# APLICAÇÃO DO PARALELISMO À RESOLUÇÃO NUMÉRICA DE PROBLEMAS DE OTIMIZAÇÃO NÃO LINEAR

Luiz Antonio Alves de Oliveira

Universidade Federal do Rio de Janeiro
Instituto de Matemática – NCE

Tese de MSc

Professor Orientador:

João Lauro Dorneles Facó

Dr - Ing.

RIO DE JANEIRO, RJ - BRASIL
2002

# APLICAÇÃO DO PARALELISMO À RESOLUÇÃO NUMÉRICA DE PROBLEMAS DE OTIMIZAÇÃO NÃO LINEAR

Luiz Antonio Alves de Oliveira

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO PROGRAMA DE PÓS-GRADUAÇÃO DO INSTITUTO DE MATEMÁTICA E NÚCLEO DE COMPUTAÇÃO ELETRÔNICA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO – UFRJ, COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM INFORMÁTICA.

Aprovada por:

_____

João Lauro Dorneles Facó, Dr – Ing. - Professor Orientador

_____

Maurício G. C. Resende, Ph.D.

_____

José Herskovits Norman, Dr – Ing.

_____

Abílio Pereira de Lucena Filho, Ph.D.

RIO DE JANEIRO, RJ - BRASIL
2002

Oliveira, Luiz Antonio Alves de.

Aplicação do paralelismo à resolução numérica de problemas de otimização não linear. Rio de Janeiro, 2002.

v, 63f. ; 29,7 cm

Orientador Prof. João Lauro Dorneles Facó

Dissertação (Mestrado) – Universidade Federal do Rio de Janeiro, IM-NCE, 2002.

1. Programação paralela. 2. Otimização não linear 3. Controle ótimo discreto I. Facó, João Lauro Dorneles. II. Universidade Federal do Rio de Janeiro. Instituto de Matemática. III Universidade Federal do Rio de Janeiro. Núcleo de Computação Eletrônica. IV. Título.

Agradecimentos

A minha mãe Judith Alves e minha irmã Cintia Alves pelo eterno carinho e ajuda.

A minha noiva, Regina Carvalho pela ajuda e apoio nos momentos de dificuldades.

Ao meu orientador, professor Facó pela sua paciência, compreensão e orientação segura.

Aos amigos de turma Raquel Defelippo, Denise Candal e André Martins pelo espírito de equipe e solidariedade.

Aos amigos de trabalho Alexandre Tavares, Denise Correa, Edson Santos, Fabio de Azevedo e Paulo Roberto pelo apoio ao meu desenvolvimento profissional.

A Dona Deise e toda sua equipe pela ajuda e atendimento exemplar.

Resumo da tese apresentada ao IM-NCE/UFRJ como parte dos requisitos necessários a obtenção do grau de Mestre em Ciências (M. Sc.)

**Aplicação do paralelismo à resolução numérica de problemas de otimização não linear**

Luiz Antonio Alves de Oliveira
Junho/2002

Orientador: João Lauro Dorneles Facó

Os métodos de programação não linear são, em princípio, seqüenciais.Talvez por este motivo seja difícil encontrar na literatura algoritmos paralelos para resolver problemas não lineares. No entanto há pelo menos duas partes integrantes de muitos métodos de programação não linear onde parece vantajoso implementar técnicas de paralelismo:

(i)Na rotina de resolução de sistemas de equações lineares (Gauss/LU);

(ii)Na rotina de computação da direção de busca pelo método do gradiente conjugado.

O objetivo desta tese é verificar estas hipóteses para o caso de programação não linear geral.

Neste sentido será utilizado o software LSGRG2 (Leon S. Lasdon, University of Texas) onde substituiremos a rotina de inversão da base por um LU paralelo e a direção de busca por um método de gradiente conjugado paralelo.

Abstract of the thesis presented to IM-NCE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M. Sc.) in Computer Science

**Application of parallel computing to the numerical resolution of nonlinear optimization problems**

Luiz Antonio Alves de Oliveira

June/2002

Advisor: João Lauro Dorneles Facó

Nonlinear programming methods are, in principle, sequential. Possibly for this reason it is difficult to find parallel algorithms in the literature to solve nonlinear problems. However, there are at least two components of most nonlinear programming methods that seems suitable to be implemented by parallel techniques:

(i) The resolution of the linear equation systems (Gauss/LU);

(ii) The conjugate gradient method to compute the search direction.

This objective of this thesis is to check these hypothesis for the general nonlinear programming problem.

Thus, the LSGRG2 software (Leon S. Lasdon, University of Texas) is used, insuch away that two parts of it are replaced : the basis matrix inversion routine is replaced with a parallel LU algorithm, and the nonlinear search direction is replaced with a parallel conjugate gradient.

# Sumário

# 1. Introdução

Históricamente, sempre existiu a necessidade de resolver problemas nos menores tempo e custo possíveis. Esta necessidade fomentou o desenvolvimento da ciência e da tecnologia.

A programação matemática, embora tenha surgido há mais tempo, somente na segunda metade do século passado, com o desenvolvimento da ciência da computação e dos computadores eletrônicos, tornou-se ferramenta essencial para a análise dos problemas e sua resolução computacional com apoio à tomada de decisão.

Um dos objetivos da programação matemática, como ramo da ciência, é de propor algoritmos eficientes para a resolução de problemas de decisão. Ao longo desse período novos algoritmos têm sido pesquisados, como por exemplo, os algoritmos GRG (Gradiente Reduzido Generalizado) e GRECO (Gradiente REduzido para Controle Ótimo), sobre os quais falaremos no decorrer deste trabalho.

O avanço tecnológico é responsável pelo surgimento de novas gerações de equipamentos objetivando maior velocidade de processamento e maior capacidade de armazenamento, e consequentemente, pelo desenvolvimento de *software* para controle e gerência dos sistemas de computação.

Com a possibilidade de decomposição dos problemas em partes individualizadas e independentes fica viável executar a resolução de cada uma destas partes de forma simultânea através da computação paralela, e com isso ter ganho de performance em relação à computação seqüencial tradicional.

Esta tese visa verificar as vantagens em implementar técnicas de paralelismo para o caso de resolução numérica de problemas programação não linear com restrições.

# 2. Paralelismo

## 2.1. Conceitos básicos

O conceito de paralelismo não é novo. Em 1842, Menabrea [22] já sugeria a possibilidade de execução de instruções em paralelo na máquina analítica de Charles Babbage. A necessidade de resolver problemas cada vez mais complexos sempre esteve fomentando o desenvolvimento de novas máquinas.

A computação serial utiliza um único processador para executar em ordem uma sequência de instruções e obter um resultado. Em qualquer momento existe apenas uma única instrução sendo executada pelo processador.

Já a computação paralela preocupa-se em produzir os mesmos resultados utilizando-se de vários processadores. Em síntese, é o mesmo que afirmar que um problema deve ser dividido em pequenas unidades independentes. A decomposição do problema é crítica para a performance da computação e o algoritmo responsável pela mesma é eficaz na medida em que mantém todos os processadores ocupados e minimiza a comunicação entre eles. Portanto, **paralelização** consiste em decompor um problema em tarefas e alocá-las a cada processador disponível. Assim, os problemas de decomposição simples são candidatos naturais à paralelização.

Entende-se pela expressão **processamento paralelo** a idéia de um único programa rodando simultaneamente em vários processadores.

Para o usuário final, a principal preocupação é a combinação custo e performance, isto é, o *hardware* não pode ser excessivamente caro como também as aplicações devem ter performance aceitável.

Máquinas paralelas podem oferecer alta performance a um custo mais baixo que máquinas seriais que utilizem processadores específicamente desenvolvidos. Adicionalmente a escalabilidade das máquinas paralelas permite que as atualizações ocorram de acordo com as necessidades.

Numa máquina serial a atualização ocorre necessáriamente pela substituição do processador enquanto que na máquina paralela uma das atualizações possíveis seria o acréscimo de um novo processador. Máquinas seriais possuem sua performance limitada, isto é a velocidade na qual a informação trafega no processador é limitada pela velocidade da luz.

Por estas razões, a computação de alto desempenho está associada à idéia de paralelismo.

## 2.2. Arquiteturas paralelas

Os diferentes tipos de arquiteturas de computadores possibilitam estratégias distintas para explorar o paralelismo em uma máquina. As estratégias mais comuns são o aumento do número de unidades funcionais do processador e o aumento do número de processadores no sistema, podendo-se, também utilizar ambas as estratégias. Dependendo da estratégia adotada o paralelismo pode ocorrer em níveis : paralelismo à nível de instruções, paralelismo a nível de dados e paralelismo a nível de programação.

Atualmente existem vários modelos de classificação de arquiteturas, porém o modelo proposto em 1966 por Flynn [11], é ainda atual e apresenta as vantagens de ser simples, fácil de entender e fornece uma boa aproximação da realidade.

A taxonomia de Flynn considera o fluxo de instruções executadas em paralelo e o fluxo de dados tratados em paralelo. A classificação é mostrada a seguir :

- SISD (*Single Instruction stream / Single Data stream*)
  (Um fluxo de instruções atuando sobre um fluxo de dados)
- SIMD (*Single Instruction stream / Multiple Data stream*)
  (Um fluxo de instruções atuando sobre vários fluxo de dados)
- MISD (*Multiple Instruction stream / Single Data stream*)
  (Mútiplos fluxos de instruções atuando sobre um fluxo de dados)
- MIMD (*Multiple Instruction stream / Multiple Data stream*)
  (Mútiplos fluxos de instruções atuando sobre vários fluxos de dados)

Até o momento, não se tem conhecimento de implementação da arquitetura MISD. Não a comentaremos neste trabalho.

### 2.2.1. Arquitetura SISD

As máquinas SISD são os computadores seriais convencionais. Um processador serial executa apenas um fluxo de instruções sobre uma unidade de dados a cada passo. Sem perda de generalidade podemos dividir um computador em pelo menos três unidades básicas a saber : memória principal, unidade central de processamento (CPU) e um sistema de entrada e saída (I/O). A CPU consiste em um conjunto de registradores, um *program counter* e uma unidade lógica e aritmética (ALU) onde as operações são realizadas uma de cada vez.

Mesmo nas máquinas SISD o paralelismo pode ser explorado ao nível de instruções. Um primeiro passo seria dividir a ALU por funcionalidade. Por exemplo, unidades de ponto flutuante para adição e multiplicação operando em paralelo. É necessário que o compilador utilize as múltiplas unidades funcionais na exploração do

paralelismo. Este trabalho consiste em escalonar as operações dentre as múltiplas unidades funcionais, toda vez que o *hardware* estiver ocupado, ou seja, as operações aritméticas independentes e de funcionalidade distintas , quando encontradas pelo compilador, são escalonadas no fluxo de execução de tal forma que cada operação é realizada em uma unidade funcional em paralelo com as demais.

O *pipelining* é a técnica de segmentação das unidades funcionais em diferentes partes cada uma responsável pelos seguintes procedimentos : busca da instrução na memória, decodificação/interpretação e execução. Uma analogia ao *pipelining* é uma linha de montagem. O *pipelining* é realizado pela divisão de uma tarefa em uma sequência de pequenas tarefas denominadas estágios do *pipelining*, cada estágio é executada por uma parte do *hardware* que opera concorrentemente com os outros estágios. Esta técnica é tão antiga quanto os computadores e a cada nova geração emprega novas variações.

Outra técnica semelhante ao *pipelining* é o *overlapping* onde operações, que podem ser executadas em unidades funcionais independentes, são sobrepostas. O *pipelining* como o *overlapping* partem da idéia de particionamento mas em contextos diferentes. O *pipelining* ocorre quando todas as seguintes características estão presentes : cada cálculo é independente do cálculo anterior ; cada cálculo necessita da mesma sequência de estágios ; os estágios estão relacionados e o tempo de computação de diferentes estágios é aproximadamente o mesmo. Já o *overlapping* ocorre quando qualquer uma das seguintes características estão presentes : cada cálculo pode necessitar de uma sequência diferente de estágios ; os estágios são relativamente distintos com respeito ao seu propósito ; existe alguma dependência entre cáculos e o tempo por

estágio não é necessariamente constante, porém é uma função do estágio e dos dados passados.

