Divisão e Conquista:

uma técnica para

paralelização de algoritmos

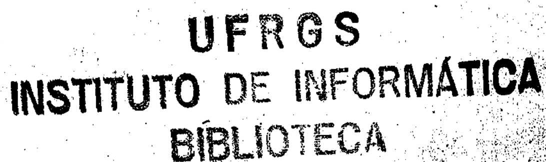
por

Tiaraju Asmuz Diverio
Dalcídio Moraes Claudio
Philippe Olivier Navaux

RP-177 Maio/92

UNIVERSIDADE FEDERAL DO RIO GRANDE SO SUL
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
Av Bento Gonçalvez, 9500 - Agronomia
91501 - Porto Alegre - RS - BRASIL
Telefone: (051) 336-8399 /339-1355 Ramal 6161
Telex: (051) 2680 - CCUF BR
FAX: (051) 336-5576
E-MAIL: PGCC@INF.UFRGS.BR

Correspondência: UFRGS-CPGCC
Caixa Postal 15064
91501 - Porto Alegre - RS - BRASIL


UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

Editor: Ricardo Augusto da Luz Reis (interino)

Matemática computacional - SBC
Algoritmos
Algoritmos paralelos
Processamento paralelo
Divisão; Conquista

CNPq 1.01.04.00-3

U F R G S
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

Nº CHAMADA		Nº REG.:
FL 4781		5474
		DATA: 13/08/92
ORIGEM: D	DATA: 14/05/1992	PREÇO: Cr$ 60.000,00
FUNDO: CPGCC	FORN.: CPGCC	

UFRGS

Reitor: Prof. TUISKON DICK
Pró-Reitor de Pesquisa e Pós-Graduação: Prof. ABÍLIO BAETA NEVES
Coordenador do CPGCC: Prof. Ricardo A. da L. Reis
Comissão Coordenadora do CPGCC: Prof. Cirano Iochpe
Prof. Clesio Saraiva dos Santos
Prof. José Mauro V. de Castilho
Prof. Raul Fernando Weber
Prof. Ricardo A. da L. Reis
Profa. Rosa Maria Viccari

Bibliotecária CPGCC/II: Celina Leite Miranda

Divisão e Conquista:

uma técnica para

paralelização de algoritmos

por

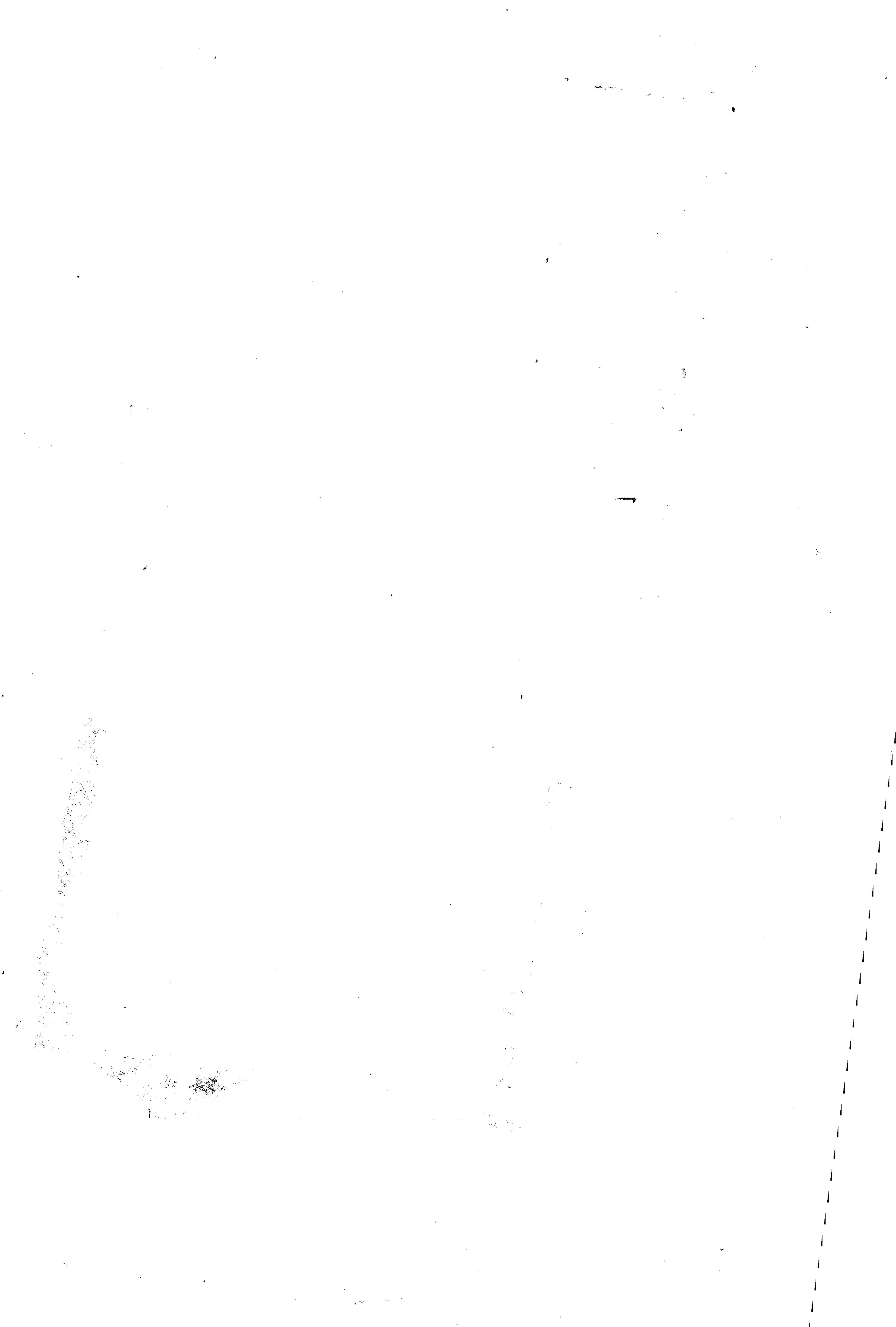
Tiaraju Asmuz Diverio
Dalcídio Moraes Claudio
Philippe Olivier Navaux

RP-177 Maio/92

UNIVERSIDADE FEDERAL DO RIO GRANDE SO SUL
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
Av Bento Gonçalvez, 9500 - Agronomia
91501 - Porto Alegre - RS - BRASIL
Telefone: (051) 336-8399 /339-1355 Ramal 6161
Telex: (051) 2680 - CCUF BR
FAX: (051) 336-5576
E-MAIL: PGCC@INF.UFRGS.BR

Correspondência: UFRGS-CPGCC
Caixa Postal 15064
91501 - Porto Alegre - RS - BRASIL

SABi



SUMÁRIO

LISTA DE SINAIS 4
LISTA DE FIGURAS 5
LISTA DE ALGORITMOS 6
RESUMO 7
ABSTRACT 8

1 INTRODUÇÃO 9

2 INTRODUÇÃO AO PROCESSAMENTO PARALELO 11
2.1 Caracterização do paralelismo 11
2.2 Classificação de arquiteturas segundo Flynn 12
2.3 Máquinas paralelas 15
2.3.1 Máquinas Pipeline 15
2.3.2 Processadores Matriciais 15
2.3.3 Multiprocessadores 16
2.4 Medidas de desempenho de máquinas paralelas 17
2.5 Linguagens paralelas 18

3 PROJETANDO ALGORITMOS PARALELOS 20
3.1 Introdução 20
3.2 Alguns modelos de computação 22
3.2.1 Modelo PRAM (Parallel Random Access Machine) . 23
3.2.2 Modelo SIMD 23
3.2.3 Modelo SIMDAG 24
3.2.4 Modelo MTA (Máquina de Turing Alternada) 24
3.3 Metodologias de obtenção de algoritmos paralelos ... 24

4 NOÇÕES DA ANÁLISE DE ALGORITMOS 29
4.1 Medidas de complexidade e suas cotas 30
4.2 Análise do caso pessimista e do caso médio 34
4.3 Ordem máxima, mínima e exata 35
4.4 Distinções entre funções de complexidade 36
4.5 Tamanho de instâncias de problemas 39
4.6 Questões importantes de complexidade paralela 39
4.7 Complexidade de algoritmos paralelos 40
4.7.1 Tempo de execução 41
4.7.2 Número de processadores 43
4.7.3 Custo 44
4.7.4 Outras medidas 44

5 CLASSES DE COMPLEXIDADE DE TEMPO 46
5.1 Classes de computação sequencial 46
5.1.1 Os espaços P e **NP** e seu relacionamento 47
5.1.2 Problemas em **NP** 48
5.1.3 Problemas **NP-Completos** e Redução Polinomial .. 49
5.2 Classes de computação paralela 50
5.2.1 Classe de problemas **AC** e **NC** 51
5.3 Relação entre várias classes de complexidade de tempo52
5.4 Questões em aberto 53

6 TÉCNICA DE DIVISÃO E CONQUISTA 55
6.1 Formalização da Técnica 55
6.2 Caracterização via aplicações 57
6.2.1 Determinação do máximo e mínimo de uma lista . 57
6.2.2 Ordenação de listas 59
6.2.3 Multiplicação de inteiros 61
6.2.4 Multiplicação de matrizes 64
6.3 Potencialidades da divisão e conquista na paralelização 66
6.4 Exemplo de algoritmo paralelo 68
6.4.1 Ordenação por intercalação Par-Impar 68
6.4.2 Ordenação por intercalação Bitônica 70

7 CONCLUSÕES 73
7.1 Considerações finais 73
7.2 Constatações 74
7.3 Propostas para novos estudos 76

BIBLIOGRAFIA 81
ANEXO A Revisão matemática 86
ANEXO B Considerações sobre algoritmos numéricos paralelos. 90
ANEXO C Levantamento Bibliográfico nos periódicos da biblioteca da PGCC da UFRGS......................116

LISTA DE SINAIS

$\forall$	Para todo
R	Conjunto dos números reais
$\exists$	Existe
$\nexists$	Não existe
$\log_b x$	Logaritmo na base b de x
$\log x$	Logaritmo na base 2 de x
$>$	maior
$<$	menor
$>=$	maior ou igual
$<=$	menor ou igual
PA	Progressão Aritmética
PG	Progressão Geométrica
$C_{n,p}$	Combinação de n elementos p a p
$A_{n,p}$	Arranjo de n elementos p a p
P_n	Permutação de n elementos
$n!$	Fatorial de n
$\sum$	Somatório
Cs(n)	Cota superior de complexidade
Ci(n)	Cota inferior de complexidade
$\Theta(n)$	Ordem exata
$\Omega(n)$	Ordem mínima
$O(n)$	Ordem máxima
$\lceil x \rceil$	Menor inteiro k tal que k >= x
$\lfloor x \rfloor$	Maior inteiro k tal que k <= x
[<=]	Redução Polinomial
RAM	Randomic Access Memories
VLSI	Very Large Scale Integration
SISD	Single Instruction Stream, Single Data Stream
SIMD	Single Instruction Stream, Multiple Data Stream
MISD	Multiple Instruction Stream, Single Data Stream
MIMD	Multiple Instruction Stream, Multiple Data Stream

LISTA DE FIGURAS

2.1 Arquitetura de máquinas SISD 13
2.2 Arquitetura de máquinas SIMD 13
2.3 Arquitetura de máquinas MISD 14
2.4 Arquitetura de máquinas MIMD 14
2.5 Quadro comparativo das arquiteturas 17

3.1 Etapas no desenvolvimento de arquiteturas paralelas 20
3.2 Etapas na resolução paralela de problemas numéricos 21
3.3 Classificação do controle de concorrência 25
3.4 Classificação do módulo de granularidade 26

4.1 Execução do algoritmo 4.1 para entrada <5,47,0,-9> 31
4.2 Gráfico das funções típicas de complexidade 37
4.3 Comparação dos valores das funções típicas
de complexidade 38
4.4 Comparação das funções de complexidade ([TER91]) 38

5.1 Circuito NC ([TER90])................................... 52
5.2 Classes de complexidade ([TER90]) 53

6.1 Estrutura recursiva da técnica de divisão e conquista .. 55
6.2 Pilha de execução recursiva do algoritmo MaxMin 58
6.3 Pilha de execução recursiva do algoritmo OrdInter 61
6.4 Pilha de execução recursiva do algoritmo Mult 63
6.5 Diagrama do método par-impar para n=2 69
6.6 Circuíto para ordenar a sequência S 69
6.7 Intercalador bitônico para uma sequência com 4 elementos 71
6.8 Intercalador bitônico (m=min M=max) 71
6.9 Circuíto para ordenar (4,8,1,3,2,7,5,6) 72
6.10 Comparação das complexidades 72

b1 Unidades de tempo necessárias para somar 16 números
utilizando p processadores 91
b2 Comparação entre processamento sequencial e paralelo 92
b3 Execução do algoritmo da soma parcial por cascata (modelo
com 8 processadores)..................................... 94
b4 Interpretação geométrica do método da Bissecção 98
b5 Valores parciais do método da Bissecção para o
polinômio Pn(x) 100
b6 Interpretação geométrica do método da Bissecção para uma
máquina com dois processadores 101
b7 Resolução pelo método da Bissecção por uma máquina de
dois processadores 103
b8 Interpretação geométrica do método Regula-falsi 104
b9 Valores parciais do método Regula-falsi para o
polinômio Pn(x) 106
b10 Gráficos da determinação dos zi 107
b11 Valores parciais do método Regula-falsi versão paralela
para Pn(x), usando máquina com dois processadores 109
b12 Problema do estado cte de distribuição de temperatura ..111

RESUMO

Neste trabalho é feito um estudo de algoritmos baseados na técnica de divisão e conquista. Para tanto, foi descrita a técnica com seus princípios e exemplificada através de alguns algoritmos. Entre os objetivos deste trabalho estão presentes a análise do ganho de rapidez na solução do problema (complexidade) e a análise das potencialidades de paralelização. Uma introdução à complexidade de algoritmos e noções sobre as classes de complexidade. Por fim, foram analisados vários problemas em que esta técnica tem se mostrado eficiente na solução dos problemas.

