PARALELIZAÇÃO DE UM ALGORITMO HÍBRIDO PARA SOLUÇÃO DE SISTEMAS NÃO LINEARES COM RAÍZES MÚLTIPLAS

Dalmo Stutz – stutz@iprj.uerj.br
Universidade do Estado do Rio de Janeiro, Instituto Politécnico, 28630-050 – Nova Friburgo, RJ, Brasil

Resumo. *Este trabalho apresenta uma estratégia de implementação paralela de um algoritmo híbrido para a solução de sistemas não lineares com múltiplas soluções (raízes), cujo modelo é construído a partir de dois métodos: Luus-Jaakola - usado para a geração de pontos iniciais como possíveis candidatos a solução do sistema - e Hooke-Jeeves, usado para produzir uma solução mais refinada (ótima) obtida a partir desses pontos.*

Palavras-chave: *Otimização, algoritmo híbrido, programação paralela*

1. INTRODUÇÃO

Nas últimas décadas, a contínua evolução do hardware tem proporcionado melhorias significativas nos desempenhos dos computadores e uma oferta maior de recursos computacionais. Essa evolução teve início na década de 60 com a miniaturização dos computadores e a produção dos primeiros circuitos integrados e se estende até os dias atuais, apresentando uma taxa de crescimento exponencial que dobra o número de transistores dentro de um chip de computador a cada dois anos aproximadamente (Fig.1). Essa tendência ficou conhecida na literatura como Lei de Moore (1965).

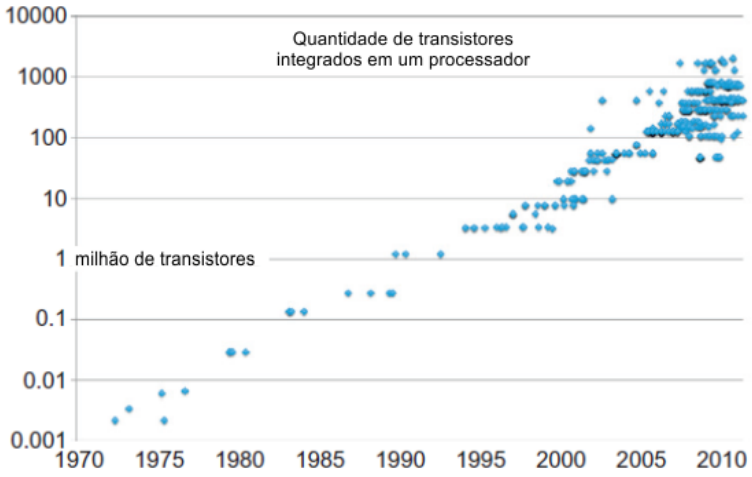

Figura 1- Gráfico da evolução da quantidade de transistores integrados em um microprocessador (McCool, Robinson e Reinders. 2012).

Ao longo dos anos, esse crescimento possibilitou, consequentemente, o desenvolvimento de microprocessadores cada vez mais complexos com avanços significativos na frequência dos relógios (*clocks*), que controlam os tempos de execução das instruções em um processador. Apesar da velocidade dos relógios não ter seguido a mesma taxa de crescimento dos transistores, no entanto, ela apresentou aumentos consideráveis. No período

compreendido entre 1973 e 2003, por exemplo, a velocidade dos relógios teve um aumento de 1000 ×, partindo de 1MHz em 1973 e chegando a 1GHz em 2003.

Apesar dos avanços, no entanto, essa taxa de crescimento não é infinita, uma vez que as condições (aumento da velocidade do *clock*, paralelismo em baixo nível, velocidade de acesso à memória, consumo de energia, etc.) que tornaram isso possível começaram a atingir os seus limites (Fig. 2). Só para citar um exemplo, por volta de 2005, a velocidade dos *clocks* parou de crescer e estabilizou-se em torno da faixa 3GHz. Os motivos para isso ficou conhecido como "*Three Walls*": i) *Power wall* - à medida que a frequência do clock aumenta, o consumo de energia para mantê-la cresce de forma desproporcional em níveis inaceitáveis, principalmente, para dispositivos móveis que dependem de baterias, tais como: notebooks, celulares, tablets, etc.; ii) *Instruction-level parallelism wall* (ILP) - o processo de paralelismo automático de instruções em baixo nível, realizado pelos atuais processadores, já atingiu o seu limite máximo de capacidade e eficiência; iii) *Memory wall* - aumento na discrepância entre a velocidade do processador e a velocidade de acesso à memória (tempo de latência), que tem crescido a uma taxa bem menor que a dos processadores.