### 2.2.2. Arquitetura SIMD

Nesta arquitetura, todos os elementos de processamento recebem a mesma instrução de uma única unidade de controle, porém operam em diferentes conjuntos de dados (módulos de memória em paralelo). As máquinas SIMD, freqüentemente denominadas *array processors*, permitem expressar o paralelismo a nível de dados. A experiência tem mostrado limitações nas máquinas SIMD na resolução de problemas científicos. Alta performance só é alcançada em alguns tipos de aplicações como por exemplo o processamento de imagens, porém a grande vantagem das máquinas SIMD é a sincronização, pois existe apenas uma única seqüência de instruções.

As máquinas SIMD operam sobre vetores de dados e com uma única instrução vetorial pode-se realizar operações do mesmo tipo em cada elemento do vetor. Neste contexto vetor é uma lista ordenada de escalares.

A computação vetorial é um modelo muito semelhante ao SIMD, a diferença está no uso de técnicas avançadas de *pipeline* pelos processadores vetoriais. Os processadores vetoriais possuem operações de alto nível que operam sobre vetores.

As vantagens dos processadores vetoriais sobre os processadores SISD são as seguintes :

- Uma única instrução vetorial realiza uma boa quantidade de trabalho, assim menos dados serão buscados da memória e menos instruções de desvio serão executadas
- As instruções vetoriais buscam um bloco de dados a cada acesso à memória

### 2.2.3. Arquitetura MIMD

As máquinas MIMD são o sustentáculo da atual computação científica paralela. Consistem de vários processadores independentes executando um fluxo de instruções e atuando sobre dados distintos. A arquitetura MIMD é dividida em duas subclasses : **memória compartilhada** e **memória distribuída**.

A subclasse memória compartilhada é composta por processadores e módulos de memória conectados por um *hardware* de alta performance tais como *crossbar switch* ou uma rede de roteamento eficiente. Este tipo de arquitetura é aplicável quando um número pequeno de processadores é utilizado. Todos os processadores compartilham todos os módulos de memória e existe a possibilidade de executar diferentes instruções em cada processador utilizando diferentes fluxos de dados.

O termo memória compartilhada entende-se por um espaço de endereçamento único acessado por todos os processadores do sistema, ou seja, cada posição de memória pode ser lida ou gravada por qualquer processador. Assim a comunicação entre processadores é feita por intermédio de variáveis compartilhadas armazenadas na memória.

Nesta subclasse, existe a necessidade de estabelecer uma sincronização para tornar possível a operação sobre dados compartilhados. Um dos métodos de sincronização é a operação de *lock* : apenas um processador pode adquirir, por meio da execução bem-sucedida do *lock*, o direito de acessar dados compartilhados, sendo que todos os demais processadores que tentem acessar o mesmo dado compartilhado devem esperar até que o processador original execute a operação *unlock*.

Nesta arquitetura, os sistemas com espaço de endereçamento único são também classificados com respeito ao acesso à memória. O primeiro tipo UMA (*Uniform Memory Access*) gasta o mesmo tempo para acessar a memória principal, não importando qual dos processadores requisitou o acesso, nem o tipo de trabalho requisitado. O segundo tipo NUMA (*Nonuniform Memory Access)* alguns dos acessos à memória são mais rápidos do que outros.

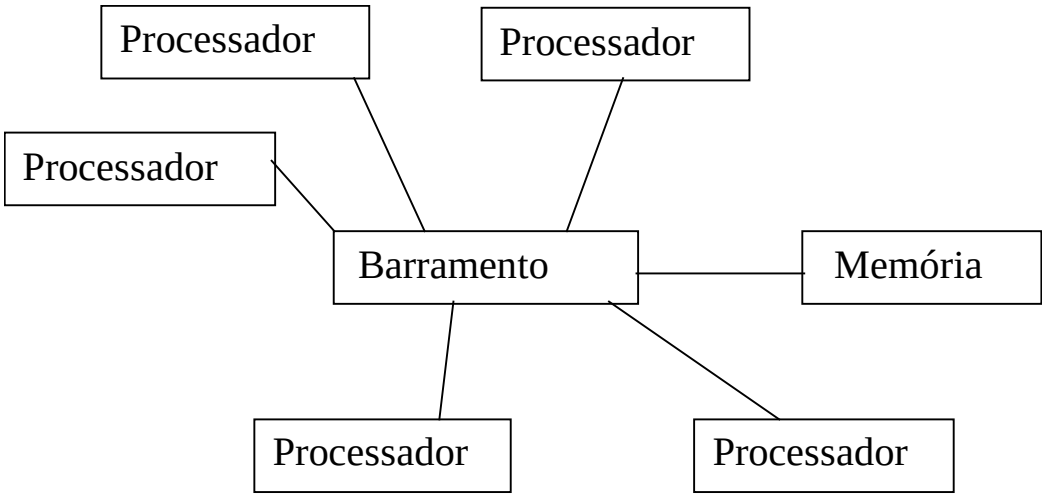


Figura 1.1 – Arquitetura MIMD com memória compartilhada

A subclasse memória distribuída é composta por nós de processamento, cada um contendo um ou mais processadores, memória local e uma interface de comunicação com outros nós para troca de mensagens. Neste tipo de arquitetura, também denominada memória privada, é possível utilizar um número grande de nós e não existe compartilhamento de memória entre nós, toda troca de dado entre nós deve utilizar a interface de comunicação. Esta subclasse é escalável, ou seja, dependendo da

necessidade das aplicações pode-se agregar mais nós ao sistema ou aumentar a memória local de cada nó.

A comunicação entre os nós é feita através de troca de mensagens, se um nó deseja acessar ou operar um determinado dado localizado em outro nó, deve enviar uma mensagem. O nó destinatário executa a operação necessária e retorna uma mensagem para o nó remetente. Existe então um padrão para troca de mensagens denominado MPI [28] que oferece ao programador todo tipo de operação de comunicação.

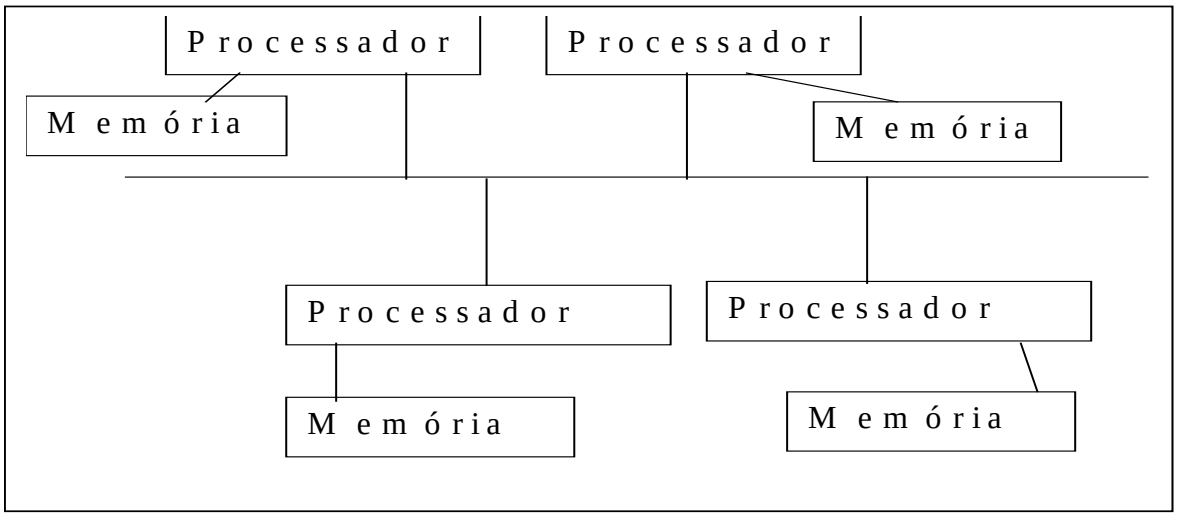


Figura 1.2 – Arquitetura MIMD com memória distribuída

## 2.3. Performance

A velocidade de computação é normalmente expressa em termos de Mflops. Por definição Mflops é a razão entre o número (*N*) de operações de ponto flutuante realizadas pelo tempo (*t*) de computação em microsegundos, assim a performance é dada pela velocidade de computação (*r*) :

$$r = \frac{N}{t} Mflops \tag{2.1}$$

Da mesma forma, quando *N* operações de ponto flutuante são executadas com velocidade média de *r* Mflops, então o tempo de processamento é :

$$t = \frac{N}{r}\mu seg \tag{2.2}$$

Diferentes instruções podem ser executadas em tempos diferentes. Isto depende, do *pipeline*, do acesso à memória e de outros fatores. É incorreto realizar a análise da performance de um algoritmo considerando tão somente a quantidade de operações de ponto flutuante. O fato de que instruções realizadas por um algoritmo, são executadas em tempo diferente. Implica em que quando duas partes de uma tarefa são executadas, cada uma com velocidade diferente, o tempo total de processamento pode ser expresso como uma função destas velocidades.

Amdahl [3], percebeu que a baixa velocidade de execução de uma tarefa influência na performance global do sistema. A relação entre performance e tempo de processamento é frequentemente conhecidada como **Lei de Amdahl**.

Para o caso simples da Lei de Amdahl, assumimos que a execução de um algoritmo envolve *N* operações de ponto flutuante e que em um determinado computador *f* destas operações são realizadas à velocidade de *V* Mflops enquanto o resto é executado à taxa de *S* Mflops. Quando que $V >> S$ , o tempo de processamento de acordo com ( 2.2) é expresso por

$$t = \frac{fN}{V} + \frac{(1-f)N}{S} = N\left(\frac{f}{V} + \frac{1-f}{S}\right)\mu seg \tag{2.3}$$

De acordo com (2.1) temos a performance global $r$ , comprovando a Lei de Amdahl :

$$r = \frac{N}{t} = \frac{1}{\left( \frac{f}{V} + \frac{(1-f)}{S} \right)} Mflops \tag{2.4}$$

Por (2.3) temos que :

$$t > \frac{(1-f)N}{S} \mu seg \tag{2.5}$$

Se o algoritmo for executado completamente com a baixa velocidade S, o tempo de processamento deve ser $t = N/S$ . Isto implica que o ganho relativo no tempo de processsamento obtido pela execução da partição $fN$ a velocidade $V$ é limitado por 1 / (1-$f$ ). Para que o ganho de performance tenha significado, $f$ deve estar próximo de 1.

Para o caso geral da Lei de Amdahl, assumimos que um algoritmo consiste na execução consecutiva das partes $P_1$, $P_2$, ..., $P_n$ tais que $N_j$ operações de ponto flutuante da parte $P_j$ são executadas à velocidade de $r_j$ Mflops, então a performance global para $N=N_1+N_2+...+N_n$ é dada por

$$r = \frac{N}{\sum_{j=1}^{n} \left( \frac{N_j}{r_j} \right)} \tag{2.6}$$

para diferentes partes computacionais de uma tarefa.

No processamento paralelo o tempo total do sistema é normalmente mais amplo do que o tempo total de um sistema com um único processador e é muito mais amplo do que o tempo gasto por processador, como o objetivo do processamento paralelo é reduzir o tempo total, deve-se comparar o tempo total para números diferentes de processadores para estudar os efeitos da aceleração.

É possível esperar que, se o processamento pode ser feito em $p$ partes iguais então o tempo total será aproximadamente $1/p$ da execução em apenas um processador. Pela Lei de Amdahl, as partes não paralelizáveis tem influência negativa nesta redução.

Seja $t_j$ o tempo necessário para realizar determinada tarefa com $j$ processadores. O *Speedup*, $S_p$ , para um sistema com p processadores é definido como a razão entre o tempo gasto por um sistema com um processador, $t_1$ e o tempo utilizando p processadores $t_p$ .Pode ser definido tambem como o fator de redução do tempo de processamento para um sistema com p processadores.

$$S_p = \frac{t_1}{t_p} \tag{2.7}$$

A Eficiência $E_p$ é a razão entre o *Speedup* obtido com p processadores e o número de processadores, ou seja é a fração do tempo em que os processadores estão ativos.

$$E_p = \frac{S_p}{p} \tag{2.8}$$

Assumimos por simplicidade que uma tarefa consiste em operações básicas, todas executadas na mesma velocidade e que uma fração $f$ destas operações possam ser executadas em paralelo em $p$ processadores enquanto o resto do trabalho é executado em apenas um processador. O tempo total da parte paralela é então dado por $f\,t_1 / p$ , e ignorando perdas decorrentes da sincronização, o tempo da parte serial é dado por $(1-f)\,t_1$ . Consequentemente o tempo total para $p$ processadores no sistema é

$$t_p = \frac{ft_1}{p} + (1-f)t_1 = \frac{t_1(f + (1-f)p)}{p} \geq (1-f)t_1 \tag{2.9}$$

modificando a formula do *Speedup* $S_p$ temos :

$$S_p = \frac{p}{(f + (1-f)p)} \tag{2.10}$$

(2.10) mostra que o *Speedup* $S_p$ é reduzido por um fator $f+(1-f)p=O(p)$ . Sempre que $f$ for próximo de 1, deve-se considerar a redução pelo aumento do valor de $p$.