Palavras-chave:

Processamento paralelo, algoritmos paralelos, complexidade de algoritmos sequenciais e paralelos, Classes de complexidade e Divisão e conquista.

LISTA DE ALGORITMOS

1.1 Cálculo do fatorial 9

4.1 Ordenação de lista em ordem não decrescente 31
4.2 Quicksort - ordenação por concatenação 35

6.1 Determinação do máximo e mínimo de uma lista 58
6.2 Ordenação por intercalação 60
6.3 Multiplicação de dois inteiros 62

B1 Método da soma parcial por cascata 94
B2 Algoritmo da redução cíclica 96
B3 Algoritmo do método da bissecção sequencial 99
B4 Versão paralela do método da bissecção 102
B5 Algoritmo do método Regula-falsi sequencial 105
B6 Algoritmo do método Regula-falsi paralelo 107
B7 Rotina de ordenação 109

ABSTRACT

In this work is presented a algorithms study based on the divide and conquer technic. So, it was described the technic with its principles and showes by some algorithms. This work has objectives like analysis and complexity of algorithms and the analysis of the potential of the parallelization. This work approachs also the sequential and parallel algorithms and notions of classes of complexity. At last, many problems, in which this technic is efficient in the solution of problems were analysed.

Key-word:

Parallel processing, parallel algorithms, sequential and parallel complexity of algorithms, Classes of complexity, Divide-and-conquer Technic.

1 INTRODUÇÃO

Um algoritmo por divisão e conquista divide o problema em vários subproblemas do mesmo tipo, mas menores, que podem ser resolvidos diretamente ou subdivididos novamente, usando a mesma técnica, até que possam ser resolvidos. Então, acha-se uma forma de combinar as soluções parciais para se obter a solução para o problema original.

Os subproblemas resultantes da aplicação do método são do mesmo tipo que o problema original; assim a aplicação sucessiva do método pode ser expressa naturalmente por um algoritmo recursivo, isto é, dentro de um algoritmo chamado M, com uma entrada de certo tamanho, usamos o próprio M, k vezes para resolver subentradas de tamanhos menores. Um exemplo é o algoritmo 1.1 que calcula a função matemática fatorial (n!).

```
ALGORITMO 1.1
  Fat(n) - Cálculo do fatorial de n;
Entrada:   n - um número inteiro maior ou igual a zero;
Saída:     Fat o valor do fatorial de n.
1  Se n=0
2     então Pare com saída(1)
3     senão val=n.Fat(n-1)
4           Pare com saída(val)
5  fim-se.
```

Observa-se que se n=0 o valor do fatorial é um, pois 0!=1. Para os demais valores, o algoritmo chama-se a si mesmo n vezes com as instâncias de n-1 a zero.

Os algoritmos recursivos são relativamente fáceis de serem escritos e compreendidos, além de serem de fácil análise quanto a certificação.

Esta técnica de divisão e conquista para algoritmos é poderosa para resolução de problemas numéricos e não numéricos, em virtude do ganho de rapidez na solução do problema. Para tal análise é necessário recordar alguns conceitos matemáticos e de análise de algoritmos.

O estudo da área de desenvolvimento de algoritmos tem crescido pelos resultados obtidos quanto a economia de recursos computacionais que um algoritmo desenvolvido apropriadamente pode proporcionar, além do fato de compreensão do próprio algoritmo. Tem-se podido verificar como técnicas matemáticas são aplicadas a problemas de computação, de forma a obter ganhos de rapidez e facilidades no solucionar problemas eficientemente.

No capítulo dois é feito uma introdução ao processamento paralelo, através da caracterização do conceito de paralelismo e de máquinas paralelas, da forma de execução de instruções sobre dados, resultando na classificação de Flynn para arquiteuras paralelas e nos três principais tipos de máquinas paralelas: pipeline, processadores matriciais e

multiprocessadores. E, ainda, são apresentadas medidas de desempenho e as abordagens das linguagens paralelas.

No capítulo três, é feito uma introdução ao projeto de desenvolvimento de algoritmos paralelos. Uma vez que o projeto de algoritmos está intimamente ligado aos modelos de computação, estes são brevemente comentados. São, ainda, apresentadas metodologias de obtenção de algoritmos paralelos.

No capítulo quatro são apresentados os conceitos básicos de análise de algoritmos e complexidade, como as cotas, análise do caso pessimista e do caso médio, conceito de ordem, distinções entre funções de complexidade. Algumas questões sobre complexidade de algoritmos paralelos são apresentadas e comentadas. São definidos alguns critérios úteis na avaliação da complexidade de algoritmos paralelos, como: tempo de processamento, número de processadores, custo e outras medidas.

No quinto capítulo, procurou-se introduzir uma noção das classes de complexidade de tempo e espaço e estabelecer uma relação entre as várias classes de complexidade de tempo.

No sexto capítulo é então caracterizada a técnica de divisão e conquista através de vários problemas, os quais possuem soluções conhecidas através de algoritmos por divisão e conquista. São apresentados estes algoritmos e é feito uma certificação e uma análise de seu ganho computacional. Entre os problemas apresentados estão: localização do máximo e mínimo de uma lista; ordenação de listas; busca binária; multiplicação de inteiros e multiplicação de matrizes.

São, ainda, ressaltadas algumas caracteristicas desta técnica de "dividir para conquistar", que são identificadas como qualidades para facilitar a paralelização de algoritmos. Por fim, são apresentadas as conclusões deste trabalho e as sugestões e propostas para novos estudos.

No anexo A, Revisão de matemática, são apresentados conceitos básicos matemáticos e fórmulas úteis quando se quer desenvolver estudos de análise de algoritmos e complexidade. No anexo B, é apresentado um estudo parcial, contendo considerações sobre algoritmos numéricos paralelos. Neste estudo, foram abordados alguns problemas numéricos como equações de recorrência, resolução de equações algébricas, sistemas de equações lineares e equações diferenciais. E no anexo C, consta um levantamento bibliográfico em periódicos disponíveis na biblioteca do PGCC da UFRGS, de temas afins, no período de 1976 a janeiro de 1992.

Finalmente, queremos agradecer aos bolsistas Luis da Cunha Lamb, Paulo Ricardo Trainini e Leonardo Carissimi pelo auxílio no desenvolvimento dos anexos e ao CNPq pelo auxilio aos nossos projetos de pesquisa.

2 INTRODUÇÃO AO PROCESSAMENTO PARALELO

2.1 Caracterização de Paralelismo

A busca de computadores com desempenho superiores aos existentes em cada momento e o aumento do poder computacional, tem sido atingido pela exploração de aspectos básicos, como tecnologia, algoritmos e arquitetura. O aperfeiçoamento tecnológico refere-se à pesquisa para confecção de componentes eletrônicos mais velozes. Entretanto, existem limitações que se devem a fatores como velocidade máxima de propagação de sinais e chaveamento de transistores.

Quanto ao aspecto do desenvolvimento de novos algoritmos, tem sido um processo lento, devido especialmente ao fato que muitos dos algoritmos já estão próximo de seus limites teóricos de eficiência.

O aspecto da evolução das arquiteturas dos computadores, em particular com o uso de processamento paralelo, permite superar os limites de desempenho impostos pelos fatores tecnológicos de seus componentes.

O emprego de algoritmos e arquiteturas paralelas tem se constituído em uma alternativa promissora para superar as barreiras de desempenho imposta pelo estágio da tecnologia de componentes eletrônicos.

Uma arquitetura paralela geralmente é um sistema de propósito específico, otimizado para realizar de forma eficiente uma certa gama de funções dentro de uma aplicação.

Máquinas paralelas, também denominadas de computadores paralelos, são sistemas que executam explicitamente avaliações computacionais em paralelo, explorando a concorrência de eventos. A concorrência pode ser de três tipos: paralelismo, simultaniedade e pipeline.

No Paralelismo os eventos paralelos ocorrem em múltiplos recursos no mesmo instante de tempo. É o paralelismo de recursos assíncronos.

Na Simultaniedade, os eventos são simultâneos, ocorrem no mesmo instante de tempo. É o paralelismo de recursos sincronos.

No Pipeline os eventos pipelines ocorrem em instantes sobrepostos. É o paralelismo temporal.

Em função do tipo de paralelismo empregado, os computadores paralelos são divididos em três grupos: máquinas pipelines, processadores matriciais e multiprocessadores.

As máquinas Pipeline superpõem a execução de instruções explorando um paralelismo denominado temporal, onde os eventos ocorrem em instantes de tempos sobrepostos.

Processadores matriciais empregam a multiplicidade de unidades processadoras executando instruções sincronamente utilizando o paralelismo dito espacial síncrono, onde os eventos são simultâneos, pois ocorrem em múltiplos recursos no mesmo instante de tempo.

Por último, temos o paralelismo espacial assíncrono, que é empregado em sistemas multiprocessadores através da divisão de recursos comuns (memória, periféricos, etc...) por um conjunto de processadores, onde os eventos ocorrem em múltiplos recursos no mesmo instante de tempo.

As estruturas que exploram o paralelismo são vinculados à maquina em que a aplicação será executada. Essa característica implica em que o programador deve ter o conhecimento da arquitetura da máquina empregada e levar isso em consideração quando da escrita do algoritmo ou programa. Isso se constitui em uma dificuldade adicional na elaboração de programas para máquinas paralelas.

2.2 Classificação de arquiteturas segundo Flynn

M. Flynn ([FLY72]) propos a classificação que leva em conta a forma pela qual é executada uma instrução em um conjunto de dados, pois em qualquer computador, seja ele sequencial ou paralelo, opera pela execução de instruções sobre dados. O fluxo de instruções (algoritmo) diz ao computador o que fazer a cada passo. O fluxo de dados é afetado por estas instruções. Dependendo do número destes fluxos, Flynn distinguiu quatro categorias ou classes de arquiteturas:

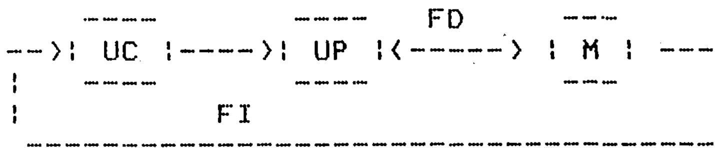
SISD - Single Instruction stream Single Data stream

SIMD - Single Instruction stream Multiple Data stream

MISD - Multiple Instruction stream Single Data stream

MIMD - Multiple Instruction stream Multiple Data stream

A categoria **SISD** tem fluxo de instrução e de dados únicos. São máquinas baseados nos princípios de Von Neumann. As instruções são executadas sequencialmente, mas podem ser sobrepostas nos seus estágios de execução. Exemplos são arquiteturas pipelines. Foi este tipo de arquitetura que proporcionou o surgimento de máquinas de alto desempenho chamadas de supercomputadores.



UC Unidade de controle
M Memória
FI Fluxo de Instruções (único)
UP Unidade de Processamento
FD Fluxo de Dados (único)

Fig.2.1 Arquitetura de máquinas SISD

A categoria **SIMD** tem um único fluxo de instruções com vários fluxos de dados. Nesse tipo de arquitetura vários elementos de processamento (EP) são supervisionados por uma única unidade de controle, que encaminha a mesma instrução para execução nos elementos sobre seus dados. Nestas máquinas, uma única operação é executada simultaneamente por diversos elementos de processadores sobre dados distintos. A memória pode ser dividida em módulos vinculados a cada elemento de processamento ou então, ser de acesso global. Exemplos são arquiteturas matriciais.

No caso de utilização de memória local, existe uma rede de interconexão. O acesso para leitura e gravação pode ser exclusivo ou concorrente, o que subdivide mais ainda esta categoria, como pode ser visto em [AKL89] e em [HWA84].