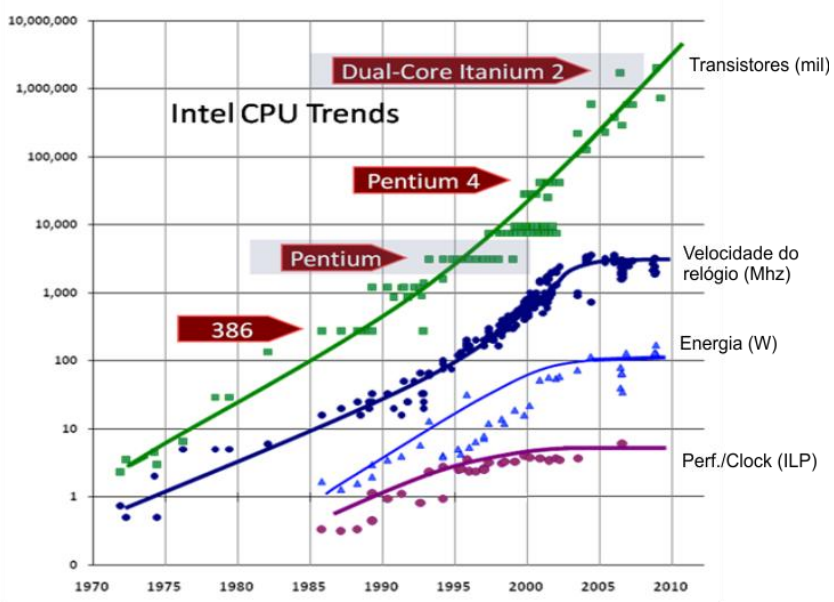

Figura 2- Avanços nos microprocessadores Intel® (Sutter, 2005).

Em suma, para se alcançar maiores desempenhos de processamento hoje em dia não se pode mais contar com aumentos nas frequências do relógio, incrementos nas taxas de transferências de dados de/para a memória, paralelismos de baixo nível, etc., é preciso escrever programas paralelos, levando-se em consideração não só um bom algoritmo, mas também pensando em diversos outros aspectos tão importantes quanto e que interferem de alguma forma nos tempos de processamento, tais como: redução nos custos de comunicação entre processos e unidades de processamento, acesso compartilhado à memória, distribuição e compartilhamento de dados, processamento distribuído, balanceamento de cargas, entre outros.

3.1. O algoritmo híbrido

Desenvolvido por Silva (2009, 2011), o algoritmo híbrido para solução de problemas não lineares com múltiplas raízes é um método composto por dois outros métodos concatenados: Luus-Jaakola (1973) e Hooke-Jeeves (1961).

A ideia central do algoritmo (Fig.3) é utilizar o método Luus-Jaakola para a geração de pontos iniciais, dentro dos limites de uma dada região de busca, como possíveis candidatos a solução do sistema e, então, usar esses pontos para alimentar o método de Hooke-Jeeves de modo que se produza uma solução mais refinada (mais próxima do ótimo) do problema que está sendo avaliado.

Como o algoritmo Luus-Jaakola localiza apenas um ponto a cada execução, para se identificar outras possíveis soluções e para que outros candidatos possam ser encontrados dentro da região de busca, faz-se necessário que o algoritmo híbrido seja executado em um processo iterativo, repetidas vezes. A cada iteração, o algoritmo híbrido produz uma possível solução para o problema e, à medida que são gerados, esses valores vão sendo armazenados dentro de um vetor de raízes (R).

Ao longo das execuções, esse procedimento acaba produzindo algumas raízes repetidas ao encontrar um mesmo conjunto de pontos. A solução para esse problema é, então, resolvido filtrando-se os resultados encontrados, selecionando valores distintos e produzindo um conjunto solução (S) com todas as diferentes raízes localizadas pelo algoritmo híbrido dentro do espaço de busca, solução final do problema.