Na prática, a situação ideal de (2.10) não ocorre e pode ser complicada pelos fatores:

- O número de partições paralelas de uma tarefa deve ser grande compatível ao número de processadores disponíveis.
- A influência negativa na sincronização não pode ser ignorada.

## 2.4. MPI (*Message Passing Interface*)

O modelo de programação de troca de mensagens está baseado no fato de que existem processadores no sistema com memória local podendo realizar comunicação entre eles para troca de informação e dados. Este modelo é de vital importância para computadores paralelos com memória distribuída.

As características principais do modelo de programação de troca de mensagens são:

- Cada processador possui seu próprio espaço de endereçamento.Dados podem ser transferidos de um espaço de endereçamento para outro.
- Se uma variável é declarada em um programa rodando em $p$ processadores então existem $p$ variáveis diferentes com o mesmo nome e com valores diferentes.
- O desenvolvedor deve dividir a estrutura de dados do problema e distribuí-la entre os diferentes processadores.

O MPI [28] é um padrão de interface de programação, idealizado por pesquisadores com o objetivo de implementar a interface de comuniação entre nós de processamento. É portável, estando presente nas mais diversas máquinas paralelas. Alem disso, o MPI é um padrão tanto no meio acadêmico como na industria.

Como foi dito anteriormente, a troca de mensagens é largamente utilizada nas diferentes implementações de arquiteturas MIMD com memória distribuida. Devido a portabilidade do MPI é possível garantir que um código que utilize a biblioteca MPI numa implementação execute também em outra, desde que a biblioteca esteja disponível.

Esta portabilidade deve-se ao fato da transparência do MPI, ou seja, o conhecimento das características particulares de cada implementação de arquitetura não é necessário para o desenvolvimento das aplicações. Por exemplo, o comando para enviar uma mensagem entre processadores é o mesma para qualquer máquina paralela que tenha a biblioteca.

O desempenho não é prejudicado pela necessidade do padrão MPI ser geral. Na prática o MPI só se tornou um padrão, pelo mercado, por ser eficiente. Isto só foi possível devido aos seguintes fatores :

- O padrão especifica apenas o que a operação faz logicamente, o desenvolvedor da aplicação não precisa preocupar-se com os detalhes da máquina e sim com a aplicação

- Cabe ao desenvolvedor da biblioteca MPI, para uma máquina específica, implementar as diversas operações eficientemente dentro das características próprias da máquina.
- Apenas as informações necessárias são enviadas em cada mensagem.
- Os cabeçalhos de mensagem são facilmente manipuláveis.

O MPI é um padrão adequado para o desenvolvimento de aplicações paralelas para as arquiteturas MIMD com memória distribuída.

## 2.5. IBM – SP2

O computador paralelo IBM SP2 é uma implementação da arquitetura MIMD. Um sistema SP(Scalable POWERparallel) é composto por um ou mais *frames.* Essencialmente um *frame* é um gabinete com *switch* de alta performance dedicado exclusivamente a execução de programas paralelos onde é possível conectar através de *slots*, no máximo dezesseis nós, de diferentes tipos.

É possível estabelecer conexão direta entre qualquer par de nós permitindo boa escalabilidade, baixa latência, largura de banda grande e flexibilidade.

Nó é uma estação de trabalho completa, possuindo processador baseado na arquitetura POWER2 ou POWERPC, memória (cache, principal e secundária), interface de rede própria além de uma cópia do sistema operacional.

Existem três tipos diferentes de nó : *Thin* (ocupa um *slot*) ; *Wide* (ocupa dois *slots* horizontais adjacentes) e *High* (ocupa quatro *slots* adjacentes)

O sistema SP disponível no NCE e utilizado nos testes, possui as seguintes características : 6 processadores com clock de 66.7 Mhz com 256 MB de memória principal ; *switch* de 40 Mbytes/s ; compilador C da própria IBM versão 3.66 ; sistema operacional AIX 4.2 e biblioteca MPI versão 1.0

# 3. Programação não linear

Os primeiros algoritmos de programação não linear eram bastante limitados. Tornaram-se significativos com a introdução dos métodos de máximo declive e de métrica variável, sendo capazes de solucionar problemas de muitas variáveis em pouco tempo.

A programação não linear caracteriza-se por não possuir um método geral de resolução. Seus algoritmos são limitados e voltados para aplicações específicas, sendo então uma área experimental de pesquisa e desenvolvimento. Estes algoritmos são iterativos, gerando soluções parciais a cada passo.

Ao estudarmos um sistema, estamos interessados no desenvolvimento de um modelo matemático, representativo deste sistema, e na sua resolução. O modelo é a síntese da análise de muitas variáveis de decisão onde procuramos a distribuição eficiente de recursos limitados entre atividades competitivas. Assim, procuramos uma solução para um problema de decisão levando em consideração um objetivo bem definido que será a quantificação desta decisão.

Uma decisão está de um modo geral ligada a certo objetivo que pode maximizar ou minimizar a velocidade ou a distância em um problema físico, minimizar custos e maximizar os lucros de produção, dentre outros. Sempre satisfazendo um conjunto de restrições de capital, de recurso humanos, de área, etc.

O modelo matemático que representa o sistema será o conjunto de equações e inequações que quantificam a região de viabilidade da solução. Assim o problema consiste em buscar uma solução que otimize uma função, denominada função-objetivo e que respeite um conjunto de restrições de igualdade e/ou desigualdade.

Seja $X$ o vetor da variáveis de decisão e dado o problema $P_0$ :

($P_0$)

Minimizar $f_0(X)$

Sujeito a $f_i(X) \leq 0 \qquad 1 \leq i \leq m$

Quando qualquer uma das funções $f_i$ , para $0 \leq i \leq m$ , for não linear então o $P_0$ é um problema de programação não linear restrita, no caso de não existir restrições estaremos trabalhando com um modelo de programação não linear irrestrito.

## 3.1. Condições de otimalidade

### 3.1.1. Condições de otimalidade para problemas irrestritos

Para a determinação do ponto de mínimo da função objetivo, uma série de condições necessárias e suficientes devem ser satisfeitas. Apresentaremos estas condições na forma de teoremas que estão demonstrados em [18, 21] :

Teorema 3.1 Seja $S \subseteq R^n$ e $f$ uma função sobre S de classe $C^1$. Se $x^*$ é um ponto de mínimo local de $f$ sobre S então para qualquer $h \in R^n$ , direção viável em $x^*$, temos que $\nabla f(x^*).h \geq 0$

Teorema 3.2 Seja $S \subseteq R^n$ e $f$ uma função sobre S de classe $C^2$ . Se $x^*$ é um ponto de mínimo local de $f$ sobre S então para qualquer $h \in R^n$ , direção viável em $x^*$, temos que

- $\nabla f(x^*).h \geq 0$
- Se $\nabla f(x^*).h = 0$ então $h^t \nabla^2 f(x^*).h \geq 0$

Teorema 3.3 Seja $x^*$ ponto interior ao conjunto S e *f* uma função sobre S de classe $C^2$. Se $x^*$ é um ponto de mínimo local de f sobre S então

- $\nabla f(x^*).h = 0$
- $\forall$ direção $h \in R^n$, $h^t \nabla^2 f(x^*).h \geq 0$

Teorema 3.4 Seja *f* uma função sobre S de classe $C^2$ tal que $x^* \in S$ e $x^*$ é interior. Suponhando que :

- $\nabla f(x^*) = 0$
- $H(x^*)$ é definida positiva

Então $x^*$ é um mínimo local estrito de *f*.

Teorema 3.5 Seja *f* uma função convexa definida sobre um conjunto convexo S. Então o conjunto R de pontos onde, onde *f* atinge seu mínimo, é convexo e qualquer mínimo local de f é um mínimo global.

Teorema 3.6 Seja a função *f* de classe $C^1$ convexa sobre o conjunto convexo S. Se existe um ponto $x^* \in S$ tal que para todo $y \in S$, $\nabla^t f(x^*).(y-x^*) \geq 0$ então $x^*$ é um ponto de mínimo global de *f* sobre S.

Os teoremas 3.1 e 3.2 garantem respectivamente as condições de necessidade de primeira e segunda ordem para existência de um mínimo (local ou global). Os teoremas 3.3 e 3.4 asseguram respectivamente as condições de necessidade e de suficiência para

existência de um ponto interior em S que seja um mínimo local, situação que engloba o caso particular dos problemas irrestritos, em que $S = R^n$ .

O teorema 3.5 garante que com convexidade de *f*(x) o mínimo local é um mínimo global de *f* em S e finalmente, o teorema 3.6 assegura que com convexidade a condição de necessidade de primeira ordem , é também condição de suficiência para existência de um mínimo global de *f* em S.

### 3.1.2. Condições de otimalidade para problemas restritos

Especificaremos as condições de otimalidade associadas ao seguinte problema de programação não linear (PNL) com restrições :

Minimizar $f(X)$

Sujeito a $h_j(X) = 0 \quad 1 \leq j \leq p \quad \lambda_j$ (multiplicadores de Lagrange)

$g_i(X) \leq 0 \quad 1 \leq i \leq m \quad \mu_i \geq 0$ (multiplicadores de KKT)

As condições de necessidade neste caso se tornam as condições de KKT (Karush-Kuhn-Tucker). Para isto a caracterização das direções viáveis $d \in R^n$ é feita em função dos gradientes $\nabla g_i(x)$, dos multiplicadores de KKT $\mu_i \geq 0$ associados às restrições de desigualdade $g_i(x)$, dos gradientes $\nabla h_j(x)$ e dos multiplicadores de Lagrange $\lambda_j$ associados às restrições de igualdade $h_j$.

Estes multiplicadores, chamados também de variáveis duais, têm papel importante na formulação de condições de suficiência que não fazem uso direto nem do conceito de direções viáveis, nem de convexidade e nem de diferenciabilidade.

Estes resultados teóricos têm grande importancia na prática. Podemos repartir as restrições em fáceis e difíceis, deixando as restrições fáceis implícitas no conjunto S e as difíceis explícitas nas funções $g_i(x)$ e $h_j(x)$.

No problema PNL as restrições $g_i(x) \leq 0$ podem se referir à disponibilidade do recurso escasso $i$ , numa situação em que a função objetivo $f(x)$ contabiliza o custo total de outros recursos livres necessários para a operação das atividades ao nível x. Nesse contexto uma abordagem intuitiva para evitar as dificuldades impostas pelas restrições, $g_i(x) \leq 0$ consiste em associar penalidades $\mu_i \geq 0$ à não satistação dessas restrições, adicionando à função objetivo os termos $\mu g(x)$, onde $\mu$ é um vetor linha de variáveis duais e de dimensão m. Essa idéia está presente na função Lagrangeana, definida para $x \in S$ , $\mu \geq 0$ e $\lambda$ por

$$L(x,u,\lambda) = f(x) + \mu g(x) + \lambda h(x)$$

Seja $x \in S$ uma solução específica computada na avaliação da função dual para um vetor específico $\mu \geq 0$.

Condição de qualificação das restrições : Independência linear dos gradientes das restrições de igualdade e das restrições de desigualdade ativas no ponto

As chamadas condições de KKT, que para $(x, \mu, \lambda)$ com $\mu \geq 0$ e $x \in S$ são definidas por:

- $\nabla f(x) + \mu \nabla g(x) + \lambda \nabla h(x) = 0$
- $\mu g(x) = 0$
- $g(x) \leq 0$ e $h(x) = 0$

Lema 3.1 (Farkas) A afirmação de que $bh \geq 0$ para todo h tal que $Ah \leq 0$ é equivalente à afirmar que existe $u \geq 0$ tal que $b + uA = 0$

As condições de KKT resultam diretamente da aplicação do Lema 3.3 a um sistema de desigualdades lineares que relacionam os gradientes da função-objetivo e das restrições num ponto de mínimo x.