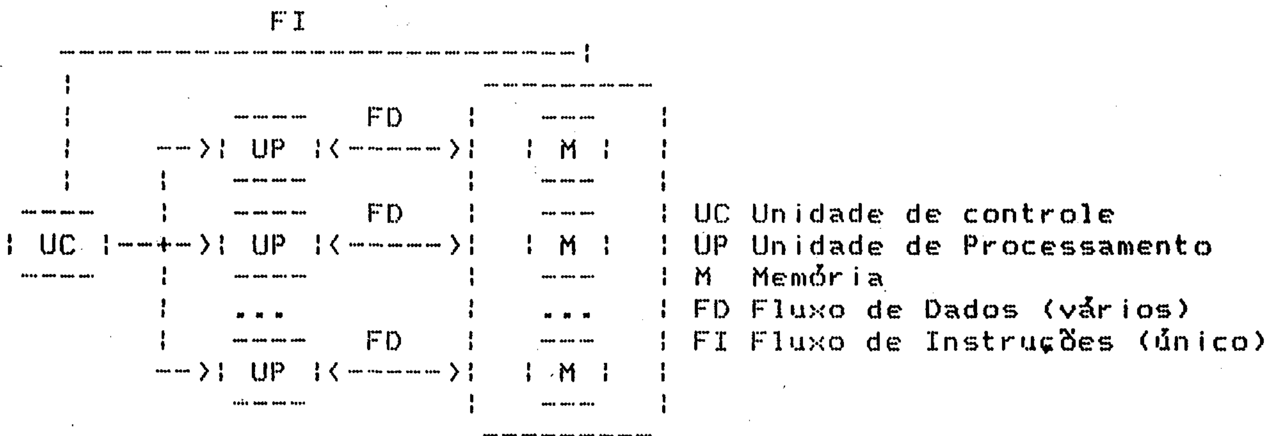


Fig.2.2 Arquitetura de máquinas SIMD

A categoria **MISD** se caracteriza por ter múltiplos fluxos de instruções e um único fluxo de dados. Esta categoria corresponde a uma máquina que possua vários elementos de processamento recebendo instruções distintas de várias unidades de controle, que processam o mesmo fluxo de dados.

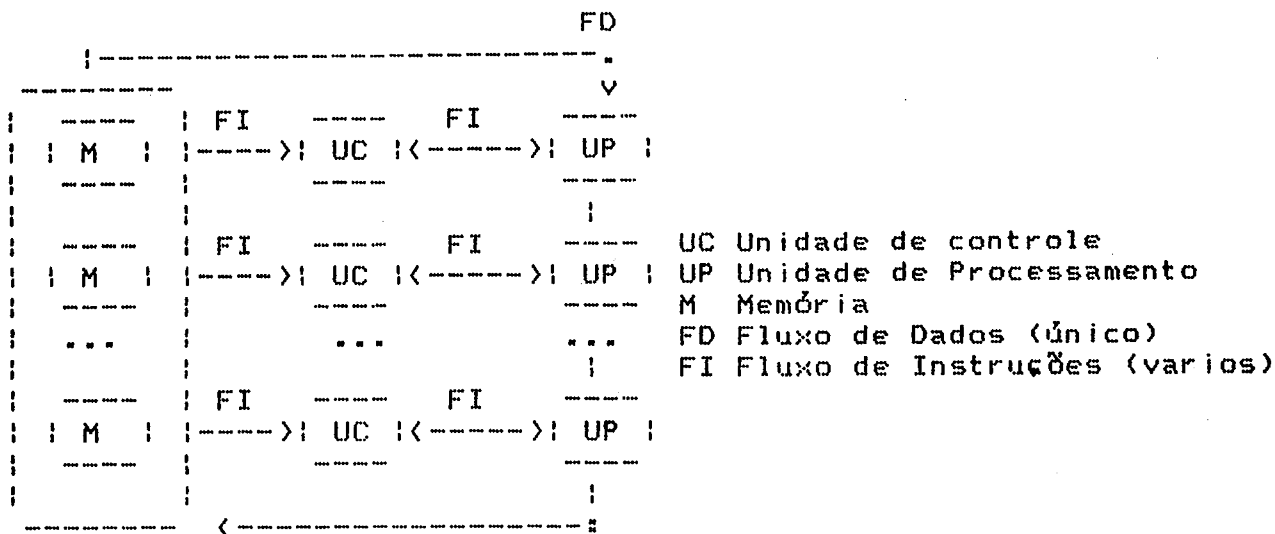
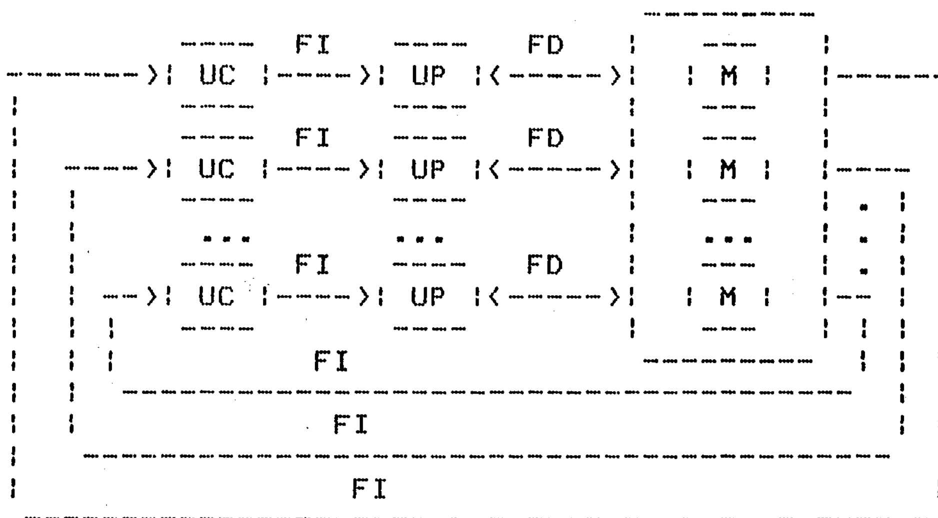


Fig.2.3 Arquitetura de máquina MISD

Por fim, a categoria **MIMD** se caracteriza por ter múltiplos fluxos de instruções com múltiplos fluxos de dados. Nessa arquitetura cada unidade de controle possui sua unidade de processamento executando instruções sobre um conjunto de dados de forma independente. São as máquinas capazes de executar simultaneamente diversas instruções, sobre dados distintos. A maioria dos sistemas multiprocessadores são deste tipo. A interação entre os processadores é obtida através da memória global ou sistema de memórias gerenciado por um único sistema operacional.



UC Unidade de controle
UP Unidade de Processamento (Elemento de Processamento)
M Memória
FD Fluxo de Dados (varios)
FI Fluxo de Instruções (varios)

Fig.2.4 Arquitetura de máquinas MIMD

Computadores MIMD que compartilham memória comum são referenciados como multiprocessadores, enquanto que todos os com rede de interconexão são referenciados como multicomputadores, algumas vezes são referenciados como sistemas distribuídos.

As máquina SIMD correspondem algoritmos síncronos com passos de espera, os quais requerem controle central, já as máquinas MIMD correspondem algoritmos assíncronos com uma relativa granularidade.

2.3 Máquinas paralelas

2.3.1 Máquinas Pipeline

É um exemplo de paralelismo temporal, onde uma tarefa é subdividida numa sequência de subtarefas de modo que seja executada mais fácil e mais rapidamente, pois cada uma é executada por um estágio de hardware específico, que trabalha concorrentemente com os outros estágios do pipeline, criando desta forma, um paralelismo temporal na execução das subtarefas. Um exemplo clássico é uma fábrica de automóveis, os quais são feitos em série ou por uma linha de produção, onde cada grupo especializado realiza uma subtarefa, de forma que um carro demore o mesmo tempo de fabricação, mas assim que o primeiro for produzido, a cada período t de tempo, teremos outro carro produzido. Exemplos de máquinas: CRAY 1 e 2, CYBER 200, NETS, FUJITSU (VP 200 e VP 400).

2.3.2 Processadores Matriciais

Também denominados de array processors ou SIMD, são exemplos de paralelismos espacial síncrono.

Um processador matricial é um conjunto paralelo síncrono com múltiplas unidades lógicas e aritméticas (ULA), denominados de elementos processadores (PE), que podem operar em paralelo de forma síncrona, realizando uma mesma função em um mesmo instante. Uma unidade de controle distribui para vários elementos de processamento, a mesma instrução que será executada em paralelo por esses EPs sobre seus dados no mesmo instante de tempo. Um exemplo , é a operação com matrizes, por exemplo multiplicar a matriz por dois. Isto implica em nove multiplicações (matriz quadrada de ordem três), mas como são feitos pelos elementos de processamento, leva-se apenas uma unidade de tempo. Exemplos de máquinas são: Illiac IV, MPP, Pepe, Staran.

Os processadores matriciais utilizam o endereço do dado para acesso, existe uma variação de processadores, os quais acessam os dados via conteúdo; esses exploram a capacidade de memórias associativas e, por isso, são denominados **matriciais associativos.**

Os processadores matriciais são compostos por uma unidade de controle, um conjunto de elementos processadores, uma memória associada a cada elemento processador e uma rede de interconexão. A unidade de controle é responsável pelo gerenciamento da operação dos elementos processadores decodificando as instruções a serem executadas e gerenciando os sinais de controle.

Alguns exemplo de emprego de máquinas SIMD são: algoritmo de mapeamento do nível de cinza como o gradiente de Sobel (Processamento de imagens); Processamento simbólico, como algoritmos de sorting e manipulação de grafos e solução de sistemas de equações [CAP85], operações com matrizes, resolução de equações diferenciais [SCH84].

Cada um dos elementos de processamento pode operar com operandos do tipo palavra ou com um único bit, durante cada ciclo da memória. Esta característica define dois tipos de sistemas SIMD; o da organização de palavras (word-organized) e o de organização de bits (bit-organized). As operações aritméticas em um sistema de organização por palavras é análogo ao de máquinas seriais; já os organizados por bit, tem influenciado grandemente o projeto de algoritmos paralelos.

O termo processador de array tem certa ambiguidade, pois pode referir a máquinas SIMD ou máquinas MIMD. O processador matricial geralmente incorpora um grande número de elementos processadores idênticos conectados numa particular topologia.

2.3.3 Multiprocessadores

Também conhecidos como arquiteturas MIMD são exemplos de paralelismo espacial assíncrono. É o multiprocessamento propriamente dito, onde diversos processadores trabalham em paralelo, processando suas tarefas concorrentemente de forma assíncrona para num intervalo de tempo concluirem a tarefa. É um conjunto de processadores que se comunicam e cooperam para resolverem uma dada tarefa.

Classif de Flynn	Concorrência	Máquinas	Paralelismo	Exemplos de Maq.
SISD	Pipeline	Pipeline	Temporal	Cray 1 2 Ciber200 Nets Fujitsu
SIMD	Simultaniedade	Matriciais Arrays	Espacial Síncrono	Illiac IV MPP Pepe Staran
MISD	-- x --	-- x --	-- x --	Não Há
MIMD	Paralelismo	Multipro- cessadores	Espacial Assíncrono	Cray-X-MP CM* HELP

Fig.2.5 Quadro comparativo das arquiteturas

2.4 Medidas de desempenho de máquinas paralelas

Benchmarking consiste em executar um conjunto de programas bem conhecidos, comparando o desempenho com os obtidos em outras máquinas para os mesmos programas. O objetivo do benchmarking não é obter um único número representando o desempenho de uma determinada máquina e sim, entender como o desempenho geral é afetado por cada um dos itens componentes do mesmo (uma medida relativa).

As medidas absolutas para desempenho mais usuais são: a quantidade de instrução realizada por segundo - **MIPS (million of instructions per second)** e a quantidade de operações em ponto-flutuante executadas por segundo - **MEGAFLOPS (millions of floating operations per second).** Uma relação que se pode estabelecer entre elas é que um MFLOPS é aproximadamente 2 a 5 MIPS.

Outra medida é o **LIPS (logical inference per second),** que mede a quantidade de inferências que se pode fazer por segundo. É utilizada em máquinas paralelas para inteligência artificial.

2.5 Linguagens paralelas

No desenvolvimento de linguagens paralelas e na expressão do paralelismo, existem três abordagens a serem consideradas (segundo [PER90]). A primeira abordagem é denominada de abordagem **explícita**, que consiste em desenvolver uma linguagem paralela. Entre os exemplos existentes desta abordagem pode-se destacar: ADA como uma linguagem imperativa; STRAND como uma linguagem de programação funcional e POOL como exemplo de uma linguagem paralela orientada a objeto.

Um dos problemas solucionados com o tempo foi o de sincronização. Nesta abordagem, duas técnicas foram empregadas. A primeira técnica é baseada na construção de um monitor e variáveis de condição. A outra técnica é baseada em processos que passam informações diretamente quando eles desejam se comunicar. Entre as linguagens que usam monitores como mecanismo de sincronização temos: Concurrent Pascal (1974), Modula (1977), Pascal Plus (1979). Entre as linguagens que permitem que os processos troquem informações têm-se CSP (communicating sequential process). A sincronização é feita através da troca de mensagens através dos comandos de entrada e saída.

Outro exemplo é a notação conhecida como Processos Distribuídos (DP), na qual o processo chamado não necessita saber o nome do processo que o chamou e é utilizada nas linguagens ADA e OCCAM.

Entre as vantagens da utilização de uma nova linguagem é o fato dela promover uma abordagem coerente para paralelismo, possibilitando uma notação capaz de se desenvolver novos algoritmos paralelos; por outro lado um desvantagem do uso de uma nova linguagem é a necessidade de treinamento aos usuários e a reconstrução de toda a base de software (bibliotecas), um trabalho intenso e sujeito a erros.

Um ponto desfavorável na criação de linguagens paralelas é o fato que muitas vezes as estruturas a serem criadas para explorar o paralelismo estão fortemente vinculadas à maquina em que serão executadas e a suas aplicações. As linguagens paralelas já desenvolvidas estão vinculadas a uma determinada máquina, não existindo, portanto, uma linguagem que seja universal.