1: Defina um número total de iterações nte
2: Defina R = vetor de raízes
3: Defina S = vetor solução
4: Para k=1 até nte faça
5: Execute a rotina Luus-Jaakola
6: Faça $x_{inicial}(\text{Hooke-Jeeves}) = x_{inicial}(\text{Luus-Jaakola})$
7: Execute a rotina Hooke-Jeeves
8: Faça $r_k = x_{final}(\text{Hooke-Jeeves}), r_k \in R$
9: Fim_para
10: Filtre as raízes repetidas em R, classificando-as por aproximação
11: Minimize cada classe de vetores, gerando um valor por classe (raízes distintas)
12: Copie as raízes distintas de R em S
13: Imprima o vetor solução S

Figura 3 - Algoritmo híbrido (Silva, 2011).

A estrutura do método híbrido, além de apresentar as boas características dos métodos de otimização estocástico (Luus-Jaakola) e determinístico (Hooke-Jeeves), consome um número menor de avaliações da função-objetivo (melhor desempenho), exigindo um menor tempo computacional.

2. DESENVOLVIMENTO DE ALGORITMOS PARALELOS

O desenvolvimento de algoritmos é um componente crucial na solução de problemas computacionais. Um algoritmo serial é basicamente uma sequência estruturada e finita de passos (instruções), ordenados logicamente, para se resolver uma determinada tarefa ou solucionar um dado problema computacional, usando um computador serial.

Similarmente, um algoritmo paralelo é também um conjunto de passos para se resolver um dado problema, porém, usando múltiplos processadores.

Especificar um algoritmo paralelo, no entanto, envolve mais do que simplesmente especificar esses passos, é preciso pensar na concorrência e no conjunto de passos que podem ser executados simultaneamente. Na prática, a especificação de um algoritmo paralelo típico deve incluir algumas (se não todas!) das seguintes etapas: i) identificar porções do algoritmo que podem ser executadas concorrentemente; ii) mapear as partes de trabalho concorrente em múltiplos processos rodando em paralelo; iii) distribuir os dados de entrada e saída; iv) gerenciar o acesso aos dados compartilhados em múltiplos processadores; v) sincronizar os processos nos vários estágios da execução do programa em paralelo; etc. (Grama *et.al.*, 2003).

Uma outra característica que se procura quando se desenvolve programas em paralelo é a escalabilidade. Esse atributo compreende a habilidade de uma sistema se adaptar a uma demanda crescente de dados ou trabalho, seja processando volumes maiores de trabalho de maneira uniforme e/ou estar preparado para crescer (Bondi, 2000).

Uma forma de tornar o algoritmo híbrido um programa escalável é permitindo que um mesmo problema seja executado em um tempo menor ou permitindo que um volume maior de trabalho possa ser executado em um mesmo tempo. Isso pode ser resolvido, por exemplo, distribuindo-se a carga de trabalho, que seria executada serialmente (algoritmo híbrido serial) por um único processador, para np unidades de processamento rodando em paralelo (algoritmo híbrido paralelo).

Assim, podemos distribuir a carga de trabalho para mais de um processador, de forma que o método híbrido em paralelo possa executar o processo em um tempo menor, ou, então, de modo a podemos aumentar a carga de trabalho, estendendo a área de busca ou processando funções-objetivo mais complexas, por exemplo, de forma que a execução do programa em paralelo leve um mesmo tempo de execução, porém, usando mais processadores.

3. ESTRATÉGIA DE IMPLEMENTAÇÃO PARALELA

Em uma rápida análise do algoritmo híbrido (Fig.3), pode-se verificar que a cada iteração uma nova solução é obtida e que cada uma delas é uma solução independente das outras. Isso significa que, para uma primeira abordagem de paralelização do algoritmo serial, podemos aproveitar essa independência para distribuir o número total de execuções (nte) do algoritmo híbrido (carga) entre várias unidades de processamento, fazendo:

$$ni = nte/np \tag{1}$$

onde ni é o número de iterações individuais executadas em cada unidade de processamento, nte o número total de iterações do algoritmo híbrido dado *a priori* e np o número de unidades de processamento envolvidas na execução do algoritmo híbrido paralelo.

No caso em que nte não for divisível por np, já que o número de iterações é uma valor inteiro, teríamos uma situação em que existiriam algumas unidades de processamento

executando um número de iterações maior do que outras. Isso acabaria fazendo com que algumas unidades executassem mais rápido do que outras e ficassem esperando (*idle time*) por aquelas que ainda não tivessem terminado o seu trabalho.