Teorema KKT Seja x um ponto de ótimo local do problema PNL, onde S é um conjunto aberto , todas as funções são diferenciáveis, e é satisfeita a condição de qualificação em x. Então existem multiplicadores $\mu_i$ e $\lambda_j$ e tais que as seguintes condições são satisfeitas:

- $\mu_i g_i(x) = 0$ para i = 1, 2, ..., m.
- $\mu_i \geq 0$ para i = 1, 2, ..., m.
- $g_i(x) \leq 0$ para i = 1, 2, ..., m.
- $h_j(x) = 0$ para j = 1, 2, ..., p.
- $\nabla f(x) + \mu_1 \nabla g_1(x) + ... + \mu_m \nabla g_m(x) + \lambda_1 \nabla h_1(x) + ... + \lambda_p \nabla h_p(x) = 0$

## 3.2. Método Gradiente Reduzido Generalizado (GRG)

O método do gradiente reduzido proposto por Wolfe (1963) [32] para minimizar uma função diferenciável não linear sujeita a restrições lineares de igualdade, foi generalizado por Abadie e Carpentier (1969) [2], para o caso de restrições não lineares de igualdade, e variáveis limitadas superior e inferiormente (variáveis canalizadas). O método do gradiente reduzido generalizado (GRG) combina as idéias do método do gradiente reduzido com as estratégias de métodos de linearização das restrições não lineares, sendo considerado um dos mais eficientes para tratar o caso mais geral de programação não linear em que a função-objetivo, bem como as restrições, são não lineares. Segundo [9, 10] o método GRG pode ser especializado para a resolução de problemas de controle ótimo não lineares.

Dado o seguinte problema de programação não linear

$$\text{Minimizar} \quad f_0(X)$$

$$\text{Sujeito a} \quad f(X) = 0 \qquad (3.1)$$

$$a \leq X \leq b \qquad (3.2)$$

onde *X, a, b* são vetores coluna de dimensão N ; *f(X)* é um vetor-coluna de dimensão m correspondente às restrições não lineares; e a função-objetivo $f_0(X)$ é uma número real. $f_0$ e $f$ são continuamente diferenciáveis no paralelotopo P definido por (3.2). Em (3.1) foram introduzidas variáveis de folga nas restrições de desigualdade para transformá-las em igualdade como é usual em programação linear clássica.

O gradiente de $f_0(X)$ é um vetor-linha, denotado por $\nabla f_0(X)$. Analogamente o gradiente de $f_i(X)$, i=1, ..., m é um vetor-linha denotado por $\nabla f_i(X)$. A notação $\nabla f(X)$ representa a matriz jacobiana, uma matriz (mxN) onde cada linha i é o vetor-linha $\nabla f_i(X)$.

O vetor $X$ é particionado em $(x, y)$, onde $y$ é o vetor das variáveis básicas de dimensão m, enquanto $x$ é o vetor das variáveis não básicas ou independentes de dimensão n=N-m.

A notação $\frac{\partial f_o}{\partial x}$ e $\frac{\partial f_o}{\partial y}$ representa o vetor-linha das derivadas parciais de $f_o$ com respeito a $x$ e $y$ respectivamente; a matriz $\nabla f(X)$ é particionada na matriz $\frac{\partial f}{\partial x}$ de dimensão (mxn) e na matriz $\frac{\partial f}{\partial y}$ de dimensão (mxm)

Hipótese de não-degenerescência: Dado um ponto viável $X^0$ existe uma partição de $X^0$ em (x,y), tal que:

(i) $a_j < y^0_j < b_j \quad \forall j \in \{1,2,...,m\}$ (3.3)

(ii) A matriz $\frac{\partial f}{\partial y^0}$ de dimensão (mxn) é não singular (3.4)

## 3.2.1. Algoritmo GRG

Passo 1 – Computar a direção $d^0$ de movimento da variável independente $x$ da seguinte forma :

Passo 1.1 Computar $g^0$ , o gradiente reduzido

$$g^0 = \frac{\partial f_0}{\partial x^0} - \frac{\partial f_0}{\partial y^0}\left(\frac{\partial f}{\partial y^0}\right)^{-1}\frac{\partial f}{\partial x^0} \qquad (3.5)$$

Passo 1.2 Computar $p^0$ , o gradiente reduzido projetado

$$\forall j \in \{1,2,...,n\},\ p^j = \begin{cases} 0 \text{ se } x^0_j = a_j & \text{e} \quad -g^0_j < 0, \\ 0 \text{ se } x^0_j = b_j & \text{e} \quad -g^0_j > 0, \\ g^0_j \text{ senão.} \end{cases}$$

Passo 1.3 Computar $d^0$ , a direção de deslocamento para o vetor independente *x*. Esta direção pode ser simplesmente $d^0 = p^0$ , ou computada por algum outro método de programação não linear irrestrita, como o Gradiente-Conjugado ou quase-Newton.

Passo 2 : Computar $d^0_B$ , direção de deslocamento do vetor básico *y* no plano tangente;

Passo 3 : Computar um primeiro valor para o passo $\theta_1$ ;

Passo 4 : Computar $x^0 - \theta_1 d^0$, e sua projeção no paralelotopo: $a_j < x_j < b_j \ \forall j \in \{1,2,...,m\}$ para obter $\underline{x}^1_j$ isto é,

$$\forall j \in \{1,2,...,m\}, \underline{x}^1_j = \begin{cases} a_j \text{ se } x^0_j - \theta_1 d^0_j < a_j , \\ b_j \text{ se } x^0_j - \theta_1 d^0_j > b_j , \\ x^0_j - \theta_1 d^0_j \text{ senão.} \end{cases}$$

Passo 5 : Restauração da viabilidade não linear: computar um ponto viável $X^1$ resolvendo, com respeito a *y*, um sistema não linear de m equações e m incógnitas, mantendo $\underline{x}^1$ fixo pelo método de Newton:

$$f(\underline{x}^1, y) = 0 \qquad (3.6)$$

Passo 5.1 : Se não for observada convergência, decrementar $\theta_1$ e ir para o passo 4 com o mesmo $d^0$ ;

Passo 5.2 : De outra forma , seja *y'* a solução obtida de (3.6) e $X^1$ o ponto correspondente no espaço de dimensão N,

Passo 5.2.1 : Se $f_0(X^1) \geq f_0(X^0)$ então decremente $\theta_1$ e ir para o passo 4 com o mesmo $d^0$ ;

Passo 5.2.2 : Senão teremos um ponto $X^1$ viável que satisfaz $f_0(X^1) < f_0(X^0)$

Passo 6 : Pode-se fazer $X^1 := X^0$ e começar uma nova iteração, ou tentar aumentar $\theta_1$ e retornar ao passo 3 com o mesmo $d^0$ até que sejam satisfeitas as condições de otimalidade.

## 3.3. Software LSGRG2

LSGRG2 [17] é um *software* que resolve problemas de otimização não linear da seguinte forma:

Minimizar ou Maximizar $g_p(X)$,

Sujeito a:

$glb_i \leq g_i(X) \leq gub_i$ para $i = 0,...,m-1$, $i \neq p$

$xlb_j \leq x_j \leq xub_j$ para $j = 0,...,n-1$

$X$ é um vetor de $n$ variáveis, $x_0 ,...,x_{n-1}$, e as funções $g_0 ,...,g_{m-1}$ são dependentes de $X$.

As funções podem ser não lineares, os limites podem ser infinitos e se não existirem restrições o problema é resolvido como um problema de otimização irrestrita. Os limites inferiores e superiores das variáveis são tratados implicitamente. É utilizado o método do Gradiente Reduzido Generalizado (GRG).

O LSGRG2 usa derivadas parciais de primeira ordem para cada função $g_i$ com respeito a cada variável $x_i$ que são computadas por diferenças finitas ou são fornecidas pelo usuário. Depois que os dados iniciais são fornecidos, o sistema executa duas fases. A fase I é executada se o ponto inicial fornecido pelo usuário for inviável. A função objetivo da fase I é a soma das restrições violadas mais, opcionalmente, uma fração da verdadeira função objetivo. A fase I termina com uma mensagem de que o problema é inviável ou descobrindo um ponto viável. A fase II começa com um ponto viável ,

encontrado pela fase I ou fornecido pelo usuário. Na conclusão da fase II, o processo de otimização é terminado.

Para iniciar o uso do LSGRG2, o usuário deve fornecer funções, escritas na linguagem C, para computar os valores das restrições e da função objetivo. As informações que descrevem o problema devem ser criadas, organizadas e passadas ao LSGRG2. O dados são o número de variáveis; os limites superior e inferior das variáveis; o número de funções; o índice da função objetivo, os limites superior e inferior das restrições e o ponto inicial.

O LSGRG2 possui características para trabalhar da melhor forma com problemas de otimização de grande porte. Muitas destas características reduzem o tempo de processamento sendo possível resolver problemas com milhares de variáveis e restrições. Uma importante característica é a habilidade de representar eficientemente a matriz de derivadas parciais das restrições e da função objetivo, sendo utilizado uma representação esparsa. É comum para problemas de grande porte existir uma matriz de derivadas parciais esparsa , isto ocorre quando uma restrição utiliza poucas variáveis. Uma grande parte do tempo de processamento é gasto na computação das derivadas parciais. O usuário pode fornecer funções analíticas para o cálculo das derivadas parciais e poupar tempo de processamento.

# 4. Controle ótimo

## 4.1. Introdução

O seguinte problema de controle ótimo corresponde a um problema não linear com restrições de igualdade e desigualdade (variáveis canalizadas). Para a resolução numérica do mesmo, destaca-se o algoritmo GRECO (Gradiente Reduzido para Controle Ótimo) proposto por [9] e [10].

Maximizar $L(x_0, x_1,..., x_T; u_0, u_1,..., u_T)$

$$x_{t+1}=f_t(x_t , u_t)$$
$$\alpha_t \leq u_t \leq \beta_t \ , t=0,\ldots,T-1$$
$$a_t \leq x_t \leq b_t \ , t=0,\ldots,T$$

onde $\forall\, t$ , $x_t \in R^m$ ,vetor coluna das variáveis de estado

$u_t \in R^n$ ,vetor coluna das variáveis de controle

$L \ e \ f_t$ ,funções não lineares de classe $C^1$

$T$ ,horizonte de planejamento

## 4.2. Algoritmo GRECO (Gradiente Reduzido para Controle Ótimo)

Princípios para formar uma base

($P_1$):$\forall\, t$ , introduzir na base o máximo de variáveis de estado.

($P_2$):Se uma variável de estado estiver saturada (não folgada) procurar um substituto entre as variáveis de controle do mesmo período.

($P_3$):Se não for possível utilizar $P_2$ , manter a variável de estado na base de trabalho e introduzir na base GRG (atendendo as condições de degenerescência) uma variável de controle anterior e construir a matriz elementar de pivoteamento.

($P_4$):Fazer a decomposição LU de cada bloco diagonal.

Algoritmo

Passo 1 - $\forall$ *t* obter os conjuntos B e N de variáveis básicas e independentes aplicando os princípios $P_1$, $P_2$, $P_3$ e $P_4$ .A base GRG é fatorada da seguinte forma :

$B = \Lambda E_1 E_2 \ldots E_k$ onde, $\Lambda$ - base de trabalho

B - base GRG

$E_i$- Eta vetores

Passo 2 - Computar a direção de busca $d_N$ das variáveis independentes :

(i)Computar os multiplicadores de Lagrange pelo procedimento BACKW(*x*,*u*) :

Resolução do sistema linear

$\Pi B = b$ onde $b= -c$ (componentes independentes do gradiente)

(ii)Computar o gradiente reduzido $g^N=(w^N, \xi^N)$

$$w^t = \frac{\partial L}{\partial x_t} - \Pi^{t-1} + \Pi^t \frac{\partial f_t}{\partial x_t}$$

$$w^T = \frac{\partial L}{\partial x_t} - \Pi^{T-1}$$

$t$=1, ... , $T$-1

$$\xi^t = \frac{\partial L}{\partial u_t} + \Pi^t \frac{\partial f_t}{\partial u_t}$$

(iii) Computar o gradiente projetado

(a)$P_N=g_N$ ou 0 se uma variável independente estiver saturada e a componente do $g_N$ tirá-la do intervalo.
(b)Se $P_N = 0$, pare ; senão aplique o método do gradiente conjugado para obter uma direção $d_N$ melhor.