A segunda abordagem e, talvez, a mais utilizada, é a utilização de uma linguagem já existente onde o compilador verifica partes paralelizáveis em máquinas multiprocessadora. Esta abordagem é conhecida como **implícita**, uma vez que o usuário escreve o programa de forma convencional, de uma forma sequencial e o compilador da linguagem é quem se responsabiliza pela detecção e exploração do paralelismo do programa, a paralelização fica transparente ao usuário.

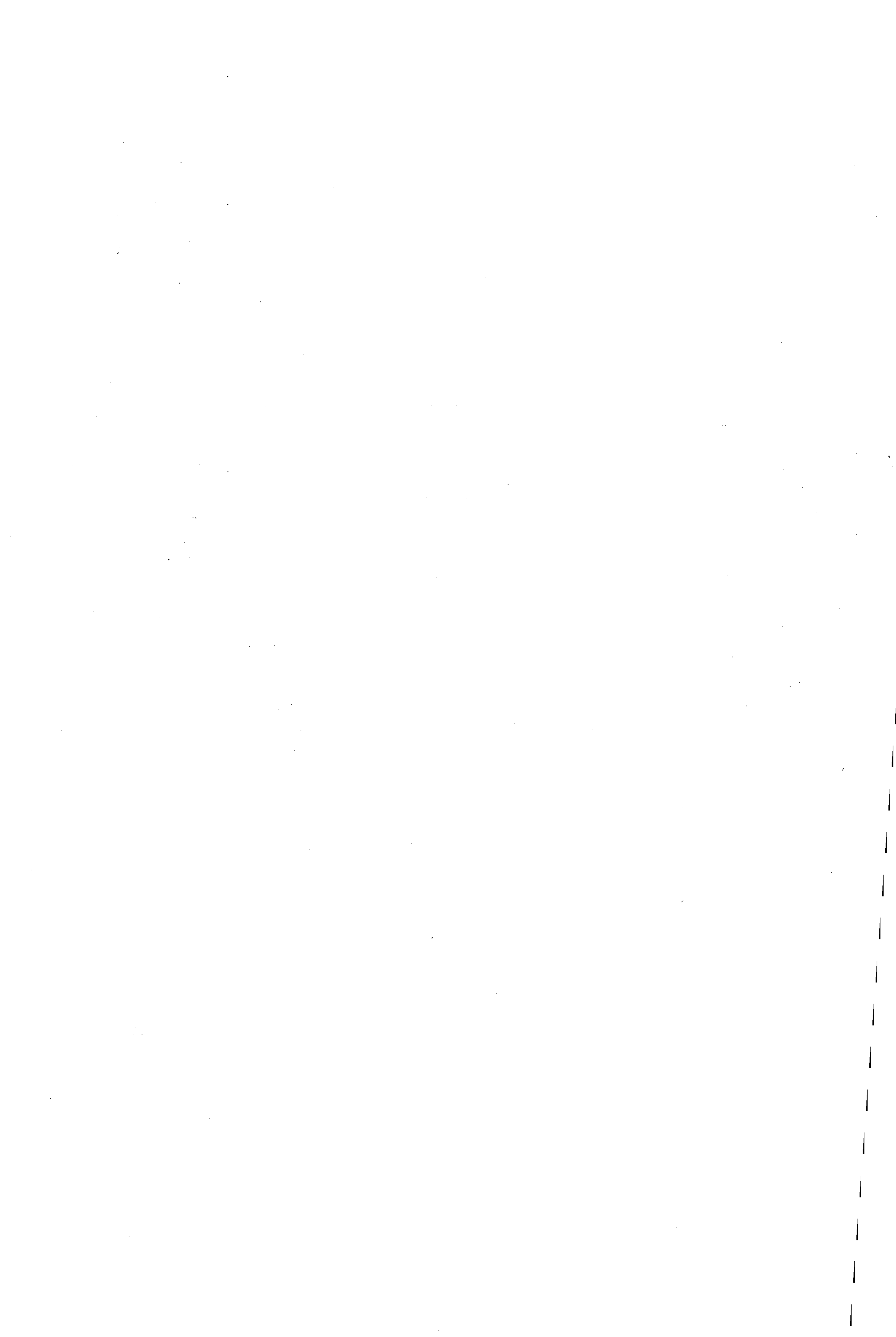
Esta abordagem pode ser subdividida em outras duas: uma associada a linguagens funcionais e a outra associada a linguagens convencionais. As linguagens funcionais são pouco confiáveis quanto a ordem de avaliação das operações em um programa. As linguagens convencionais necessitam que se utilize um compilador que determine as partes da aplicação a serem executadas em paralelo.

O principal problema desta técnica são as dependências de dados. É necessário detectá-las quando ocorrem e verificar se o código pode ser refeito automaticamente ou pelo usuário para permitir o paralelismo.

A principal vantagem desta técnica é que programas já existentes podem ser transferidos, portados a baixo custo, para uma nova máquina. Ao se projetar novos programas deve-se observar as regras de reestruturação que detectam o paralelismo. Estas regras não acrescentam facilidades para programação paralela, antes limitam a extração do paralelismo e ocasionam a geração de código não otimizada.

Por fim, a terceira abordagem que é conhecida por **extensões de linguagens**, já que são linguagens sequenciais existentes acrescidas de novas características para manipular o paralelismo.

As extensões de linguagens são vantajosas pelo fato de que estas podem ser feitas através da construção de bibliotecas de subrotinas e, ainda, os usuários não necessitam aprender uma nova linguagem, somente suas extensões, as quais são introduzidas gradativamente.



3 PROJETANDO ALGORITMOS PARALELOS

3.1 Introdução

A inovação da computação paralela tem adicionado uma nova dimensão no projeto de algoritmos e programas. As aplicações de paralelismo atingem diversos níveis: programas, algoritmos, comandos e instruções. Programação paralela não é uma simples extensão de programas sequenciais, pois para se explorar as possibilidades oferecidas pelo paralelismo, os programadores necessitam "pensar em paralelo" e reconsiderar a solução do processo. Experiências têm mostrado que julgamentos de eficiência baseados em técnicas sequenciais podem ser facilmente convertidas a ambientes paralelos.

Por outro lado, há algoritmos que são obsoletos para serem implementados em máquinas seriais por possuirem alto grau de paralelismo, isto é, conter muitos cálculos que são independentes uns dos outros, mas que podem ser executados simultaneamente, obtendo um desempenho melhor do que técnicas seriais. A dificuldade reside no fato de que a nova técnica de projeto de algoritmos - a introdução de paralelismo extendido, também requer que seja repensado o conhecimento do sistema, de linguagens paralelas, dos problemas não numéricos e dos métodos numéricos em geral, pois as características de computadores paralelos e algoritmos paralelos são qualitativamente diferentes de todas as computações sequenciais.

Um problema não trivial é como traduzir de forma eficiente um algoritmo computacinal em uma implementação física. Essa tradução deve envolver, segundo Carissimi [CAR89], pelo menos, quatro categorias: teoria da computação, uma representação implicita ou explícita do algoritmo correspondente à aplicação, uma arquitetura computacional abstrata e a implementação física, conforme é ilustado na figura 3.1.

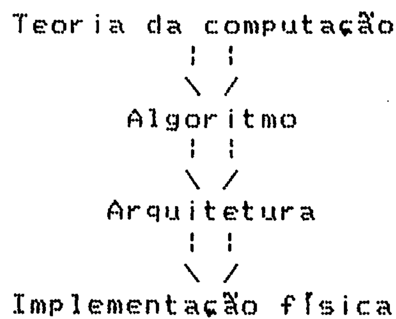


Fig.3.1 - Etapas no desenvolvimento de arquiteturas paralelas

Através desta idéia, podemos projetar uma alteração na figura que ilustra os níveis de abstração dada por Diverio,T.A em [DIV86].

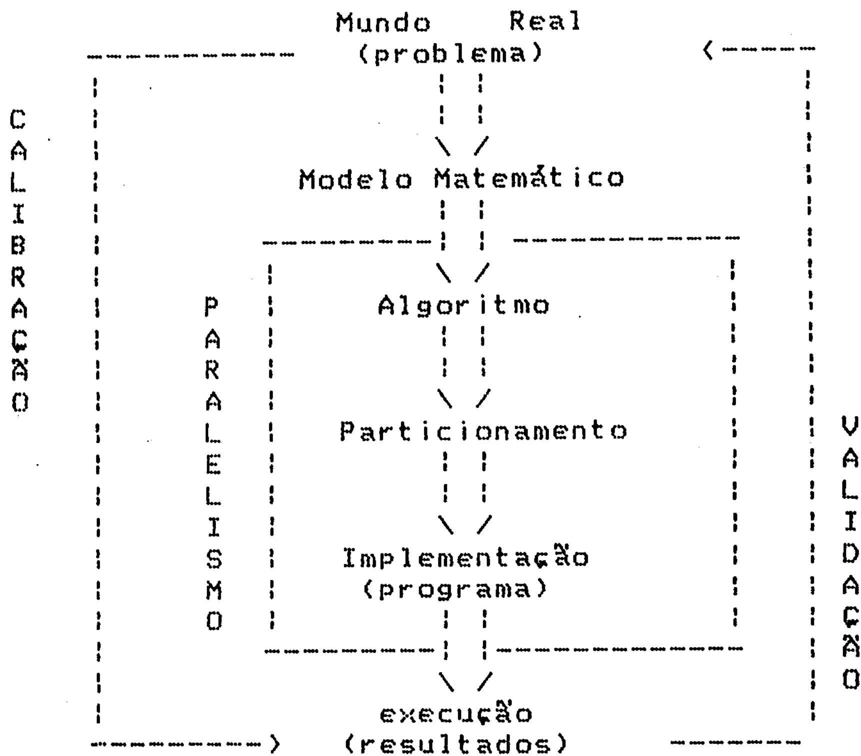


Fig.3.2 -Etapas na resolução paralela de um problema numérico

A relação aplicação-algoritmo-arquitetura exerce influência fundamental na exploração do paralelismo. Os principais fatores influenciados são: representação e descrição do paralelismo, estrutura de dados, mapeamento do algoritmo à arquitetura, esquemas para fluxo de dados e de controle, desempenho entre outros.

De uma "boa" implementação em hardware depende a eficiência na execução do algoritmo desejado. Pode ocorrer o problema de algoritmos não serem facilmente mapeados, pois uma carência na área é a falta de um formalismo eficiente para representar relações, para o mapeamento de algoritmos.

A dificuldade em obter-se um mapeamento eficiente é acentuada quando a estrutura de comunicação do algoritmo é distinta da arquitetura e quando o número de processadores é menor que o número de tarefas definidas para a execução do algoritmo.

A qualidade de um mapeamento é medida em função do balanceamento de carga entre os elementos de processadores, número de canais virtuais de comunicação existentes e pelos comprimentos máximos e total dos caminhos físicos de comunicação. A determinação do melhor mapeamento possível para cada caso é um problema NP-completo ([GAR79]), como será visto no capítulo 5.

O mapeamento de algoritmos pode ser feito através de quatro linhas básicas, estas são:

a) A aplicação da heurística pura - não existe um método específico, o projeto de array é desenvolvido conforme a inspiração e experiência do projetista;

b) A avaliação do balanceamento das operações - consiste em realizar um levantamento estático das operações necessárias em uma dada classe de algoritmos e, então, implementar um array com alguma topologia conveniente, onde cada célula apresente todos os operadores básicos na proporção do levantamento estático;

c) A utilização de regras de projeto pré-definidas - determina-se um conjunto de regras segundo a experiência dos projetistas de antemão. Esta proposta baseia-se numa linguagem para descrição do algoritmo, num conjunto de regras de transformação e num sistema automático para aplicação dessas regras;

d) A utilização de técnicas semi-automáticas de projeto - onde o princípio básico adotado consiste em eliminar completamente as dependências existentes no fluxo de dados de forma automática.

3.2 Alguns modelos de computação

O desenvolvimento de algoritmos mais rápidos tem sido muito lento, devido especialmente ao fato que muitos dos algoritmos já estão próximos de seus limites teóricos de eficiência. O emprego de algoritmos e arquiteturas paralelas tem se constituido em uma alternativa promissora para superar as barreiras de desempenho impostas pelo atual estágio da tecnologia de componentes eletrônicos.

As estruturas que exploram o paralelismo são vinculadas à máquina em que a aplicação será executada. Esta característica implica em que o projetista ou programador deva ter o conhecimento da arquitetura da máquina empregada e levar isto em consideração quando do projeto do algoritmo ou escrita do programa. Isto se constitui em uma dificuldade adicional na elaboração de projetos de algoritmos e de programas para máquinas paralelas.

Tem-se que ter em mente que a tarefa de projetar algoritmos paralelos é uma tarefa que depende do tipo de modelo computacional, do tipo de máquina que será utilizado ou que se está considerando, no caso teórico. O tipo de máquina implica não só na quantidade de processadores disponíveis, mas também o tipo de disposição deles, da topologia dos processadores ou elementos processadores, do tipo de memória e na forma de comunicação entre os processadores.

Uma grande variedade de modelos de computação paralela têm sido definidos, e é interessante estudar as regras que existem entre eles, a viabilidade de construção de cada um e quais outros modelos de computação paralela podem ser imaginados e construídos.