Afim de evitar que haja uma distribuição desigual da carga de trabalho (desbalanceamento de carga) entre as unidades de processamento envolvidas na execução do programa em paralelo, podemos redefinir a Eq.(1) de forma que o número total de iterações (nte) não seja mais informado, mas sim obtido indiretamente em função de um número de iterações individuais (ni) dado *a priori*, de modo que todas as unidades de processamento executem um mesmo número de iterações.

$$nte = ni \times np \tag{2}$$

3.1. Algoritmo híbrido paralelo

Em uma primeira abordagem, o algoritmo híbrido paralelo pode ser implementado utilizando a troca de mensagens - técnica computacional paralela que permite a transferência de dados, distribuição de trabalhos e sincronização de ações entre processos paralelos, realizados através da chamada de funções de envio (*Send*) e de recebimento (*Receive*) de mensagens -, utilizando a biblioteca MPI (*Message Passing Interface*) (Pacheco, 1997 e 1998; Snir *et al.*, 1996).

O modelo de programação utilizado na implementação paralela do algoritmo híbrido é o SPMD (*Single Program Multiple Data*), em que um conjunto de unidades de processamento são reunidas através de um canal de comunicação físico (p.ex. barramento, cabo de rede, etc.), onde cada uma das unidades recebe uma cópia de um mesmo programa (*Single Program*) mais os dados necessários para o seu processamento, que são divididos entre os processadores (*Multiple Data*).

Quando o programa paralelo é executado, as unidades de processamento recebem um número de processo, ou *rank*, que as identifica e, tão logo podem, iniciam a execução do programa em paralelo, independente dos demais processadores. Quando necessário, as unidades podem parar a execução normal do programa para fazerem trocas de dados e/ou controles com as outras unidades de processamento.

Para que haja um controle centralizado das ações e um ponto de distribuição e coleta de dados, elege-se uma das unidades de processamento, também conhecida como *root*, para executar essa tarefa. Normalmente, a unidade escolhida é aquela cujo rank é igual a zero (*rank*=0).

De modo a minimizar os impactos causados pelo tempo de comunicação gasto com a troca de dados (*overhead* de comunicação) entre as unidades de processamento e o *root*, ao invés de transmitir todas as raízes encontradas em cada unidade de processamento, pode-se diminuir a quantidade de dados a serem enviados, retirando-se as raízes repetidas do conjunto de soluções obtidas em cada unidade de processamento, reduzindo-as para um conjunto menor de raízes distintas, que deverá ser transmitido para o *root*, onde se reunirão todas as soluções para serem selecionadas (retira-se as raízes repetidas do conjunto completo e, escolhe-se aquela que minimiza a função-objetivo) e, finalmente, será apresentado como o conjunto de raízes solução do algoritmo híbrido paralelo (Fig.4).

1: Defina um número de iterações individuais ni
2: Defina R_i = vetor de raízes obtidas pelo processo de *rank*=i
3: Para k=1 até ni faça
4: Execute a rotina Luus-Jaakola
5: Faça $x_{inicial}(\text{Hooke-Jeeves}) = x_{inicial}(\text{Luus-Jaakola})$
6: Execute a rotina Hooke-Jeeves
7: Faça $r_{ik} = x_{final}(\text{Hooke-Jeeves}), r_{ik} \in R_i$
8: Fim_para
9: Filtre as raízes repetidas em R_i, classificando-as por aproximação
10: Minimize cada classe de vetores, gerando um valor por classe (raízes distintas)
11: Envie as raízes distintas para o processo *root*
12: *Se root então*
13: Defina S = vetor solução
14: *Para i=0 até np faça*
15: *Receba* as raízes distintas do processo i
16: Fim_para
17: Filtre as raízes repetidas, selecionando aquelas que apresentam um melhor resultado
18: Copie as raízes distintas de R em S
19: Imprima o vetor solução S
20: Fim_se

Figura 4 - Algoritmo híbrido paralelo.

4. TESTES E ANÁLISE DE RESULTADOS

4.1. Arquitetura empregada

Os testes e os resultados apresentados nesse trabalho foram realizados e obtidos em uma estação de trabalho instalada no Laboratório de Experimentação e Simulação Numérica em Transferência de Calor e Massa (LEMA) do Campus Regional Nova Friburgo, IPRJ/UERJ.

O equipamento é um Dell Precision Workstation T7400, com dois processadores Intel® Xeon® Quad-Core E5405 2 GHz, 2x6 MB L2 cache, 1333 MHz FSB - totalizando oito unidades de processamento -, Memória de 4 Gb DDR2 SDRAM 667MHz.

As aplicações em paralelo foram processadas sob o sistema operacional Linux (Ubuntu 14-02 LTS), usando o pacote MPICH (versão 3.0.4).