Passo 3 - Computar a direção de deslocamento das variáveis básicas pelo procedimento FORW(*x*,*u*)

Resolução do sistema linear
$Bd_B = b$ , onde $b = -A^N d_N$

Passo 4 - Computar uma solução viável melhor

(i)busca linear inexata no hiperplano tangente => novo ponto $(x_\theta,u_\theta)$ não viável, em geral.
(ii)Restauração da viabilidade : Resolver um sistema de equações não lineares, mantendo fixas as variáveis independentes por um método pseudo-Newton, ajustar θ se necessário (troca de base) , obter um novo ponto tal que

$L(x_\theta,u_\theta) > L(x,u)$

(iii) Ir para o passo 1 ;

A estrutura escada da matriz jacobiana é mostrada na figura 4.1, onde $H_0$, ... , $H_{T-1}$ ; $K_0$ , ... ,$K_{T-1}$ são da seguinte forma :

$$H_t = \frac{\partial f}{\partial x^t} \qquad K_t = \frac{\partial f}{\partial u^t}$$

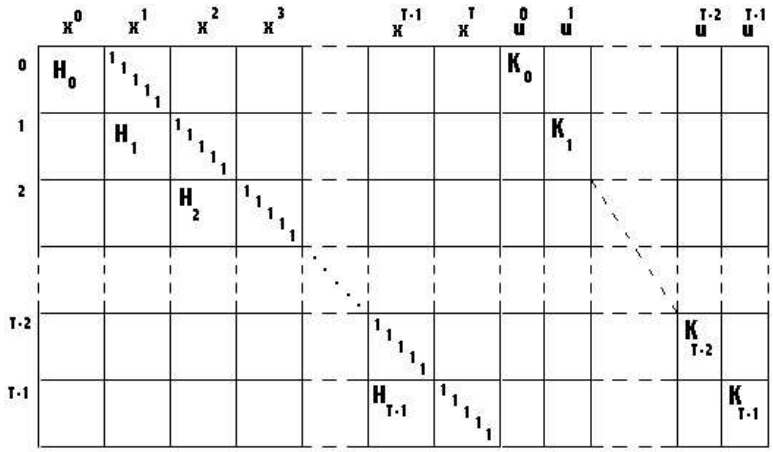


Figura 4.1

# 5. Decomposição LU

A resolução de sistemas lineares no algoritmo GRG, está presente nas seguintes fases :

(i) Computação do gradiente reduzido : $u= -c^B B^{-1}$ ; $g_N = c_N + uA_N$

(ii) Computação da direção de busca das

variáveis básicas : $Bd_B = -A_N d_N$

(iii) Nas iterações de Newton : $B\Delta x = - f(x_B, x_N)$

O objetivo da decomposição LU é fatorar uma matriz A, em um produto de duas matrizes triangulares da seguinte forma A = L.U, onde L é uma matriz triangular inferior e U é uma matriz triangular superior.

Para uma matriz A de ordem n, são necessários n-1 estágios para completar o processo de decomposição. Neste capítulo usaremos a seguinte convenção : s é o indice do estágio de decomposição, I = {1, ..., m} é o conjunto de indices de linhas e J = { 1, ..., m} é o conjunto de indices de coluna, $I^s$ é o conjunto de indices de linhas que não foram pivoteados até o estágio s, $J^s$ é o conjunto de indices de colunas que não foram pivoteados até o estágio s, $A^s$ é a submatriz ativa no estágio s, ou seja, a parte da matriz A que ainda não foi decomposta até o estágio s. Os elementos de A são denotados por $a^s_{i,j}$ .

O algoritmo serial da decomposição LU é dado por:

Para s=1 até n-1 faça

Escolher pivô adequado : ${}^{(s)}a_{s,s}$

Para r=(s+1) até n faça

Computar multiplicador : $m_{r,s} := {}^{(s)}a_{r,s} \ / \ {}^{(s)}a_{s,s}$

${}^{(s)}a_{r,s} := m_{r,s}$

Para c= (s+1) até n faça

$${}^{(s+1)}a_{r,c} := {}^{(s)}a_{r,c} - m_{r,s} * {}^{(s)}a_{s,c} \qquad (5.1)$$

Fim Para

Fim Para

Fim Para

No final da decomposição, os elementos $a^{n}_{i,j}$, tal que $i \leq j$ formam a matriz U e os elementos $a^{n}_{i,j}$, tal que $i > j$ formam a matriz L.

## 5.1. Tratamento da esparsidade

Quando a matriz A é esparsa, novos elementos não nulos podem surgir na eliminação (6.1), quando ${}^{(s)}a_{r,c}$ é nulo, ${}^{(s)}a_{s,c} \neq 0$ e $m_{r,s} \neq 0$ (*fill-in*). A escolha do elemento pivô é fundamental para a minimização do *fill-in*, ou seja para cada estágio *s* deve-se escolher um pivô ${}^{(s)}a_{s,s}$ tal que assegure a estabilidade numérica mas que também minimize o *fill-in*.

O pivoteamento parcial garante a estabilidade numérica, porém não é apropriado para matrizes esparsas, poís o *fill-in* é ignorado. Deve existir liberdade na escolha do pivô sem comprometer a estabilidade numérica, assim dado uma linha i da submatriz ativa $A^s$ um elemento $a^s_{i,j}$ é candidato a pivô se satisfaz o seguinte critério :

$$| a^s_{i,j} | \geq u * \max \{ | a^s_{i,k} | \text{ para todo } k \in J^s \} \qquad (5.2)$$

onde u é um número real fixo tal que $0 < u \leq 1$. O pivoteamento parcial ocorre quando u = 1.

Para minimizar o *fill-in* é aplicado o critério de Markowitz [19] :

- seja $r_i$ o número de elementos não nulos na linha i e $c_j$ o número de elementos não nulos na coluna j então o elemento pivô escolhido é o que minimiza a expressão $(r_i - 1)*( c_j - 1)$

Apesar de antigo, o critério de Markowitz [19] é largamente aplicado para o caso geral de matrizes esparsas, devido a sua simplicidade e facilidade de implementação.

## 5.2. Escolha do pivô

Para a escolha do pivô foi adotado o algoritmo Find_Pivot proposto por Suhl [29], onde ocorre a busca por até p colunas ou linhas utilizando o critério de Markowitz ou por mais de p colunas/linhas, se nenhum pivô foi escolhido anteriormente.

A seguinte notação é usada :

$C^s_k = \{ j \mid j \in J^s$ e a coluna j de $A^s$ possui k elementos não nulos $\}$

$R^s_k = \{ i \mid i \in I^s$ e a linha i de $A^s$ possui k elementos não nulos $\}$

M(j, s) é definido como o mínimo de Markowitz sobre todos os elementos não nulos $a^s_{i,j}$ , $i \in I^s$ que satisfazem o critério de estabilidade (5.2) se a coluna j não contem elemento então definimos $M(j, s) = n^2$. Analogamente é definido M(i,s) como o mínimo de Markowitz sobre todos os elementos não nulos $a^s_{i,j}$ , $j \in J^s$ que satisfazem (5.2)

### 5.2.1. Algoritmo Find_Pivot

Se $C_1^s$ é não vazio então
    Pegue uma coluna de $C_1^s$ como pivô e retorne
Fim Se
Se $R_1^s$ é não vazio então
    Pegue uma linha de $R_1^s$ como pivô e retorne
Fim Se
MM:=$n^2$ ; n0:= 0;
Para k:=2 até n faça
Se $C_k^s$ é não vazio então
    Para todo j $\in$ $C_k^s$ faça
        Se M(j,s) < MM então
            MM:= M(j,s) e seja i a linha correspondente, grave (i,j);
            Se MM $\leq$ $(k-1)^2$ então retorne
        Fim Se
        n0 := n0 +1;
        Se n $\geq$ p e MM < $m^2$ então retorne
    Fim para
Fim Se
Se $R_k^s$ é não vazio então
    Para todo j $\in$ $R_k^s$ faça
        Se M(j,s) < MM então
            MM:= M(i,s) e seja j a coluna correspondente, grave (i,j);
            Se MM $\leq$ k(k-1) então retorne
            Fim Se
        n0 := n0 +1;
        Se n $\geq$ p e MM < $m^2$ então retorne
    Fim para
Fim Se

## 5.3. Versão paralela da decomposição LU

Segundo Duff [7], para o caso de matrizes não simétricas onde a estabilidade numérica não está garantida, não é trivial determinar qual parte da matriz será atualizada em cada estágio da decomposição tornando os algoritmos complexos e difícieis de implementar eficientemente em máquinas de arquiteturas paralelas.

Na arquitetura MIMD, a distribuição dos dados é de extrema importância na exploração do paralelismo. Adotamos a distribuição na qual cada processador escravo é dono de um conjunto de linhas e fica responsável pela eliminação (5.1) das mesmas. O processador mestre além de ser dono de um conjunto de linhas também é responsável pela escolha do pivô, ou seja, recebe informações dos processadores escravos sobre suas linhas e utilizando o algoritmo Find_Pivot escolhe um novo pivô.

Versão paralela da decomposição LU

Cada processador executa

Para s=1 até n-1 faça

SE não ocorreu distribuição ENTÃO processador mestre realiza distribuição

SE processador escravo não possui linha então recebe linhas do processador mestre

O processador mestre, escolhe pivô adequado: $^{(s)}a_{s,s}$

O processador mestre, envia linha pivô para cada processador escravo

Os processadores escravos recebem a linha pivô

Para r=(s+1) até n faça

SE o processador é dono da linha r ENTÃO

Computar multiplicadores : $m_{r,s} := {}^{(s)}a_{r,s} \ / \ {}^{(s)}a_{s,s}$

$^{(s)}a_{r,s} := m_{r,s}$

Para c= (s+1) até n faça

$$^{(s+1)}a_{r,c} := {}^{(s)}a_{r,c} - m_{r,s} * {}^{(s)}a_{s,c} \qquad (5.1)$$

Fim Para

Fim Se

Fim Para

SE processador escravo é dono de alguma linha ENTÃO

Envia linhas para processador mestre

SE foi enviado linha para o processador mestre ENTÃO

Processador mestre recebe linha dos processadores escravos

Fim Para

# 6. Gradiente-Conjugado

Os algoritmos iterativos em geral, fazem uma aproximação inicial da solução do problema e a cada nova iteração, realizam uma nova aproximação conforme as regras do método. As iterações chegam ao fim quando o critério de parada é satisfeito.

Os métodos iterativos podem ser estacionários ou não. Os estacionários manipulam componente a componente do sistema durante a resolução. Os não es tacionários trabalham sob a ótica da minimização da função quadrática ou por projeção e incluem hereditariedade em suas iterações. O Gradiente Conjugado é um exemplo de método iterativo não estacionário e é considerado como um dos métodos iterativos mais eficientes para resolução de sistemas lineares de grande porte e esparsos.

Neste trabalho o gradiente conjugado será utilizado para a computação da direção de busca do método GRG (Gradiente Reduzido Generalizado). O Gradiente Conjugado parte do princípio de que o gradiente, que é um campo vetorial, aponta sempre na direção mais crescente da função quadrática. O gradiente da função quadrática, se a matriz é simétrica e definida positiva, se resume à solução de um sistema de equações lineares.

O gradiente aponta na direção de máximo crescimento da função. Geometricamente, isso significa que na base da parabolóide formada pela função quadrática, o gradiente é zero, que é a solução do sistema linear.

De forma simples, a idéia do método do Gradiente Conjugado é ir dando passos, em cada iteração, no sentido oposto ao do gradiente, de tal forma que a direção já pesquisada não seja repetida. Para tanto, é gerado um subespaço de pesquisa a partir dos resíduos de cada iteração.