Os modelos de computação paralela diferem entre si de várias maneiras, incluindo o arranjo topológico e a interconexão de seus componentes (processadores). Existem instruções internas a cada processador e instruções interprocessadores. Estas instruções não estão pré-definidas e variam de modelo para modelo, caracterizando-os e determinando seu potencial de paralelização.

No objetivo de determinação de bons modelos em relação à viabilidade de construção e implementação, temos os modelos PRAM, SIMD (principalmente por ter um modelo de interconexão fixa). Nos modelos mais atuais temos modelo com memória compartilhada (onde estão SIMDAG), onde cada processador tem acesso a memória de outros processadores.

3.2.1 Modelo PRAM (Parallel Random Access Machine)

É uma extenção do modelo RAM e é constituído de vários processadores sequenciais independentes, cada um com um conjunto particular de registradores e comunicando-se com os outros processadores através de um conjunto global de registros. Em cada ciclo de execução, cada processador pode ler um registro particular ou global, executar uma instrução RAM e gravar um registro particular ou global.

Conforme as restrições de leitura ou gravação simultânea na memória global, pode-se ter uma variedade de PRAMs. Diz-se que a leitura ou gravação é concorrente se a simultaniedade é permitida e, exclusiva no caso contrário. Assim, em um PRAM do tipo EREW (de Exclusive Read, Exclusive Write) não se permite leitura ou gravação simultânea em qualquer registro global, por qualquer par de processadores. Por outro lado, em um PRAM do tipo CREW (de Concurrent Read, Exclusive Write) só leituras simultâneas são permitidas.

3.2.2 Modelo SIMD

O modelo SIMD também chamado modelo de interconexão fixa, é um sistema composto por uma unidade de controle e um conjunto de processadores com memória local, interconectadas por uma rede de interconexão. Os processadores são altamente sincronizados, ou seja, todos os processadores devem completar a execução da instrução concorrente antes de qualquer processador começar a próxima instrução.

Um processador paralelo está ativo somente quando alguma instrução é delegada a ele. Dois processadores paralelos devem estar executando a mesma instrução se ambos estão ativos ao mesmo tempo (Single Instruction Stream), mas podem estar operando sobre valores distintos (Multiple data stream).

3.2.3 Modelo SIMDAG

O modelo SIMDAG consiste do modelo SIMD com uma memória global de acesso aleatório, um processador de controle e um conjunto de unidades de processamento paralelo. Neste modelo não é permitido a interconexão entre processadores.

3.2.4 Modelo MTA (Máquinas de Turing Alternada)

Alternação é uma generalização do conceito não determinístico no qual quantificadores existenciais e universais podem alternar durante uma computação.

Uma máquina de Turing Alternada M é uma sete-upla ordenada contendo: número de fitas de trabalho (k); um conjunto finito de estados (Q); um alfabeto de entrada; um alfabeto de trabalho; regra de transição; o estado inicial e uma função que determina o estado universal.

A MTA tem uma fita de entrada onde é permitido somente leitura e k fitas de trabalho, inicialmente vazias. Um passo de computação em M consiste em ler um símbolo de cada fita, escrever um símbolo em cada fita de trabalho, mover cada cabeça de leitura uma célula para a esquerda/direita e entrar num novo estado de acordo com a regra de transição.

3.3 Metodologias de obtenção de algoritmos paralelos

Um dos problemas principais na área de processamento paralelo, encontra-se na tradução de uma aplicação para uma forma de avaliação paralela. As principais dificuldades na conversão em algoritmos paralelos vêm de três fontes: unidade de controle, diferença conceitual de programação e identificação de pontos que podem ser paralelizados.

A unidade de controle das máquinas paralelas em geral são unidades sequenciais, o que implica em que procedimentos paralelos sejam organizados de uma forma sequencial.

A diferença conceitual de programação que existe entre os computadores convencionais e os paralelos, como por exemplo em processadores do tipo matricial onde o programador deve ter sempre em mente que está operando sobre vários processadores e a participação ou não de um processador em determinada operação deve ser feito de forma a mascará-lo. Um algoritmo paralelo é dependente da máquina até o nível mais baixo, o nível da arquitetura, de como realmente a máquina vai fazer tal operação, enquanto que em algoritmos sequenciais isto é irrelevante.

A identificação de pontos que possam ser paralelizados não é de fácil visualização, podendo, as vezes, a simples troca de um índice ou a inversão na ordem de uma parcela a ser calculada ser suficiente para aumentar ou inviabilizar o desempenho da aplicação

Antes de introduzir algumas das metodologias de desenvolvimento de algoritmos paralelos, é necessário fazer algumas considerações e identificar atributos que influenciam o projeto de desenvolvimento de algoritmos paralelos, podendo inclusive gerar uma classificação de tipos de algoritmos paralelos.

No princípio, muitos pesquisadores acharam na área de projeto de desenvolvimento de algoritmos paralelos uma área fascinante e um grande desafio, sem se preocupar entretanto, se seus algoritmos na prática seriam utilizados. O interesse em algoritmos paralelos aumentou, ainda mais, com o aumento da demanda de computadores paralelos de larga-escala, nas últimas décadas. Como resultado, surgiram uma grande variedade de algoritmos que foram projetados para várias arquiteturas diferentes de computadores paralelos.

Kung (em [KUN80]) identifica três atributos, que são: controle de concorrência, módulos de granularidade e geometria de comunicação; os quais são muito importantes em algoritmos paralelos e podem, cada um deles, ser utilizado para classificar os algoritmos paralelos.

Em algoritmos paralelos, devido ao fato de que mais de um processo pode ser executado ao mesmo tempo, o **controle de concorrência** é necessário para garantir que a execução concorrente seja correta. Os algoritmos paralelos podem, então ser classificados quanto ao controle de concorrência, como mostra na figura 3.3. As folhas da árvore representam os vários tipos de controle de concorrência.

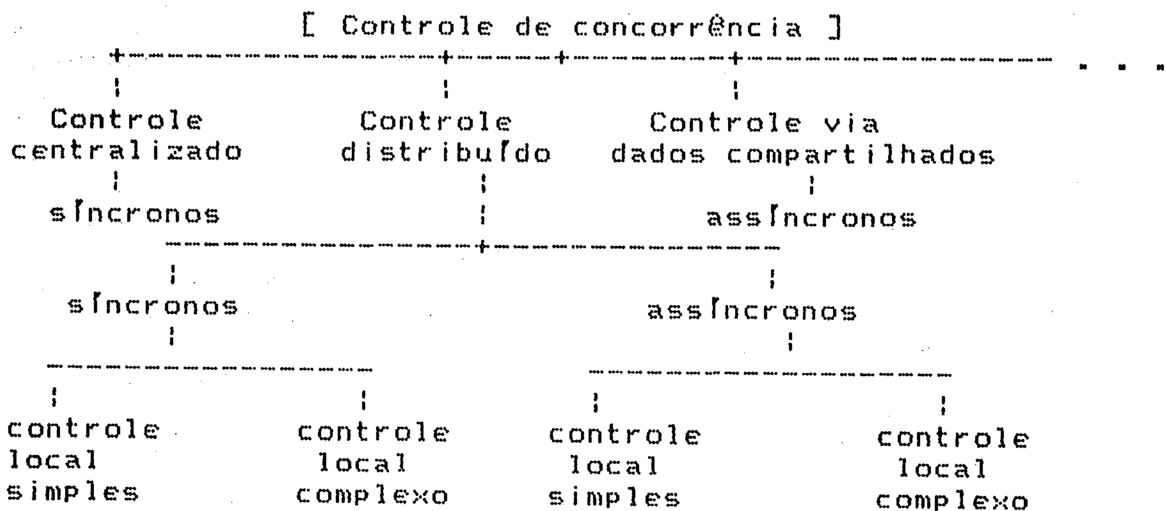


Fig. 3.3 Classificação de controle de concorrência

O módulo de granularidade de algoritmos paralelos refere-se à máxima quantidade de computações um processo típico pode afetar antes de ter que se comunicar com outros processos. O módulo de granularidade reflete se um algoritmo tem ou não tendência de ter uma comunicação intensiva. Eles podem ser classificados em três grupos segundo este atributo, como é mostrado na figura 3.4.

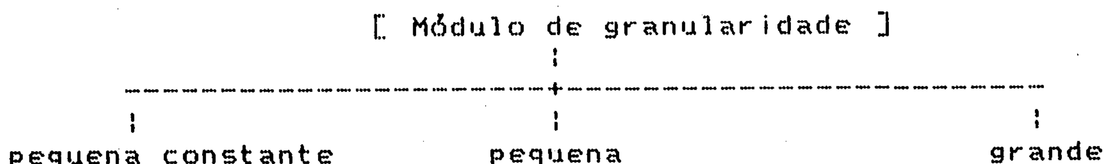


Fig. 3.4 Classificação do módulo de granularidade

Suponha que os processos ou módulos de tarefa de um algoritmo paralelo são conectados para representar a comunicação entre os módulos, então, a configuração da geometria resultante de rede de comunicação é referenciada como geometria de comunicação do algoritmo. Ela pode ser regular ou irregular; de barramento, árvore ou array de uma ou várias dimensões.

Uma vez vistos estes atributos, é necessário perceber que a inovação da computação paralela tem adicionado uma nova dimensão no projeto de algoritmos e programas. As aplicações de paralelismo atingem diversos níveis: algoritmos, programas, comandos e instruções. Programação paralela não é uma simples extensão de programas sequenciais, pois para se explorar as possibilidades oferecidas pelo paralelismo, os projetistas e programadores necessitam "pensar em paralelo", reconsiderar a solução do problema e ter uma forma eficiente de expressar e codificar tal pensamento paralelo (linguagem paralela eficiente).

Alguns algoritmos que são obsoletos para serem implementados em máquinas sequenciais por possuírem alto grau de paralelismo, isto é, conter muitos cálculos que são independentes uns dos outros e que podem ser executados simultaneamente, obtendo um desempenho melhor do que técnicas sequenciais. A dificuldade reside que a nova metodologia de desenvolvimento de algoritmos, a introdução do paralelismo extendido, também requer que seja repensado o conhecimento do sistema, de linguagens paralelas, dos problemas não numéricos e dos métodos numéricos em geral, pois características de computadores e algoritmos paralelos podem ser qualitativamente diferentes das sequenciais.

A relação aplicação-algoritmo-arquitetura ou problema-algoritmo-máquina exerce influência fundamental na exploração do paralelismo e no uso de técnicas de desenvolvimento de algoritmos paralelos. Os principais fatores influenciados são: a representação e descrição do paralelismo; estrutura de dados; mapeamento do algoritmo à arquitetura; esquemas para caracterizar fluxo de dados e de controle e desempenho entre outros.

Segundo Voight [VOI85], as principais abordagens a serem consideradas na obtenção de algoritmos paralelos, em geral são:

a) A análise da complexidade do algoritmo, que tem como objetivo a verificação da viabilidade do algoritmo a ser paralelizado em função do paralelismo apresentado pela arquitetura;

b) A manipulação da ordem de avaliação do problema. O conceito de alteração na ordem de computação pode ser visto como uma reordenação na sequência de operação visando um aumento na quantidade de operações que podem ser realizados em paralelo;

c) O aumento da quantidade de computações assíncronas. O emprego de computações assíncronas permite que não haja a necessidade de sincronizar processadores entre si, evitando que processadores que já efetuaram sua computação fiquem a espera de outros;

d) O emprego da divisão e conquista, onde o particionamento em subproblemas menores de forma a serem tratados independentemente, reduzindo a necessidade de sincronização e/ou comunicação entre elementos processadores pode muitas vezes, reduzir a dificuldade de resolução do problema e a complexidade do problema do ponto de vista computacional.

O problema de considerar esses fatores determinando sequencias de tarefas a serem executados em paralelo e a melhor distribuição de dados e processos em elementos processadores, otimizando ou diminuindo a necessidade de comunicação é denominada de problema de mapeamento.

Duff, em seu artigo que trata da influência da vetorização e paralelização na análise numérica ([DUF87]), apresenta algumas técnicas para o desenvolvimento de algoritmos paralelos, principalmente para a análise numérica.

Estas técnicas estão relacionadas nas quatro abordagens de Voight. A primeira abordagem relaciona-se com a técnica dos métodos explícitos de paralelização, os quais após uma análise da complexidade é procurado explorar características da arquitetura nos métodos, como por exemplo a vetorização, onde instruções vetoriais de hardware são introduzidas nos programas para aumentar a velocidade de processamento. Portanto, operações sequenciais que podem ser convertidas a operações vetoriais semanticamente equivalentes, são substituídas para fazer uso das vantagens de hardware vetorial. A principal abordagem para ganhar velocidade em máquinas vetoriais é a "pipelinezação", que divide operações básicas em suboperações, as quais são efetuadas por um hardware específico da mesma forma que uma linha de produção industrial.

A técnica de divisão e conquista, segundo Duff, é a mais utilizada na análise numérica. Existem muitas áreas onde métodos baseados nesta técnica são muito utilizados, como por exemplo na solução de sistemas de equações tridiagonais e esparsos do tipo banda. Na solução de equações diferenciais parciais estes métodos são conhecidos como de domínio de decomposição; em problemas de estruturas eles são conhecidos como subestruturação. A motivação original para esta técnica não foi o projeto de desenvolvimento de algoritmos paralelos, mas resolver problemas muito grandes e intratáveis diretamente, através da subdivisão em subproblemas menores e tratáveis.