4.2. Funções-teste

Para os teste de execução do programa em paralelo, foram usados as mesmas funções-teste empregadas na tese de Silva (2009), apresentadas na Tabela-1 dada abaixo.

Tabela 1 - Funções-teste.

Nº	Função-teste	Sistemas de equação	Nº de raízes da função
1	Bini e Mourrain (BM) (Bini e Mourrain, 2004)	$\begin{cases} -x_2^2x_3^2 - x_2^2 + 24x_2\ x_3\ - x_3^2 - 13 = 0 \\ -x_1^2x_3^2 - x_1^2 + 24x_1\ x_3\ - x_3^2 - 13 = 0 \\ -x_1^2x_1^2 - x_1^2 + 24x_1\ x_2\ - x_2^2 - 13 = 0 \\ 0 < x_1\ , x_2\ x_3\ , x_4\ , x_5\ \leq 20 \end{cases}$	8
2	Himmelblau (HIMM) (More *et al*., 1981)	$\begin{cases} 4x_1^3 + 4x_1\ x_1\ + 2x_2^2 - 42x_1\ - 14 = 0 \\ 4x_2^3 + 2x_1^2 + 4x_1\ x_2\ - 26x_1\ - 22 = 0 \\ -5 < x_1\ , x_2\ < 5 \end{cases}$	9
3	Sistema Quase Linear de Brown (PQL) (Grosan e Abraham, 2008)	$\begin{cases} 2x_1\ + x_2\ + x_3\ + x_4\ + x_5\ - 6 = 0 \\ x_1\ + 2x_2\ + x_3\ + x_4\ + x_5\ - 6 = 0 \\ x_1\ + x_2\ + 2x_3\ + x_4\ + x_5\ - 6 = 0 \\ x_1\ + x_2\ + x_3\ + 2x_4\ + x_5\ - 6 = 0 \\ x_1\ x_2\ x_3\ x_4\ x_5\ - 1 = 0 \\ -10 < x_1\ , x_2\ x_3\ , x_4\ , x_5\ < 10 \end{cases}$	12
4	Sistema Polinomial de Alto Grau (SPAG) (Kearfott, 1987)	$\begin{cases} 5x_1^9 - 6x_1^5x_2^2 + x_1\ x_2^4 + 2x_1\ x_3\ = 0 \\ -2x_1^6x_2\ - 2x_1^2x_2^3 + 2x_2\ x_3\ = 0 \\ x_1^2 + x_2^2 - 0{,}265625 = 0 \\ -0{,}6 < x_1\ < 6 \\ -0{,}6 < x_2\ < 0{,}6 \\ -5 < x_3\ < 5 \end{cases}$	12
5	Sistema trigonométrico (ST) (Hirsch *et al*., 2006)	$\begin{cases} -\sin(x_1)\cos(x_2) - 2\cos(x_1)\sin(x_2) = 0 \\ \cos(x_1)\sin(x_2) - 2\sin(x_1)\cos(x_2) = 0 \\ 0 < x_1\ , x_2\ < 2\pi \end{cases}$	13

Para que os sistemas de equações (Tab.1) possam ser resolvidos pelo método híbrido, é preciso transformar as equações das funções-teste em um problema de minimização, elevando cada equação ao quadrado e convertendo-as em parcelas de uma soma, de modo a formar uma função-objetivo não negativa.

$$F(x) = \sum_{i=0}^{n} [f_i(x)]^2 \tag{3}$$

Cada um dos sistemas de equações é, então, transformado em um problema de minimização, onde os mínimos globais nulos da função-objeto (Eq.3) passam a representar as soluções do sistema original. Dessa forma, o problema assume o seguinte formato:

$$L(x) = \min_x \sum_{i=0}^{n} [f_i(x)]^2 \tag{4}$$

4.3. Resultados

Na avaliação de desempenho, os tempos de execução apresentados (Fig.5) foram obtidos através da média aritmética simples dos tempos alcançados em três execuções completas do programa paralelo para cada uma das funções-teste apresentadas na Tabela-1.

Em cada execução, foram realizadas 840 avaliações (nte=840). Esse número foi utilizado por produzir quantidade inteiras de avaliações individuais (ni={480, 280, 210, 168, 140, 120, 105}) nos arranjos das unidades de processamento executando em paralelo (np={2,3,4,5,6,7,8}).