O método do Gradiente Conjugado[18] para funções quadráticas :

$f(x) = \frac{1}{2} x^t Ax - b^t x$

$\nabla f(x) = Ax - b$

é dado por :

- Passo na direção de pesquisa.
- Razão entre as normas do resíduo, usado para calcular a nova direção de pesquisa.
- Resíduo *r*, calculado da seguinte forma $r_i = b - Ax_i$
- Aproximação do valor de $x^i$, isto é, $x^i = x_{i-1} + a_i p_i$
- Cálculo da nova direção de pesquisa ; $p_i = z_{i-1} + b_{i-1} p_{i-1}$

O algoritmo do método do Gradiente Conjugado é apresentado a seguir:

Dado $x_o$ inicial
Seja $d_o = - g_o = b - Ax_0 = -\nabla f(x_0)$
para k = 1, 2, 3 ...

$g_k = Ax_k - b$
$\alpha_k = - (g^t_k d_k) / (d^t_k A d_k)$
$x_{k+1} = x_k + \alpha_k d_k$
$\beta_k = (g^t_{k+1} A d_k) / (d^t_k A d_k)$
$d_{k+1} = -g_{k+1} + \beta_k d_k$

Verifique convergência e continue se necessário
fim

**Para funções não lineares gerais**

Fazemos a seguinte associação para o ponto $x_k$

$g_k <-> \nabla f(x_k)^t$ , $Q <-> H(x_k)$

Dado $x_o$ inicial
Compute $g_0 = \nabla f(x_0)^t$ e seja $d_o = - g_o$
Para k = 0,1, .... , n-1

$\alpha_k = - (g^t_k d_k) / (d^t_k H(x_k) dk)$

$x_{k+1} = x_k + \alpha_k d_k$

$g_{k+1} = \nabla f(x_{k+1})^t$
Se k < n-1 então
$\beta_k = (g^t_{k+1} H(x_k) d_k ) / (d^t_k H(x_k) d_k)$
$d_{k+1} = -g_{k+1} + \beta_k d_k$
Fim Se
Fim Para

Uma vantagem do algoritmo é que nenhuma busca linear é necessária em qualquer estágio. O algoritmo converge em um número finito de passos para um problema quadrático. Uma desvantagem é que $H(x_k)$ deve ser avaliada a cada ponto, sendo freqüentemente impraticável, desta forma o algoritmo não apresenta convergência global.

**Métodos de Fletcher-Reeves e Polak-Ribiere**

Para evitar o esforço computacional de lidar com a matriz hessiana, deve-se fazer uma busca linear inexata para determinar o passo $\alpha_k$ (Al-Baali), e trocar-se a fórmula do passo $\beta_k$ como se segue,

Método de Fletcher-Reeves

Dado $x_o$ inicial
Compute $g_0 = \nabla f(x_0)^t$ e seja $d_o = - g_o$
Para $k = 0,1, .... , n-1$

$x_{k+1} = x_k + \alpha_k d_k$ onde $\alpha_k$ minimiza $f(x_k + \alpha d_k)$

Compute $g_{k+1} = \nabla f(x_{k+1})^t$
Se $k < n-1$ então
$\beta_k = (g^t_{k+1}\, g_{k+1}) / (g^t_k\, g_k)$
$d_{k+1} = -g_{k+1} + \beta_k d_k$
Fim Se
Fim Para

O método Polak-Ribiere só difere na fórmula do parâmetro :

$$\beta_k = ( (g_{k+1} - g_k )^t\, g_{k+1} ) / (g^t_k\, g_k)$$

No caso de funções quadráticas ambos os métodos levam à obtenção de um passo $\alpha_k$ idêntico a da formula original. Comparações experimentais indicam que Polak-Ribiere apresenta uma convergência mais rápida que os demais métodos desta classe, havendo porém um risco de não convergência em alguns casos.

**Paralelização**

A paralelização do método é feita a partir das operações básicas de álgebra linear. Após o particionamento dos dados, cada processador terá uma porção dos dados iniciais. Assim , a maior parte das operações entre vetores são executadas, em cada nó, como na forma seqüêncial, não exigindo comunicação e proporcionando ganho de performance.

No produto escalar de dois vetores há necessidade de comunicação. Cada processador calcula a sua parte do produto escalar e após, esse resultado é acumulado no processador mestre.

# 7. Resultados Numéricos

## 7.1. Decomposição LU

Uma primeira implementação da decomposição LU de matrizes densas foi feita utilizando linguagem C, e a biblioteca MPI [28], disponível no computador paralelo IBM-SP2, para troca de mensagens.

Foram realizados testes com 1, 2 e 4 processadores com bloco 0, 1 e 2 , sendo utilizadas matrizes densas de ordem 100 até 3000 geradas aleatóriamente. O tempo de processamento refere-se apenas ao tempo necessário para decomposição LU e para o caso de 2 e 4 processadores inclui-se tambem o tempo gasto com troca de mensagens. Os dados experimentais encontram-se na tabela 7.1.

A interpretação dos gráfico 7.1 e 7.2, demonstra inicialmente que apenas o aumento do número de processadores não é fator único na melhora do tempo de processamento, mas também o aumento do número de linhas em cada bloco, ou seja, na melhora da distribuição de trabalho entre processadores. Isto ocorre por que diminui-se o número de mensagens trocadas.

A análise das tabelas 7.2 e 7.3, mostra que independentemente da dimensão do problema o *speedup* e a eficiência permanecem estáveis para o mesmo bloco, consequentemente o ganho em performane é aproximadamente o mesmo para uma configuração com $p$ processadores e bloco $i$ na decomposição de uma matriz de dimensão $n$.

O número total de mensagens trocadas contribui no tempo de processamento. Dados numéricos comprovam que a utilização de bloco 0, ao minimizar o número de mensagens trocadas contribui para o ganho de performance.

| | 1 Processador | 2 Processadores | | | 4 Processadores | | |
|---|---|---|---|---|---|---|---|
| Matriz | Tempo(Seg) | Tempo(seg) Bloco 1 | Tempo(seg) Bloco 2 | Tempo(seg) Bloco 0 | Tempo(seg) Bloco 1 | Tempo(seg) Bloco 2 | Tempo(seg) Bloco 0 |
| 100 | 0,13 | 0,22 | 0,19 | 0,21 | 0,29 | 0,29 | 0,31 |
| 200 | 0,97 | 0,79 | 0,83 | 0,60 | 0,81 | 0,74 | 0,73 |
| 300 | 3,27 | 2,09 | 2,08 | 1,66 | 1,82 | 1,72 | 1,11 |
| 400 | 7,66 | 4,59 | 4,47 | 3,11 | 3,10 | 3,17 | 2,18 |
| 500 | 14,94 | 8,16 | 8,29 | 5,96 | 5,35 | 5,13 | 3,14 |
| 600 | 25,73 | 14,31 | 14,24 | 9,97 | 8,70 | 8,72 | 5,05 |
| 700 | 40,81 | 22,29 | 22,19 | 15,24 | 13,02 | 13,04 | 7,80 |
| 800 | 60,83 | 32,62 | 32,39 | 22,39 | 18,63 | 18,42 | 9,93 |
| 900 | 86,62 | 45,28 | 45,74 | 31,05 | 25,78 | 25,61 | 13,57 |
| 1000 | 118,79 | 62,19 | 62,09 | 42,83 | 34,37 | 34,18 | 17,48 |
| 1100 | 157,69 | 81,86 | 82,32 | 56,22 | 44,73 | 44,83 | 21,75 |
| 1200 | 204,87 | 105,81 | 106,03 | 73,39 | 57,41 | 57,43 | 27,70 |
| 1300 | 260,17 | 134,29 | 133,64 | 91,99 | 71,76 | 71,97 | 33,97 |
| 1400 | 324,62 | 167,46 | 167,14 | 117,78 | 89,49 | 88,67 | 40,75 |
| 1500 | 399,44 | 204,18 | 204,47 | 141,93 | 108,35 | 108,60 | 51,64 |
| 1600 | 483,47 | 248,76 | 247,90 | 170,82 | 130,43 | 127,92 | 63,00 |
| 1700 | 581,18 | 295,89 | 295,95 | 203,47 | 155,30 | 153,75 | 77,97 |
| 1800 | 688,42 | 351,95 | 351,37 | 243,45 | 183,38 | 183,16 | 84,18 |
| 1900 | 810,61 | 412,61 | 412,41 | 280,17 | 213,91 | 216,05 | 96,54 |
| 2000 | 939,53 | 481,23 | 480,84 | 335,78 | 249,56 | 248,42 | 114,29 |
| 2100 | 1.094,32 | 554,53 | 556,64 | 378,67 | 289,17 | 289,17 | 129,17 |
| 2200 | 1.257,00 | 637,51 | 639,38 | 438,01 | 330,41 | 329,17 | 150,36 |
| 2300 | 1.431,97 | 727,54 | 729,70 | 492,23 | 376,35 | 373,06 | 170,09 |
| 2400 | 1.631,55 | 828,83 | 827,53 | 576,44 | 426,51 | 426,88 | 197,75 |
| 2500 | 1.822,01 | 934,61 | 935,09 | 626,97 | 480,48 | 481,76 | 217,80 |
| 2600 | 2.045,16 | 1.048,51 | 1.051,56 | 736,66 | 539,31 | 540,02 | 232,33 |
| 2700 | 2.256,18 | 1.176,04 | 1.174,48 | 814,96 | 603,63 | 604,00 | 268,82 |
| 2800 | 2.516,04 | 1.309,01 | 1.311,57 | 918,86 | 673,58 | 663,38 | 303,73 |
| 2900 | 2.806,00 | 1.457,45 | 1.456,11 | 966,81 | 746,73 | 748,03 | 346,25 |
| 3000 | 3.118,25 | 1.613,90 | 1.611,95 | 1.078,44 | 825,72 | 824,80 | 348,70 |

Tabela 7.1 – Tempo de processamento para decomposição LU

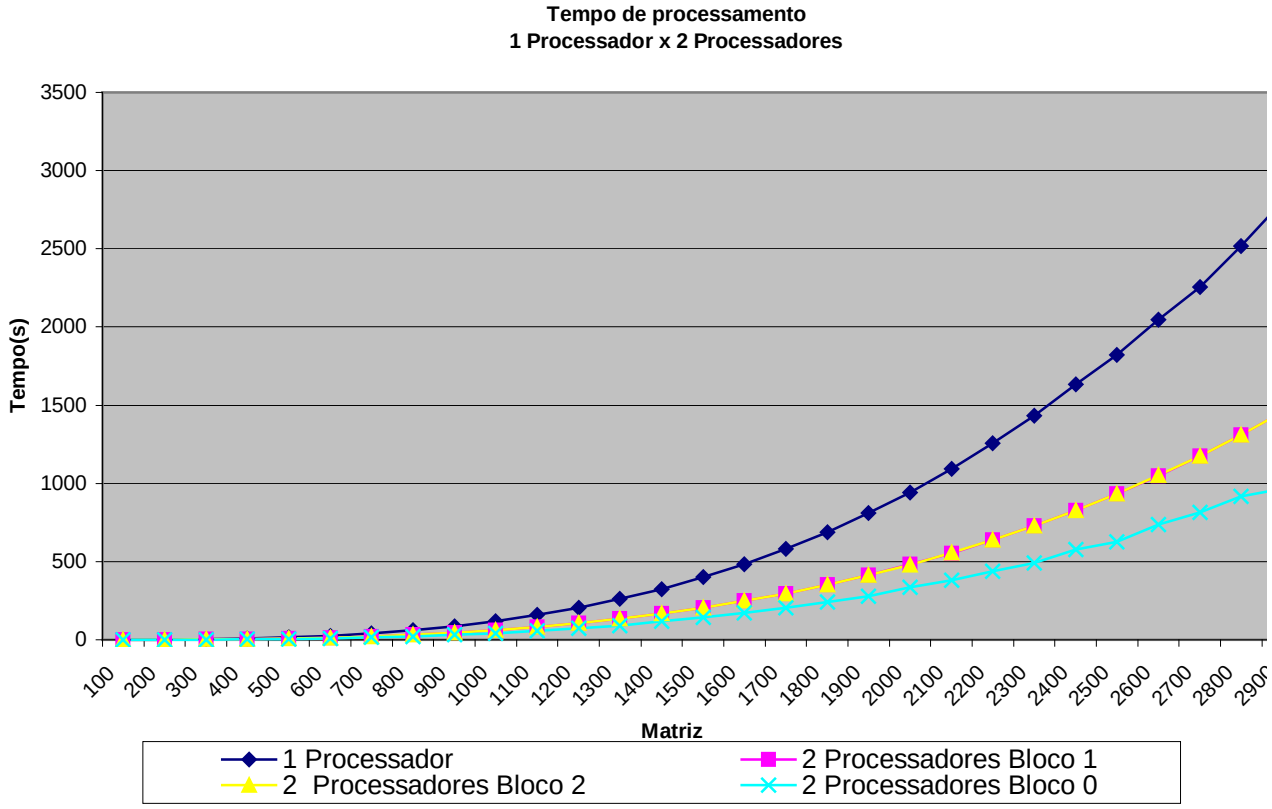
Tempo de processamento
1 Processador x 2 Processadores
Tempo(s)
3500
3000
2500
2000
1500
1000
500
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
2000
2100
2200
2300
2400
2500
2600
2700
2800
Matriz
1 Processador
2 Processadores Bloco 1
2 Processadores Bloco 2
2 Processadores Bloco 0