Quando se utiliza estruturas de árvores binárias, uma técnica equivalente a divisão e conquista é a dupla recursividade que também é eficiente no projeto de desenvolvimento de algoritmos paralelos. Nesta técnica a computação é dividida recursivamente em duas subtarefas independentes de mesma complexidade. As subtarefas em cada nível de decomposição recursiva são executadas em paralelo, em processadores independentes. O processamento paralelo inicia no nível mais baixo de subtarefas, o que corresponde ao nível das folhas da árvore binária, onde estão os operandos de entreda.

Outras técnicas citadas por Duff são a de reordenação e técnicas assíncronas. Segundo ele a reordenação é a segunda técnica mais utilizada no desenvolvimento de algoritmos paralelos para análise numérica. Estas técnicas correspondem aos itens de manipulação da ordem de avaliação e do aumento da quantidade de computações assíncronas.

4 NOÇÕES DA ANÁLISE DE ALGORITMOS

Traub (em [TRA76a]) apresenta algumas razões para se estudar análise de algoritmos. Apesar de não ser uma lista completa, ele considera as seguintes razões:

a) A seleção de algoritmos eficientes é um objetivo central na Ciência da Computação e na Matemática Aplicada. A seleção de algoritmos é um problema de otimização multidimensional, onde uma destas dimensões é a complexidade de algoritmos;

b) Há muita literatura contendo papers dando as condições de convergência de processos infinitos. Ser convergente não é suficiente para ser computacionalmente interessante, tem-se que saber o limite do seu custo. Um dos objetivos da complexidade computacional é que condições adicionais precisam ser impostas ao problema tal que o custo de sua solução possa ser "a priori" limitado;

c) Resultados da complexidade ajudam a dar a estrutura do arquivo;

d) Limites inferiores da complexidade do problema nos dão uma hierarquia natural baseada nas dificuldades intrínsecas do problema;

e) Complexidade leva a uma Teoria matematicamente interessante e satisfatória.

Por outro lado, dizer que um problema é algoritmicamente solucionável significa, informalmente, que existe um algoritmo, que para qualquer entrada, se lhe der todo TEMPO e ESPAÇO desejado, ele produz a resposta correta.

Um dos conceitos mais importantes é o de algoritmo. A palavra **algoritmo** vem de Abu al Khwarizmi, que segundo a história, foi o primeiro a fazer um algoritmo, isto cerca de 800 DC. Este conceito foi formalizado antes de 1930, antes de surgir o primeiro computador. É uma descrição passo a passo de como o problema é solucionável. A descrição deve ser finita e os passos bem definidos, sem ambiguidade.

O objetivo da análise de algoritmos é dado um algoritmo, melhorá-lo ou escolher o melhor de entre algoritmos segundo os critérios de correção; quantidade de trabalho; quantidade de espaço disponível; simplicidade e otimalidade.

Correção é se o algoritmo está certo! Isto exige conhecimentos na área sobre a qual se está fazendo o algoritmo. A quantidade de trabalho pode ser medida pela complexidade de tempo, onde é estudado o pior caso, o melhor caso e o caso médio. Este é o critério em que se concentram mais os estudos.

A quantidade de espaço disponível determina a complexidade de espaço e Otimalidade, envolve o conceito do algoritmo ótimo, o qual deve gastar menos tempo, tanto no pior caso, como nos demais casos.

Para se determinar o algoritmo ótimo, pega-se o algoritmo candidato, juntamente com uma função W e para uma entrada de tamanho n, o algoritmo irá fazer no máximo W(n) operações básicas.

Para uma função f, prova-se que, para todo algoritmo da classe em consideração, existe uma entrada de tamanho n, para a qual o algoritmo faça pelo menos f(n) operações (f(n) é o limite mínimo de operações) e se a função W for igual à função f, então o algoritmo é ótimo. Caso W não seja igual a f, pode ocorrer que:

a) f pode estar errado, ou seja, f(n) é menor que o número necessário para resolver o problema;

b) O algoritmo em questão não é ótimo! W(n) > f(n).

4.1 Medidas de complexidade e cotas

Uma maneira para comparar o desempenho de algoritmos é pelo computar de alguma de suas medidas de eficiência. Para ser útil, esta medida necessita ser independente de máquina. Bons algoritmos tendem a permanecer bons, mesmo que implementados em diferentes linguagens de programação ou se executados em diferentes máquinas.

As duas medidas mais úteis são o tempo requerido para executar o algoritmo e a memória necessária pelo algoritmo. Estas medidas são geralmente expressas em função do tamanho do problema, também denominado de instância do problema. Na quantificação destas medidas o modelo de computação é importante porque a quantificação do tempo é feita pela contagem de operações e a forma ou o problema de representação afeta no tamanho do problema.

Destas duas medidas resultam a complexidade de tempo, que é uma função do tamanho da instância que expressa o tempo máximo que o algoritmo necessita para resolver aquela instância, e a complexidade de espaço, que é uma função do tamanho da instância que expressa o pico da quantidade de espaço de memória requerido para resolver aquela instância.

O tempo da função de complexidade é o tempo dos requerimentos computacionais, os quais na prática são calculados como o número de vezes que uma operação particular ocorre.

Em algumas análises utiliza-se o **critério do custo uniforme**, no qual todas as operações elementares (comparação, soma, multiplicação,troca, etc ...) gastam uma unidade de tempo. Em outras análises emprega-se o **critério do custo**

logarítmico, no qual cada operação gasta o tempo proporcional ao número de símbolos necessários para representar os operandos. Neste caso o tamanho de entrada n é dado pelo número de bits requeridos para decodificar a instância do problema.

No comando **for i:=1 to n do** xi:=xi+1; temos que o tempo é da ordem n, uma vez que n adições são efetuadas. Já neste outro comando: **for i:=1 to n do for j:=1 to n do** xij:=2*xij; ocorrem nxn (n ao quadrado - n^2) multiplicações, sendo da ordem n^2.

Portanto, complexidade de tempo, é uma função que relaciona o tamanho de uma instância ao tempo necessário para resolvê-lo. O tamanho da instância é a medida da quantidade dos dados de entrada, no caso de um sistema de equações é da ordem do sistema ou da matriz dos coeficientes. Uma instância de um problema é obtida ao se fixarem valores particulares de todos os parâmetros do problema.

Para exemplificar, será considerado o problema de ordenar uma lista finita de n números inteiros. Os parâmetros a serem considerados são os n números inteiros <M1,M2,...,Mn>, e a solução consiste nesses números ordenados em ordem crescente (exemplo retirado de [TER91]).

ALGORITMO 4.1

```
   Ordlist - Ordenação de lista na ordem não decrescente;
Entrada:   A lista <M1,M2,...,Mn> de n >= 2 números inteiros;
Saída:     A lista ordenada em ordem não decrescente:
           <Mp(1),Mp(2),...,Mp(n)> tal que Mp(i) <= Mp(i+1) (∀i)
1 Para j:=n-1 ate 1 faça
2      Para i:=1 ate j faça
3           Se Mi  > Mi+1
4              Então "Troque os valores de Mi e Mi+1 entre si";
5           Fim-se;
6      Fim-para;
7 Fim-para;
8 Pare com saída <M1,M2,...,Mn>.
```

Para ilustrar, consideremos uma instância deste problema, que pode ser a lista <5, 47, 0, -9> para a qual n=4 e a solução é <-9,0,5,47>. A execução do algoritmo 4.1, com esta entrada é mostrada na figura 4.1, onde se tem os valores de i, j, Mi, Mi+1 na linha 3 e a lista resultante na linha 6.

j	i	Mi	Mi+1	Lista resultante (em 6)
3	1	5	47	<5, 47, 0, -9>
3	2	47	0	<5, 0, 47, -9>
3	3	-9	47	<5, 0, -9, 47>
2	1	5	0	<0, 5, -9, 47>
2	2	-9	5	<0, -9, 5, 47>
1	1	0	-9	<-9, 0, 5, 47>

Fig. 4.1 - Execução do algoritmo 4.1

Foram feitas seis comparações na linha 3 do algoritmo 4.1. De uma forma geral, verifica-se que são feitas $n(n-1)/2$ comparações e no máximo este número de trocas. A seguir é mostrado a forma do cálculo:

j= n-1 a 1, para cada j se tem i variando de 1 a j, ou seja,

j = n-1	===> i = 1,...,n-1	n-1 comparações
j = n-2	===> i = 1,...,n-2	n-2 comparações
	. . .	
j = 2	===> i = 1, 2	2 comparações
j = 1	===> i = 1	1 comparação

Portanto o número de comparações é: $1 + 2 + \ldots + (n-2)+(n-1)$, que é uma progressão aritmética com n-1 parcelas, cuja a soma é $n(n-1)/2$.

O espaço da função de complexidade é o espaço dos requerimentos computacionais do algoritmo, que podem ser para armazenar matrizes, vetores, dados intermediários e etc...

O tempo do algoritmo e o espaço de armazenagem são importantes medidas e são particularmente apropriadas se o algoritmo for implementado e executado. O tempo de um algoritmo é um fator que restringe o tamanho do problema que pode ser resolvido no computador e a profundidade do programa, no sentido de medir a simplicidade de um algoritmo.

Na tentativa de responder questões, como: Qual é o melhor algoritmo para resolver um problema específico; Qual é o número mínimo de operações elementares necessárias para resolver tal problema e se é um problema intratável, ou seja, que não existe nenhum algoritmo que o resolva em um tempo razoável, chegou-se a dois importantes resultados:

a) A possibilidade de se dividir todos os problemas para os quais existem algoritmos em duas classes distintas: algoritmos eficientes ou praticamente solúveis e algoritmos ineficientes os quais possivelmente refletem a pesquisa por força bruta ou aqueles cujo tempo de execução cresce como funções exponenciais e superexponenciais;

b) A definição do algoritmo ótimo, o qual combina o tempo real de solução do problema com o tempo mínimo exigido para se resolver o problema. Estas idéias são formalizadas nas cotas.

Para um dado problema, a **cota superior de complexidade Cs(n)** é a menor das complexidades dos algoritmos conhecidos para resolver o problema. Para a instância de tamanho n, resolve-se o problema em tempo Cs(n). É um valor que depende do algoritmo.

A cota superior depende do atual estado do conhecimento, pois não interessa utilizar um algoritmo que tenha uma cota superior a Cs(n), uma vez que se conhece o algoritmo que resolve o problema num tempo igual a Cs(n). Por esta razão, algoritmos novos são construídos e analisados na espereança de se reduzir a cota superior. Portanto, novos algoritmos são analisados para verificar se tem desempenho, no pior caso, melhor que qualquer outro algoritmo conhecido.

Aprimorar uma cota superior significa descobrir um algoritmo que tenha uma rapidez máxima menor do que a dos outros.

Existem dois métodos principais para determinar a cota superior pela análise do desempenho pessimista do algoritmo: a contagem das instruções básicas e a resolução de equações de recorrência. Ambos os métodos requerem a identificação do pior caso, isto é, uma entrada de tamenho n que maximixe a quantidade de trabalho que o algoritmo necessita fazer para resolver o problema.

A cota inferior de complexidade Ci(n) de um problema é a complexidade inerente ao problema. Ela determina que nenhum algoritmo pode resolver o problema com complexidade, no pior caso, em tempo menor do que Ci(n), para entradas arbitrárias de tamanho n. No caso de ordenação de uma lista de n números inteiros, a cota inferior de complexidade é n log n.

A determinação da cota inferior é um dos problemas mais difíceis, nos limites de complexidade, pois não há algoritmo para análise; há poucos princípios gerais para serem aplicados e os resultados variam de problema para problema. Mesmo assim, algumas técnicas e princípios têm-se mostrado úteis.

Uma forma simples para o cálculo de um limite inferior é conhecido como limite inferior trivial. O método consiste simplesmente da contagem do número de entradas que precisa ser examinado e do número de saídas que precisam ser produzidos. Por exemplo, para a multiplicação de duas matrizes nxn, são necessários que n^2 elementos de saída sejam produzidos e, portanto, a cota inferior trivial é da ordem mínima de n^2. Isto nada diz sobre o número de multiplicações ou adições necessários para resolver o problema, mas somente que a operação - elemento de saída, necessita ser computada n^2 vezes.

Um dos princípios mais úteis sobre a quantidade de trabalho mínima na determinação de cotas inferiores é que o resultado de uma comparação entre dois itens contém no máximo um bit de informação, então se existem m entradas e o algoritmo se propõe a identificar uma através de comparações, só serão necessárias ! log m ! comparações. Este princípio serve de guia na prova de cotas inferiores para os problemas de comparação, como no problema de ordenação de conjuntos, cuja cota inferior é Ci(n)= @(n log n).

Uma das maneiras mais elegantes de provar uma cota infeior de um problema P1 é mostrar que um algoritmo para resolver P1, admite com uma transformação sobre uma instância, pode ser usado para construir um algoritmo para resolver outro problema P2, para o qual a cota inferior é conhecida. Exemplo disto é a redução polinomial, usada para demonstrar a equivalência de problemas NP-completos (como será visto no próximo capítulo).