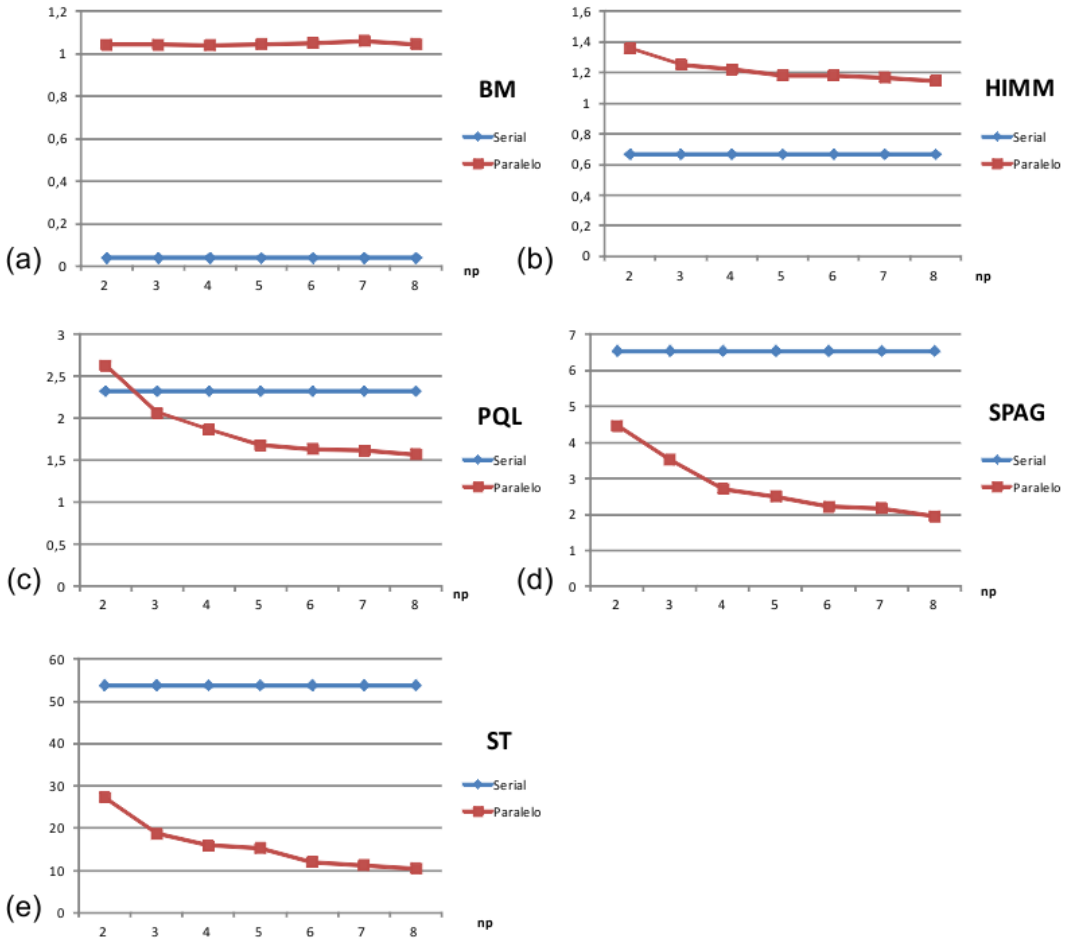

Figura 5 - Tempos de processamento (seg.).

Nos gráficos de tempos de processamento (Fig.5), observa-se que o desempenho do programa paralelo foi particularmente ruim para o grupo de funções que apresentam tempos computacionais muito pequenos (< 2seg.) quando executados serialmente. Para esses casos, o emprego do programa serial é o mais indicado.

Esse desempenho ruim explica-se em razão do *overhead* (sobrecarga) introduzido pelo tempo gasto no processamento das funções MPI e, principalmente, pelo tempo consumido com a comunicação de dados realizado entre as unidades de processamento.

No entanto, quando as funções-teste demandam um tempo maior para serem processadas, os *overheads* do programa paralelo são compensados com os ganhos do paralelismo. Assim,

quanto maiores são os tempos de processamento das funções-teste, maiores são os ganhos do programa paralelo, justificando o seu uso nesses casos.