Gráfico 7.1

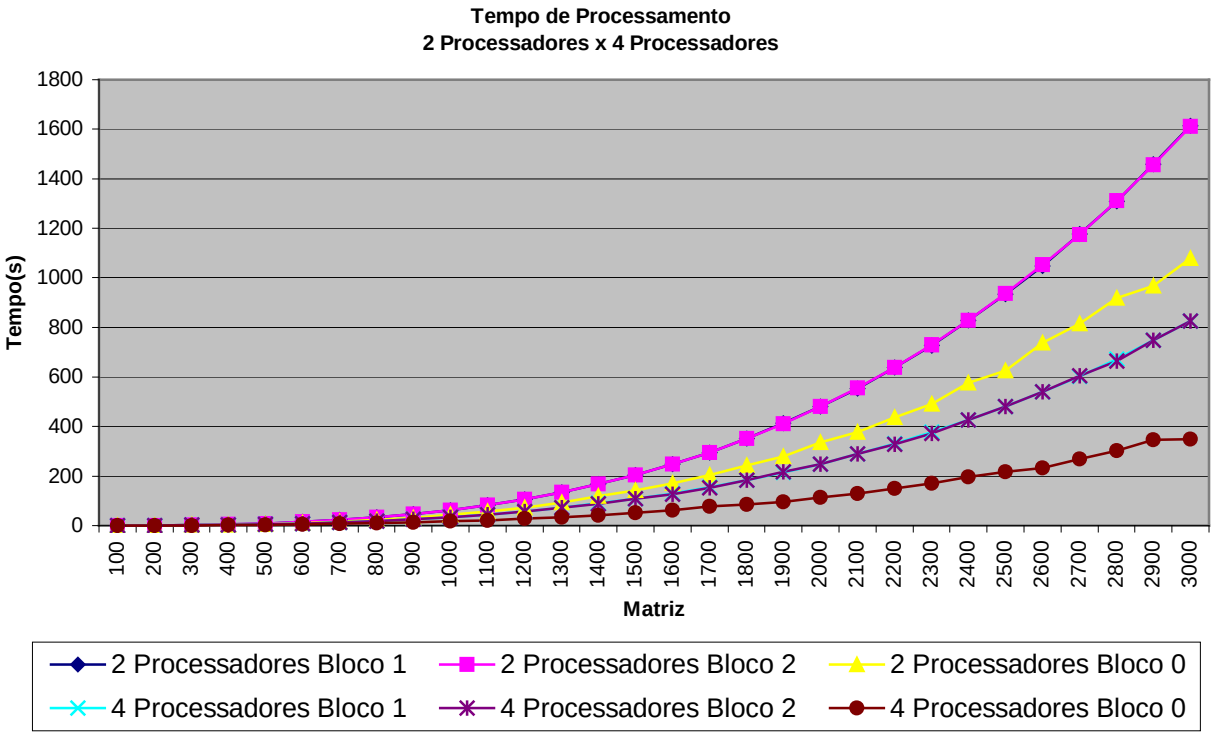
Tempo de Processamento
2 Processadores x 4 Processadores
Tempo(s)
1800
1600
1400
1200
1000
800
600
400
200
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
2000
2100
2200
2300
2400
2500
2600
2700
2800
2900
3000
Matriz
2 Processadores Bloco 1
2 Processadores Bloco 2
2 Processadores Bloco 0
4 Processadores Bloco 1
4 Processadores Bloco 2
4 Processadores Bloco 0

Gráfico 7.2

| | 2 Processadores | | | | | |
|---|---|---|---|---|---|---|
| Matriz | Speedup Bloco 1 | Speedup Bloco 2 | Speedup Bloco 0 | Eficiência Bloco 1 | Eficiência Bloco 2 | Eficiência Bloco 0 |
| 100 | 0,5909 | 0,6842 | 0,6190 | 0,2955 | 0,3421 | 0,3095 |
| 200 | 1,2278 | 1,1687 | 1,6167 | 0,6139 | 0,5843 | 0,8083 |
| 300 | 1,5646 | 1,5721 | 1,9699 | 0,7823 | 0,7861 | 0,9849 |
| 400 | 1,6688 | 1,7136 | 2,4630 | 0,8344 | 0,8568 | 1,2315 |
| 500 | 1,8309 | 1,8022 | 2,5067 | 0,9154 | 0,9011 | 1,2534 |
| 600 | 1,7980 | 1,8069 | 2,5807 | 0,8990 | 0,9034 | 1,2904 |
| 700 | 1,8309 | 1,8391 | 2,6778 | 0,9154 | 0,9196 | 1,3389 |
| 800 | 1,8648 | 1,8780 | 2,7168 | 0,9324 | 0,9390 | 1,3584 |
| 900 | 1,9130 | 1,8937 | 2,7897 | 0,9565 | 0,9469 | 1,3948 |
| 1000 | 1,9101 | 1,9132 | 2,7735 | 0,9551 | 0,9566 | 1,3868 |
| 1100 | 1,9263 | 1,9156 | 2,8049 | 0,9632 | 0,9578 | 1,4024 |
| 1200 | 1,9362 | 1,9322 | 2,7915 | 0,9681 | 0,9661 | 1,3958 |
| 1300 | 1,9374 | 1,9468 | 2,8282 | 0,9687 | 0,9734 | 1,4141 |
| 1400 | 1,9385 | 1,9422 | 2,7562 | 0,9692 | 0,9711 | 1,3781 |
| 1500 | 1,9563 | 1,9535 | 2,8143 | 0,9782 | 0,9768 | 1,4072 |
| 1600 | 1,9435 | 1,9503 | 2,8303 | 0,9718 | 0,9751 | 1,4151 |
| 1700 | 1,9642 | 1,9638 | 2,8563 | 0,9821 | 0,9819 | 1,4282 |
| 1800 | 1,9560 | 1,9592 | 2,8278 | 0,9780 | 0,9796 | 1,4139 |
| 1900 | 1,9646 | 1,9655 | 2,8933 | 0,9823 | 0,9828 | 1,4466 |
| 2000 | 1,9524 | 1,9539 | 2,7981 | 0,9762 | 0,9770 | 1,3990 |
| 2100 | 1,9734 | 1,9659 | 2,8899 | 0,9867 | 0,9830 | 1,4450 |
| 2200 | 1,9717 | 1,9660 | 2,8698 | 0,9859 | 0,9830 | 1,4349 |
| 2300 | 1,9682 | 1,9624 | 2,9091 | 0,9841 | 0,9812 | 1,4546 |
| 2400 | 1,9685 | 1,9716 | 2,8304 | 0,9842 | 0,9858 | 1,4152 |
| 2500 | 1,9495 | 1,9485 | 2,9061 | 0,9747 | 0,9742 | 1,4530 |
| 2600 | 1,9505 | 1,9449 | 2,7763 | 0,9753 | 0,9724 | 1,3881 |
| 2700 | 1,9185 | 1,9210 | 2,7685 | 0,9592 | 0,9605 | 1,3842 |
| 2800 | 1,9221 | 1,9183 | 2,7382 | 0,9610 | 0,9592 | 1,3691 |
| 2900 | 1,9253 | 1,9271 | 2,9023 | 0,9626 | 0,9635 | 1,4512 |
| 3000 | 1,9321 | 1,9345 | 2,8914 | 0,9661 | 0,9672 | 1,4457 |

Tabela 7.2 – Speedup e Eficiência para decomposição LU com dois processadores

| | 4 Processadores | | | | | |
|---|---|---|---|---|---|---|
| Matriz | Speedup Bloco 1 | Speedup Bloco 2 | Speedup Bloco 0 | Eficiência Bloco 1 | Eficiência Bloco 2 | Eficiência Bloco 0 |
| 100 | 0,4483 | 0,4483 | 0,4194 | 0,1121 | 0,1121 | 0,1048 |
| 200 | 1,1975 | 1,3108 | 1,3288 | 0,2994 | 0,3277 | 0,3322 |
| 300 | 1,7967 | 1,9012 | 2,9459 | 0,4492 | 0,4753 | 0,7365 |
| 400 | 2,4710 | 2,4164 | 3,5138 | 0,6177 | 0,6041 | 0,8784 |
| 500 | 2,7925 | 2,9123 | 4,7580 | 0,6981 | 0,7281 | 1,1895 |
| 600 | 2,9575 | 2,9507 | 5,0950 | 0,7394 | 0,7377 | 1,2738 |
| 700 | 3,1344 | 3,1296 | 5,2321 | 0,7836 | 0,7824 | 1,3080 |
| 800 | 3,2652 | 3,3024 | 6,1259 | 0,8163 | 0,8256 | 1,5315 |
| 900 | 3,3600 | 3,3823 | 6,3832 | 0,8400 | 0,8456 | 1,5958 |
| 1000 | 3,4562 | 3,4754 | 6,7958 | 0,8641 | 0,8689 | 1,6989 |
| 1100 | 3,5254 | 3,5175 | 7,2501 | 0,8813 | 0,8794 | 1,8125 |
| 1200 | 3,5685 | 3,5673 | 7,3960 | 0,8921 | 0,8918 | 1,8490 |
| 1300 | 3,6256 | 3,6150 | 7,6588 | 0,9064 | 0,9037 | 1,9147 |
| 1400 | 3,6274 | 3,6610 | 7,9661 | 0,9069 | 0,9152 | 1,9915 |
| 1500 | 3,6866 | 3,6781 | 7,7351 | 0,9216 | 0,9195 | 1,9338 |
| 1600 | 3,7067 | 3,7795 | 7,6741 | 0,9267 | 0,9449 | 1,9185 |
| 1700 | 3,7423 | 3,7800 | 7,4539 | 0,9356 | 0,9450 | 1,8635 |
| 1800 | 3,7541 | 3,7586 | 8,1780 | 0,9385 | 0,9396 | 2,0445 |
| 1900 | 3,7895 | 3,7520 | 8,3966 | 0,9474 | 0,9380 | 2,0992 |
| 2000 | 3,7647 | 3,7820 | 8,2206 | 0,9412 | 0,9455 | 2,0551 |
| 2100 | 3,7843 | 3,7843 | 8,4719 | 0,9461 | 0,9461 | 2,1180 |
| 2200 | 3,8044 | 3,8187 | 8,3599 | 0,9511 | 0,9547 | 2,0900 |
| 2300 | 3,8049 | 3,8384 | 8,4189 | 0,9512 | 0,9596 | 2,1047 |
| 2400 | 3,8253 | 3,8220 | 8,2506 | 0,9563 | 0,9555 | 2,0626 |
| 2500 | 3,7921 | 3,7820 | 8,3655 | 0,9480 | 0,9455 | 2,0914 |
| 2600 | 3,7922 | 3,7872 | 8,8028 | 0,9480 | 0,9468 | 2,2007 |
| 2700 | 3,7377 | 3,7354 | 8,3929 | 0,9344 | 0,9339 | 2,0982 |
| 2800 | 3,7353 | 3,7928 | 8,2838 | 0,9338 | 0,9482 | 2,0710 |
| 2900 | 3,7577 | 3,7512 | 8,1040 | 0,9394 | 0,9378 | 2,0260 |
| 3000 | 3,7764 | 3,7806 | 8,9425 | 0,9441 | 0,9452 | 2,2356 |