Aparentemente existem duas funções de complexidade de problemas, as cotas superiores e inferiores. O objetivo final é fazer estas duas cotas, estas duas funções coincidirem. Quando isto ocorre, o algoritmo é ótimo, tendo Ci(n) = Cs(n). Entretanto para a maioria dos problemas se tem:

Ci(n) <= Cs(n).

Ao se tentar demonstrar uma cota melhor do que a existente, necessita-se concentrar em técnicas que permitam aumentar a precisão com a qual o mínimo, sobre todos os algoritmos possíveis, pode ser limitado. Entre estas técnicas, surgiram conceitos de processamento paralelo e procedimento vetorial, os quais tendem a modificar os conceitos de cota inferior e superior.

Estudos devem ser desenvolvidos para determinar a relação entre a cota inferior, que é inerente ao problema sequencial e a sua extensão paralela.

4.2 Análise do caso pessimista e do caso médio

Complexidade pessimista tem sido a de maior interesse teórico. Algumas vezes esta complexidade pessimista pode desqualificar um algoritmo que tenha uma complexidade média aceitável, uma vez que exista, no pior caso, a possibilidade de ocorrência de desastre, como no caso de controle de trens e reatores atômicos.

A identificação do pior caso quando um algoritmo realiza a mesma quantidade de trabalho para qualquer entrada de tamanho n é fácil, pois significa que o fluxo de controle é independente dos dados. Qualquer caso é o pior caso! Exemplos são a procura do maior elemento de um conjunto de n elementos, onde são necessárias n-1 comparações, ou a multiplicação de duas matrizes nxn na forma clássica, onde são necessárias n^3 multiplicações.

Devido à utilidade prática, a complexidade média tem sido mais estudada. Um exemplo disto é na programação linear, o algoritmo simplex, que necessita de um tempo igual a uma função exponencial no tamanho da instância, no caso pessimista, mas no caso médio é extremamente rápido.

Outro exemplo é o algoritmo 4.2, conhecido como Quicksort ou ordenação por concatenação. O algoritmo Quicksort é provavelmente o melhor algoritmo prático para classificação conhecido, principalmente porque no caso médio, tem uma complexidade o(n log n) comparações, desde que se garanta que a cada passo de partição seja subdividido em sublistas de tamanhos aproximadamente iguais. Devem ocorrer cerca de log n estágios de partições, cada uma levará cerca de o(n). O problema é que no pior caso, o desempenho é da ordem n^2, portanto não é ótimo, uma vez que a cota inferior de ordenação é de ϑ(n log n) e se conhece algoritmos desta ordem.

ALGORITMO 4.2

```
   Quicksort(S) Ordenação por concatenação

1 Inicio
2   Se |S| <= 1  então retorne (S);
3   Escolha um elemento x de S;
4   Particionar S em três conjunto:
     S1 = Todos elementos menores que x;
     S2 = Todos elementos iguais a x;
     S3 = Todos elementos maiores que x;
5   Retorne (Quicksort(S1);S2;Quicksort(S3));
6 fim.
```

Se todos os elementos de S são distintos e o algoritmo por falta de sorte, escolher sempre um x, o qual é o menor elemento do conjunto S a cada estágio, então S1 é vazio, S2 contém um elemento (o x) e S3 contém um elemento a menos do que S. Neste caso serão necessário n estágios de particionamento, onde cada k-ésimo estágio examinará cerca de n-k elementos, sendo a complexidade da ordem de n^2.

4.3 Ordem máxima, mínima e exata

Para o cálculo de complexidade, pode-se medir o número de passos de execução num modelo matemático ou medir o número de segundos num computador específico. Estes são dois extremos, onde os resultados podem diferir por mais do que um simples fator de escala. Ao se escolher um modelo de computação, tem-se que equilibrar o realismo com a tratabilidade matemática.

Ao invés do cálculo exato do tempo de execução em máquinas específicas, a maioria das análises leva em conta o número das operações elementares. A medida de complexidade é então o crescimento assintótico desta contagem de operações.

Na matemática, para expressar o fato de que uma função g(n) não cresce, assintoticamente, mais rapidamente do que uma outra função f(n), utiliza-se o conceito de **ordem máxima**, ou

seja, g(n) = o(f(n)). Ordem f(n), pode ser interpretado como o conjunto das funções com a propriedade de que, para qualquer função g(n) em o(f(n)), existe uma constante c tal que

$$g(n) <= c\ f(n).$$

De forma análoga, @(f(n)) **(ordem mínima f(n))** é o conjunto das funções com a propriedade de que, para qualquer função g(n) em @(f(n)), existe uma constante c>0 tal que

$$g(n) >= c\ f(n).$$

Por fim, o(f(n)) (ordem exata f(n)) é o conjunto das funções com a propriedade de que, para qualquer função g(n) em o(f(n)), existem constantes c1 > 0 e c2 tais que

$$c1\ f(n) <= g(n) <= c2\ f(n).$$

A notação de ordem máxima (o( )) é usada para descrever cotas superiores; a notação de ordem mínima (@( )) é usada para cotas inferiores e a notação de ordem exata (o( )) é utilizada quando é possível caracterizar a complexidade com um fator constante.

4.4 Distinções entre funções de complexidade

Quando se estuda a análise de algoritmos, especilamente quando se compara algoritmos, ouve-se falar em problemas de ordem n, ordem n log n, n ao quadrado, ordem polinomial ou exponencial, sem que se perceba o que significa o crescimento da função de complexidade em relação ao aumento da instância do problema.

Neste item, pretende-se fazer uma distinção entre as funções de complexidade, de forma a se perceber as implicações no tempo de processamento de cada algoritmo destas ordens típicas de complexidade. Para isto a figura 4.2, apresenta os gráficos das ordens: n; n log n; n^2; n^3; 2^n e 3^n.

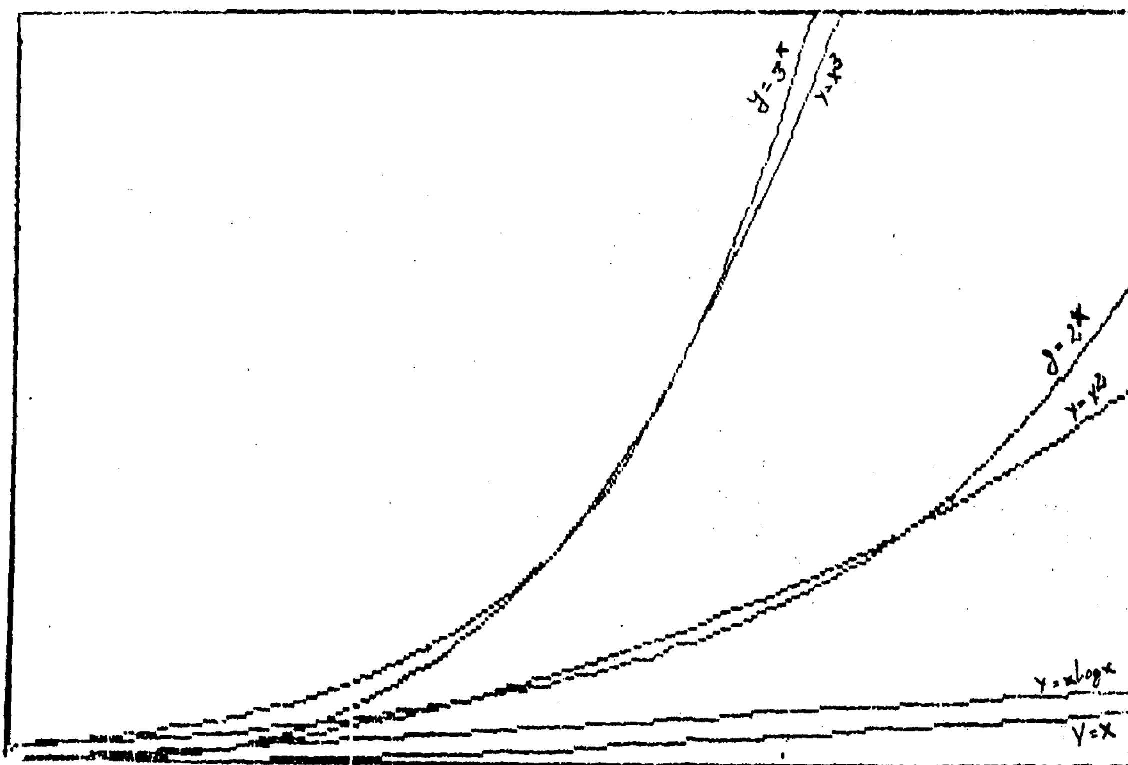


Fig. 4.2 Gráfico das funções típicas de complexidade

Observando o gráfico da figura 4.2, vê-se que para instâncias pequenas ($n<5$) não se chega a diferenciar o crescimento de uma função polinomial (n^2, n^3) por uma exponencial (2^n, 3^n). Pode-se até se ter uma idéia errônea uma vez que, para n entre 2 e 4, as funções n^2 e 2^n desempenham um comportamento ($n^2 >= 2^n$) contrário do resto do domínio ($n^2 <= 2^n$).

n	n. log n	n^2	2^n	n^3	3^n
1	0,000	1	2	1	3
2	2,000	4	4	8	9
3	4,754	9	8	27	27
4	8,000	16	16	64	81
5	11,609	25	32	125	243
6	15,509	36	64	216	729
7	19,651	49	128	343	2.187
8	24,000	64	256	512	6.561
9	28,529	81	512	729	19.683
10	33,219	100	1.024	1.000	59.049
15	58,603	225	32.768	3.375	14.348.907
20	86,438	400	1.048.576	8.000	3.486.784.401

Fig. 4.3 Comparação das funções típicas de complexidade

Na figura 4.4 é ilustrado as diferenças em rapidez de crescimento das funções típicas de complexidade. Os valores expressam os tempos de execução quando uma operação elementar no algoritmo é executável em um décimo de microsegundo (0,00001). Mais uma vez pode ser observado o crescimento extremamente rápido das funções de complexidade exponencial.

função de complexidade	tempo em segundos (Uma oper.elem. 10^-5)			
	n = 20	n = 40	n = 60	n = 100
n	0,0002	0,0004	0,0006	0,0010
n log n	0,0009	0,0021	0,0035	0,0066
n^2	0,0040	0,0160	0,0360	0,1000
n^3	0,0800	0,6400	2,1600	10,0000
2^n	10 segundos	127 dias	3656 séculos	4.10^15 séculos
3^n	581 minutos	38551 séculos	1,3.10^14 séculos	1,6.10^33 séculos

Fig. 4.4 Comparação de funções de complexidade ([TER91])

Devido ao desempenho negativo dos algoritmos de complexidade exponencial, conforme ilustrado nas figuras anteriores, tais algoritmos são considerados inaceitáveis, ineficientes, enquanto que os de complexidade polinomial são considerados aceitáveis.

Como as complexidades são em geral caracterizadas em termos de ordem (usando a notação o( )), as constantes multiplicativas das funções de complexidade não são explicitas. Entretanto é importante ressaltar que um algoritmo de complexidade rapidamente crescente pode ter uma constante multiplicativa menor do que um outro de complexidade lentamente crescente. Neste caso, o primeiro algoritmo pode ter um desempenho superior para instâncias pequenas. Isto pode ser observado na figura 2.3, se comparado as funções de complexidade de ordem $o(n^2)$ e $o(2^n)$, onde nas instâncias pequenas $2^n < n^2$, sendo que para $n>5$, já temos o real comportamento das ordens, ou seja $2^n > n^2$.

4.5 Tamanho de instâncias de problemas

O tamanho de instância é um conceito que depende da natureza do problema. Ele pode ser exato se tomarmos n como sendo o número de símbolos necessários para codificar a instância num determinado modelo computacional.

É necessário definir explicitamente o tamanho de instâncias em cada caso, para que o resultado da análise tenha significado. Nos problemas de ordenação, n é o número de objetos a serem ordenados; para problemas de grafos, n pode ser o número de vértices. Algumas vezes, pode ser mais conveniente descrever o tamanho do problema através de dois ou mais parâmetros, como no caso de grafos onde podemos ter n como o número de vértice e m o número de arestas.

4.6 Questões importantes de complexidade paralela

Neste item são enumerados algumas questões importantes quanto a complexidade paralela, que foram levantadas pelo professor Terada durante a VII Escola de Computação, realizada em São Paulo no ano de 1990.

a) Quais os fatores que determinam a complexidade de algoritmos paralelos?;

b) Que tipo de problemas computacionais são ou não são solucionáveis eficientemente em paralelo?;

c) Os algoritmos paralelos são completamente diferentes dos melhores algoritmos sequencias para o mesmo problema?;

d) Quais são as cotas superiores mínimas e as cotas inferiores máximas para complexidade paralela de tempo e de espaço, de problemas básicos, como: ordenação, operações aritméticas, operações sobre matrizes, busca em grafos, etc...;

e) É possível caracterizar problemas inerentemente sequenciais?

O ensaio da resposta destas peguntas encontram-se no livro do professor Terada da Escola de Computação. Aqui, nesta introdução foram feitas algumas considerações, as quais são aprofundadas no desenvolvimento deste trabalho.

Entre os fatores que determinam a complexidade de algoritmos paralelos, pode-se identificar o número de processadores, a arquitetura da máquina (ou modelo computacional) e a quantidade de trabalho para resolução do problema.