O valor da aceleração (*speedup*), apresentado nos gráficos da Figura-6, foi obtido através da Equação-5, também conhecida como aceleração absoluta (*absolute speedup*), que usa o tempo de processamento $T_s(x)$ alcançado pelo programa serial e o tempo obtido pela execução do programa paralelo $T_p(x, np)$, empregando np unidades de processamento.

$$S_p(x) = \frac{T_s(x)}{T_p(x, np)} \tag{5}$$

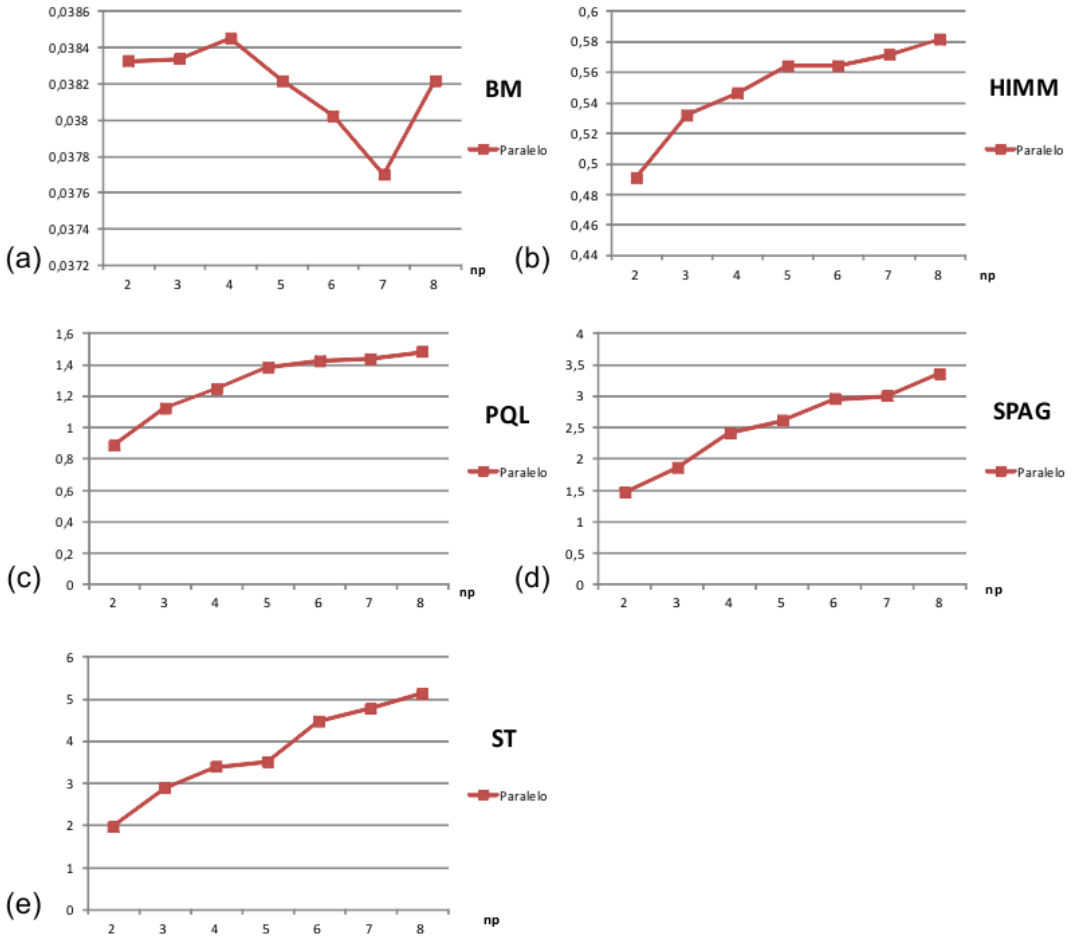

Figura 6 - *Speedups*.

Aqui mais uma vez, verifica-se que o desempenho do programa paralelo é maior nos casos em que as funções-objetivo demandam um esforço computacional maior para serem processadas.

5. CONCLUSÃO

Em uma primeira abordagem de paralelização do algoritmo híbrido, o programa paralelo mostrou-se promissor. Nos casos em que a função-objetivo demande um pequeno esforço computacional para ser processada, aconselha-se o uso do programa serial, visto que este apresenta um desempenho melhor nesses casos.

Para as situações em que a função-objetivo demande maiores esforços computacionais, consumindo maiores tempos de processamento, o programa paralelo apresenta um melhor desempenho em relação à versão serial, justificando o seu emprego para estes casos.

