Janeiro, 3ª edição, 2005.