Tabela 7.3 – Speedup e Eficiência para decomposição LU com quatro processadores

## 7.2. Problemas não lineares

Utilizamos o eficiente código computacional LSGRG2 de Leon S. Lasdon (Universidade do Texas em Austin) [17], onde substituímos a rotina de inversão da base por um LU-paralelo e a direção de busca por um método de gradiente-conjugado paralelo (Fletcher-Reeves/Polak-Ribière). Os resultados numéricos obtidos no computador IBM-SP2 do NCE/UFRJ são apresentados com diferentes problemas-testes da coleção CUTEr [24] e com o problema clássico seguinte:

Problema das Armas

$x_{ij}$ = Número de armas do tipo i associadas ao alvo j

i = 1, ..., p (número de armas) e j = 1, ... , q (número de alvos)

$a_i$ = total de armas do tipo i

$b_j$ = número mínimo de armas de todos os tipos associados ao alvo j

$\alpha_{ij}$ = probabilidade do alvo j não ser danificado por um ataque de uma unidade da arma i

$u_j$ = valor militar da destruição do alvo j

Assim temos o modelo de programação não linear com restrições:

Max $\sum_{j=1}^{q} u_j [1 - \prod_{i=1}^{p} \alpha_{ij}^{x_{ij}}]$

sujeito a

$\sum_{j=1}^{q} x_{ij} \leq a_i$ para i=1, ..., p

$\sum_{i=1}^{p} x_{ij} \geq b_i$ para j=1, ..., q

$x_{ij} \geq 0$ para i=1, ..., p e j=1, ..., q

Resultados numéricos do problema das Armas e de problemas da coleção CUTEr [24]

| | Número Variáveis | Variáveis limitadas | Restrições Igualdade | Restrições Desigualdade | Iterações | Fobj(x*) | lsgrg2 Tempo | p1 Tempo | p2 Tempo | p3 Tempo | p4 Tempo |
|---|---|---|---|---|---|---|---|---|---|---|---|
| ARMAS | 100 | SIM | 0 | 12 | 3 | -1.76E+03 | 0.04 | 0.05 | 0.06 | 0.05 | 0.05 |
| LAUNCH | 25 | SIM | 9 | 19 | 13 | 9.005069 | 0.31 | 0.31 | 0.31 | 0.30 | 0.31 |
| CLNLBEAM | 33 | SIM | 20 | 0 | 1 | 350 | 0.07 | 0.05 | 0.06 | 0.05 | 0.05 |
| DITTERT | 61 | SIM | 37 | 0 | 1 | -1 | 0.21 | 0.26 | 0.25 | 0.26 | 0.27 |
| DALLASL | 906 | SIM | 667 | 0 | 272 | -2.58E+08 | 262.57 | 745.75 | 740.25 | 739.20 | 740.10 |
| READING7 | 1002 | SIM | 500 | 0 | 1 | 0 | 169.20 | 169.41 | 169.50 | 169.42 | 169.43 |
| CHAIN | 802 | NÃO | 401 | 0 | 18 | 5.137 | 49.49 | 84.12 | 83.98 | 83.99 | 83.87 |
| READING1 | 4002 | SIM | 2000 | 0 | 3 | 0 | 909.13 | 912.97 | 910.54 | 910.47 | 910.49 |

| | Speedup | | | | Eficiência | | | |
|---|---|---|---|---|---|---|---|---|
| | p1 | p2 | p3 | p4 | p1 | p2 | p3 | p4 |
| ARMAS | 0.8000 | 0.6667 | 0.8000 | 0.8000 | 0.8000 | 0.3333 | 0.2667 | 0.2000 |
| LAUNCH | 1.0000 | 1.0000 | 1.0333 | 1.0000 | 1.0000 | 0.5000 | 0.3444 | 0.2500 |
| CLNLBEAM | 1.4000 | 1.1667 | 1.4000 | 1.4000 | 1.4000 | 0.5833 | 0.4667 | 0.3500 |
| DITTERT | 0.8077 | 0.8400 | 0.8077 | 0.7778 | 0.8077 | 0.4200 | 0.2692 | 0.1944 |
| DALLASL | 0.3521 | 0.3547 | 0.3552 | 0.3548 | 0.3521 | 0.1774 | 0.1184 | 0.0887 |
| READING7 | 0.9988 | 0.9982 | 0.9987 | 0.9986 | 0.9988 | 0.4991 | 0.3329 | 0.2497 |
| CHAIN | 0.5883 | 0.5893 | 0.5892 | 0.5901 | 0.5883 | 0.2947 | 0.1964 | 0.1475 |
| READING1 | 0.9958 | 0.9985 | 0.9985 | 0.9985 | 0.9958 | 0.4992 | 0.3328 | 0.2496 |

# 8. Conclusão

O desejo de relacionar duas áreas de conhecimento distintas motivou o presente estudo das técnicas de paralelismo e de otimização não linear. Procurou-se analisar as vantagens no uso do paralelismo com o método GRG, especificamente na decomposição LU e na computação da direção de busca utilizando o método do gradiente-conjugado.

Segundo Duff [7,8] um fator de importância na performance das máquinas de arquitetura MIMD é o número de mensagens trocadas entre processadores pois o tempo de envio e recebimento de mensagens concorre com o tempo de processamento.

Para verificar os benefícios do paralelismo na decomposição LU é necessário observar os seguintes passos do algoritmo : escolha do pivô , minimização do *fill-in* e distribuição da matriz entre os processadores.

Para o caso geral de otimização não linear de grande porte encontramos matrizes jacobianas esparsas[24], implicando matrizes de base não necessariamente simétricas, o que não garante a estabilidade numérica. Ao contrário das matrizes simétricas definidas positivas, a escolha do pivô deve ser feita a cada estágio da decomposição, em conjunto com a eliminação de cada linha, com o objetivo de minimizar o *fill-in* e garantir a estabilidade numérica [7]. Deste modo, existe a necessidade de maior troca de mensagens entre processadores, o que poderá comprometer a performance geral de processamento.

Na decomposição LU o paralelismo é explorado na eliminação de variáveis, porém em alguns problemas testados foram encontradas matrizes com muitas linhas ou colunas com poucos elementos não nulos, o que trouxe pouco aproveitamento do paralelismo.

No gradiente-conjugado o paralelismo é explorado na computação de produtos internos, indicando que o ganho de performance poderá ocorrer apenas para problemas de grande dimensão.

Em um futuro trabalho podendo utilizar diversos procedimentos aqui sugeridos, no caso de problemas de controle ótimo de sistemas não lineares. O algoritmo GRECO (Gradiente Reduzido para o Controle Ótimo) [9] apresenta uma estratégia com uma hierarquia na composição da matriz de base procurando explorar a estrutura em escada da matriz jacobiana. As técnicas de paralelismo poderão acrescentar mais eficiência.

# 9. Referências Bibliográficas

[1]ABADIE, J. ed**.** (1970) **- Application of the GRG algorithm to optimal control problems**. in : ABADIE, J. **Integer and nonlinear programming**. Amsterdam : North-Holland , p.191-211.

[2]ABADIE, J. & CARPENTIER, J.(1969) - **Generalization of the reduced gradient method to the case of nonlinear constrains**. in Fletcher, R. (ed.) **Optimization**, Academic P.

[3]AMDAHL, G. (1967) – **The validity of the single processor approach to achieving large scale computing capabilities,** in **Proceedings of the AFIPS Computing Conference**, v.30, p. 483-485.

[4]BERTSEKAS, D. P. & TSITSIKLIS, J. N. (1989) - **Parallel and distributed computation : numerical methods.** New Jersey : Prentice-Hall, 715p.

[5]CARRIERO, N. & GELERNTER, D. (1992) - **How to write parallel programs : a first course**. Cambridge , Mass. : Massachusetts Institute of Technology, 232p.

[6]DONGARRA, Jack J. et al ... (1998) - **Numerical linear algebra for high-performance computers**. Philadelphia : SIAM, 342p.

[7]DUFF, I. S. & REID, J. K. (1989) - **Direct methods for sparse matrices.** New York : Oxford University Press, 341p.

[8]DUFF, I. S. (1999) – **The impact of high performance computing in the solution of linear systems: trends and problems**, Technical Report TR/PA/99/41.

[9]FACO, J.L.D. (1977) - **Commande optimale des systèmes dynamiques non linéaires à retards avec contraintes d'inégalités sur l'état et la commande**, tese de Docteur-Ingénieur, Institut de Programmation, Université Pierre et Marie Curie (Paris VI), 102 p.

[10]FACO, J.L.D. (1989) - **A generalized reduced gradient algorithm for solving large-scale discrete-time nonlinear optimal control problems**, in Eigth IFAC WorkShop: Control Applications of Nonlinear Programming and Optimization, June 7-9/89, p. 27-35, Ecole Polytechnique, Paris, França.

[11]FLYNN, M. J. (1966) - **Very high-speed computing systems**. Proceeding of the IEEE, v.54, n.12, p.1901-1909.

[12]FOSTER, I. (1995) - **Design and building parallel programs : concepts and tools for parallel software engineering**. Reading, Mass : Addison-Wesley, 379p.

[13]GALLIVAN, K. A. et al ... (1994) - **Parallel algorithms for matrix computations**. Philadelphia : SIAM, 197p.

[14]GOULD, NICHOLAS I. M. et. al. – **Cuter (and SifDec), a Constrained and Unconstrained Testing Environment, revisited.** Technical Report No. TR/PA/01/04

[15]GROPP, William, LUSK, Ewing & SKJELLUM, Anthony. (1994) - **Using MPI : portable parallel programming with the message-passing interface**. Cambridge : Massachusetts Institute of Technology, 307p.

[16]HERSKOVITS, J (1995), **A View on Nonlinear Optimization** in Herskovits, J., (ed.), **Advances in Structural Optimization,** Kluwer, p. 71-117.

[17]LASDON, Leon S. et al. (1998) - **LSGRG2** [software]. Austin, Tx : Optimal Methods Inc.

[18]LUENBERGER, G. D. (1984), **Linear and nonlinear programming**, Addison-Wesley, 490p.

[19]MARKOWITZ, H. M., (1957). **The Elimination Form of Inverse and Its Applications to Linear Programming**, Management Science 3, p.255-269.

[20]MARKUS, L. , LEE, E. B. (1967) - **Foundations of optimal control theory**. New York : John Wiley & Sons, 575p.

[21]MATEUS, G. R. (1986), **Programação não linear**, Belo Horizonte : UFMG, 289p.

[22]MENABREA, L. F. (1842) - **Sketch of the analytical engine invented by Charles Babbage.** Geneve : Bibliothèque Universelle de Genève.

[23]MOROZOWSKI Filho, Marciano (1981) - **Matrizes esparsas em redes de potência : técnicas de operação.** Rio de Janeiro : LTC, 172p.

[24]NUMERICAL ANALYSIS GROUP. Computational Science & Engineering Department. **The CUTEr Test Problem Set**. Disponível em : http://cuter.rl.ac.uk/cuter-www/Problems/mastsif.shtml

[25]PACHECO, Peter S. (1997) - **Parallel programming with MPI** San Francisco : M. Kaufmann, 418p.

[26]RIBEIRO JR., Maurício Vicente (2000) - **Paralelização de métodos numéricos para solução de sistemas lineares esparsos**. Rio de Janeiro, 136f. Dissertação (mestrado) - UFRJ-NCE-IM.

[27]SCHITTKOWSKI, K. & HOCK, W. (1981) **Test example for nonlinear programming codes.** Springer Verlag, Berlin. Lecture notes in economics and mathematical systems, volume 187

[28]SNIR, Marc ... et al. (1996) - **MPI : the complete reference**. Massachusetts Institute of Technology, 336p.

[29]SUHL, U. H. (1987), **Computing sparse LU-factorizations for large-scale linear programming bases**, Math. Prog., Vol. 24, p.55-69.

[30]TABAK, Daniel & KUO, B. C. (1971) - **Optimal Control by mathematical programming.** New Jersey : Prentice-Hall, 237p.

[31]TEWARSON, R. P. (1973) - **Sparse Matrices.** New York : Academic Press, 160p.

[32]WOLFE, P. (1963) - **Methods of Nonlinear Programming**, in Graves & Wolfe (eds), **Recent Advances in Mathematical Programming**, p. 67-86