Longe de esgotar as possibilidades e a aplicação de outras abordagens de paralelização para se alcançar níveis maiores de paralelismo e, portanto, melhores desempenhos computacionais, essa primeira versão do programa paralelo do algoritmo híbrido é uma ferramenta útil para a solução de problemas que exijam um nível maior de esforço computacional, com uma característica intrínseca muito importante, que é a escalabilidade do programa paralelo. O programa foi desenvolvido de forma a permitir o aumento do número de unidades de processamento envolvidas na execução do algoritmo híbrido em paralelo.

Agradecimentos

O autor agradece ao apoio do Prof. Antônio José da Silva Neto pela cessão do equipamento do laboratório LEMA (Laboratório de Ensaios Mecânicos) localizado no Instituto Politécnico (IPRJ/UERJ), Campus Nova Friburgo, usado na realização dos testes computacionais apresentados nesse trabalho.

REFERÊNCIAS

Bini, D.A.; Mourrain, B. “Cyclo”, from polynomial test suite. Disp. em: http://www-sop.inria.fr/saga/POL/BASE/2.multipol/cyclohexan.html. Acesso em: 2014.

Bondi, A.B. Characteristics of Scalability and Their Impact on Performance. Proceedings of the 2nd International Workshop on Software and Performance,Ottawa, Ontário, Canadá, 2000, p. 195-203. ISBN 1-58113-195-X.

Grama, A.; Gupta, A.; Karypis, G.; Kumar, V. Introdution to Parallel Computing. 2ed. Pearson. 2003.

Grosan C.; Abraham A. Multiple solutions for a system of nonlinear equations. International Journal of Innovative Computing, Information and Control, v. 4, n. 9, sept. 2008.

Hirsch M. J.; Meneses C. N.; Pardalos P. M.; Resende M.G.C. Global optimization by continuous grasp. Mar. 8, 2006. Disp. em: http://www.optimization-online.org/DB_FILE/2006/03/1344.pdf. Acesso em: 2014.

Hooke, R.; Jeeves, T.A. (1961) "*direct search solution of numerical and statistical problems*", Journal of the Association for Computing Machinery, v. 8, p. 212-229.

Kearfott, R. B. Abstract generalized bisection and a cost bound. Math. Comput. v. 49, n.179, 1987.

Luss, R.; Jaakola, T. (1973) "*Optimization by direct search and systematic reduction of the size of search region*", AIChE Journal, vol. 19.

McCool, M.; Robinson, A.D.; Reinders, J. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Pub. 2012.

More J.J.; Garbow, B.S.; Hillstrom, K.E. Testing Unconstrained Optimization Software, ACM Transactions on Mathematical Software, 7, 136, 1981

Moore, G.E. (1965). "*Cramming more components onto integrated circuits*". Electronics Magazine. p.4. 19 April 1965.

Pacheco, P.S. *Parallel Programming with MPI*. Morgan Kaufmann Pub. 1997. ISBN: 1-55860-339-5.

Pacheco, P.S. *A User's Guide to MPI*. 1998. Disp. em: ftp://math.usfca.edu/pub/MPI/mpi.guide.ps. Acesso em: 19/08/2014.

Silva, M. R. (2011) "*Um novo método híbrido aplicado à solução de sistemas não-lineares com raízes múltiplas*", XLIII Simpósio Brasileiro de Pesquisa Operacional, Ubatuba/SP.

Silva, M. R. "*Um novo método híbrido aplicado à solução de sistemas não-lineares com raízes múltiplas*", Tese de Doutorado, UERJ, Nova Friburgo, RJ, 2009.

Snir, M.; Otto, S.; Huss-Lederman, S.; Walker, D.E.; Dongarra, J. *MPI: The Complete Reference*. Cambridge: The MIT Press, 1996.

Sutter. H. *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*. Dr. Dobb's Journal, 30(3), March 2005. Disp. em: http://www.gotw.ca/publications/concurrency-ddj.htm. Acesso em: 26/08/2014.

PARALLELIZATION OF A HYBRID ALGORITHM FOR A SOLUTION OF NONLINEAR SYSTEMS WITH MULTIPLE ROOTS

Abstract. *This paper presents a strategy for parallel implementation of a hybrid algorithm for the solution of nonlinear systems with multiple (roots) solutions, the model is constructed from two methods: Luus-Jaakola - used to generate initial points as possible candidates the system solution - and Hooke-Jeeves, used to produce a more refined solution (optimum) obtained from these points.*

Keywords: *Optimization,hybrid algorithms , parallel programming*