Outra questão é diferenciar a medida teórica e a medida prática de tempo de processamento. A medida teórica, conta o número de passos para a resolução do problema. A medida prática leva em conta o número de processadores, a divisão do problema em tarefas que podem ser executadas em paralelo, o tempo de espera dos processadores e o tempo de gerenciamento do sincronismo dos processos.

Referente ao número de processadores, necessita-se verificar qual a relação do número de processadores disponíveis e a complexidade da resolução de um problema nesta máquina. E ainda, como determinar o número ideal de processadores em uma determinada arquitetura, para se resolver um dado problema.

Por fim, sabe-se que o tempo e o espaço são os principais fatores que são analisados na complexidade de algoritmos sequenciais; considerações sobre a importância deles na complexidade de algoritmos paralelos e como eles são caracterizados, devem ser feitas, para um bom entendimento da complexidade paralela.

4.7 Complexidade de algoritmos paralelos

Não se pode esquecer que análise de algoritmos se refere aos processos de determinar quão bom é um algoritmo, quão rápido, quanto custa executá-lo e quão eficiente é seu uso dos recursos disponíveis.

Os principais recursos sobre os quais o desempenho de algoritmos sequenciais são medidos são o tempo e o espaço. Seguindo uma analogia a estes recursos para avaliação do desempenho de algoritmos paralelos, pode-se verificar que o tempo permanece como o principal recurso no processamento paralelo, só que agora ele depende não somente da complexidade

das operações computacionais, mas também da complexidade do gerenciamento das operações, criadas pela comunicação, sincronização e restrições de acesso aos dados. O desempenho medido quanto ao espaço, depende da quantidade de memória que o algoritmo necessita.

Em computadores paralelos, este fator memória é de menor importância, pois o tamanho do hardware, que é o número de elementos de uma máquina paralela que são ativados durante a computação representa o recurso mais importante nas considerações do desempenho do programa. Numa máquina matricial (matriz de processadores) diz respeito ao número de processadores envolvidos na computação; para uma máquina vetorial é a soma dos comprimentos dos vetores.

Quando um novo algoritmo está sendo projetado, é usual avaliá-lo segundo alguns critérios como: tempo de execução, número de processadores usados e custo. Juntamente com estas medidas padrões, um número de outras medidas de tecnologia são algumas vezes utilizadas quando se sabe que um algoritmo será executado em um computador com uma tecnologia particular.

4.7.1 Tempo de execução

Uma vez que velocidade de computação aparece como sendo a principal razão, a principal questão na construção de computadores paralelos, a medida mais importante na avaliação de algoritmos paralelos é portanto o tempo de execução. Ele é definido como o tempo absorvido por um algoritmo para resolver um problema em um computador paralelo, isto é, o tempo desde o momento que o algoritmo inicia até o momento que ele termina. Se os vários processadores não iniciam e nem terminam juntos, então o tempo de execução é igual ao intervalo de tempo entre o momento do primeiro processo iniciar a computação até que o último processo termine a computação.

Kung (em [KUN76]) definiu o tempo absorvido por um programa paralelo, como o intervalo de tempo do processo em um programa que termina por último, onde o intervalo de tempo do processo é calculado como a soma de três quantidades:

a) Tempo de processamento básico, o qual é a soma dos tempos absorvidos pelos seus estágios;

b) Tempo bloqueado, o qual é o tempo total que o programa é bloqueado até o final do estágio devido a espera de dados na sincronização do algoritmo ou de espera para entrada numa seção crítica de um algoritmo assincrono;

c) Tempo de execução do gerenciamento da sincronização, a qual é feita através de operações primitivas de sincronização e de implementação e acesso a seções críticas.

Em processamento paralelo, o objetivo do menor tempo de execução não é, necessariamente, sinônimo de realizar o menor número de operações aritméticas como em processamento sequencial, pois pode haver um número maior de operações, as quais podem ser executadas simultaneamente. Em alguns casos, uma medida útil de desempenho para processamento paralelo, pode ser o parâmetro o qual é inversamente proporcional ao tempo de CPU, isto é, o tempo durante o qual unidades do computador (aritméticas e lógicas) são utilizadas por um programa, estão engajadas durante a execução do programa.

Contando passos

Antes de implementar um algoritmo (sequencial ou paralelo) em um computador, é bom fazer uma análise teórica do tempo necessário para resolver o problema computacional em questão. Isto é geralmente feito pela contagem do número de operações elementares ou passos executados pelo algoritmo no caso pessimista. Isto leva a uma expressão que descreve o número de passos como uma função do tamanho da entrada.

O tempo de execução de um algoritmo paralelo é geralmente obtido pela contagem de dois tipos de passos: passos computacionais e passos de envio. Um passo computacional consiste de uma operação aritmética ou lógica realizado sobre um dado por um processador. Por outro lado, em um passo de envio um dado vai de um processador para outro através da memória compartilhada ou através da rede de comunicação. Para um problema de tamanho n, o pior caso paralelo de tempo de execução de um algoritmo, é uma função de n, denotado por t(n). Estritamente falando o tempo de execução é também uma função do número de processadores.

Geralmente, passos computacionais e passos de envio não requerem necessariamente, o mesmo número de unidades de tempo. Um passo de envio depende da distância entre os processadores.

Limites inferiores e superiores

Dado um problema computacional para o qual um novo algoritmo sequencial tenha sido projetado, é comum entre projetistas de algoritmos questionar se o algoritmo é o mais rápido possível para problemas e, caso não o seja, como compará-lo com os demais algoritmos já existentes para o problema.

O primeiro questionamento é respondido geralmente comparando o número de passos executados pelo algoritmo com a cota inferior, que é o limite mínimo de operações ou passos requeridos para resolver o problema no pior caso.

Sabe-se, por exemplo, que para computar o produto de duas matrizes nxn, a matriz resultante tem n^2 entradas, para isto, muitos passos são necessários para cada algoritmo de multiplicação de matrizes produzir o resultado.

Já para o problema de classificação, é definido sobre um conjunto de n números dados em ordem randômica; para classificá-los em ordem não decrescente, existem n! possíveis permutações da entrada e, (log n)! bits são necessários para os distinguir entre si. Portanto no pior caso, qualquer algoritmo para classificação necessita a ordem n log n passos para produzir a solução.

A mesma idéia geral de cotas se aplica a algoritmos paralelos, mas se tem que adicionar dois fatores: tem-se que considerar o modelo de computação paralela e o número de processadores envolvidos.

Speedup

Na avaliação de algoritmos paralelos para um dado problema é bastante natural se ter uma comparação em termos do melhor algoritmo sequencial disponível. Então uma boa indicação da qualidade de um algoritmo paralelo é o **speedup**, que é definido como o quociente do tempo de execução do mais rápido algoritmo sequencial para o problema, no pior caso, pelo tempo do algoritmo paralelo, também no pior caso.

Suponha que se conheça um algoritmo sequencial para um dado problema que seja o mais rápido possível. Idealisticamente, espera-se ter o melhor speedup de N, quando se resolve tal problema utilizando N processadores operando em paralelo. Na prática tal speedup não é obtido para o problema, pois não é sempre possível decompor o problema em N processos, onde cada um requeira 1/N do tempo necessário por um processador resolver o problema original e na maioria dos casos a estrutura do computador paralelo utilizada para resolver um problema, geralmente impõe restrições que faz com que o tempo de execução desejado seja inalcansável.

4.7.2 Número de processadores

O segundo critério mais importante na avaliação de algoritmos paralelos é o número de processadores que são requeridos para resolver um problema. Quando vários processadores estão presentes, o problema da manutenção, em particular, é complexo e o preço pago para garantir o alto grau de realidade cresce facilmente. Entretanto, o grande número de processadores que um algoritmo usa para resolver um problema, torna a solução mais dispendiosa de ser obtida. Para um problema de tamanho n, o número de processadores requeridos por um algoritmo é uma função de n e será anotado por p(n).

O tamanho do hardware, o número de elementos da máquina paralela os quais estão ativos durante o processamento, representa o principal recurso a ser tomado nas considerações de desempenho do programa. Exemplos são: em processadores matriciais, o número de processadores envolvidos na computação; em máquina vetorial, a soma dos tamanhos dos vetores.

4.7.3 Custo

O custo de um algoritmo paralelo é definido como o produto de duas quantidades, ou seja, o produto do tempo de execução paralelo pelo número de processadores utilizados. Isto significa que o custo é igual ao número de passos executados coletivamente por todos processadores na solução de um problema no pior caso. Esta definição assume que todos processadores executam o mesmo número de passos. Para um problema de tamanho n, o custo de um algoritmo paralelo é uma função de n, anotado por c(n), então se tem:

$$c(n) = p(n) \cdot t(n)$$

Assume-se que o limite inferior é conhecido como o número de operações sequenciais requerido, no pior caso, para resolver o problema. Se o custo de um algoritmo paralelo para o problema atingir o limite inferior com um fator de uma constante multiplicativa, então o algoritmo é dito ser de custo ótimo. Isto é porque qualquer algoritmo paralelo pode ser simulado por um algoritmo sequencial. Se o número total de passos executados durante uma simulação sequencial for igual ao limite inferior, então isto significa que, em termos de custo, este algoritmo paralelo não pode proporcionar com sua execução o menor número de passos possíveis. Isto pode ser possível se para reduzir o tempo de execução do custo ótimo do algoritmo paralelo for utilizado mais processadores.

Um algoritmo paralelo não tem custo ótimo se existe um algoritmo sequencial que possui tempo de execução menor que o custo do algoritmo paralelo.

Uma implementação paralela de um algoritmo para resolver um problema de tamanho n é dito consistente se o número de operações elementares requeridos por esta implementação for da mesma ordem de magnitude da função que expressa a quantidade necessária para sua implementação em um computador sequencial.

4.7.4 Outras medidas

A tecnologia da VLSI (Very Large Scale of Integration) tem sido usada para garantir sucessos em processamento paralelo. Quando se avalia algoritmos paralelos por VLSI, os critérios de área de processador, largura e período do circuito tem sido utilizados.

Área. Se vários processadores são postos para compartilhar o estado real em um chip, a área necessária pelos processadores e tamanho dos fios das conecções entre eles, assim como, a geometria de interconexão determina quantos processadores um chip pode ter. Se dois algoritmos gastam o mesmo tempo para resolver um problema, mas um ocupa menos espaço (área) quando implementados como um circuíto VLSI, este geralmente é preferido.

Largura. Isto se refere a largura do tamanho dos fios das conecções do processador em uma dada arquitetura. Se o tamanho tem uma largura constante, então significa que a arquitetura é regular (tem um modelo que se repete) e modular (pode ser construído uma e repetido em módulos).

Período. Assume-se que vários conjuntos de entrada estão disponíveis e aguardam para serem processados por um circuito em pipeline. Seja A1, A2, ...,An a sequência de entradas tais que o tempo de processamento é o mesmo para todo Ai. O período do circuito é o tempo entre os momentos de processamento de Ai e Ai+1. Evidentemente, períodos pequenos são propriedades desejáveis em algoritmos paralelos.

Um dos objetivos do processamento paralelo é projetar algoritmos paralelos com desempenho que possam ser relacionados com algoritmos sequenciais ótimos. Teoricamente se tem que, um algoritmo paralelo que resolve um dado problema de tamanho n com p(n) processadores paralelos no tempo t(n), tem um rendimento correspondente a um algoritmo sequencial o qual resolve o mesmo problema em um computador sequencial equivalente, em tempo p(n).t(n). Na verdade o tempo p(n)t(n) é tomado como limite superior na complexidade de tempo do melhor algoritmo sequencial conhecido para resolver o problema.

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

5 CLASSES DE COMPLEXIDADE DE TEMPO

5.1 Classes de complexidade sequencial

Os problemas computacionais costumam ser classificados de acordo com suas cotas superiores sequenciais em quatro classes de complexidade, a saber:

a) Problemas que podem ser resolvidos por algoritmos de rapidez pessimista linear, isto é, o(n), se n é o tamanho das instâncias;

b) Problemas que podem ser resolvidos por algoritmos de rapidez pessimista não-linear, mas polinomial, isto é, o(p(n)), onde p(.) é um polinômio de grau maior do que um. Esta classe é denominada **P**, uma abreviação de polinomial;

c) Problemas que aparentemente não possuem complexidade intrínseca (isto é, cota inferior) exponencial, mas para os quais não se conhecem algoritmos de rapidez pessimista polinomial. Esta classe é conhecida como **NP**;

d) Problemas que intrinsecamente requerem, para a sua solução um número de operações igual a uma função exponencial no tamanho das instâncias, porque eles exigem a geração de um número exponencial de subproblemas.

A classificação de problemas nas três principais classes depende do "estado da arte", pois a qualquer momento podem surgir algoritmos que reduzem de classe os problemas, como o problema de se verificar a planaridade de um grafo, que com a descoberta de um novo algoritmo em 1971, o reduziu da classe polinomial para a linear.

Para problemas da classe exponencial, não há esperanças de que novos algoritmos sejam descobertos e possam vir a deslocar tais problemas para uma das duas primeiras classes, pois eles são intrinsicamente exponenciais no tamanho das instâncias.

Um problema de tempo polinomial é um problema que pode ser resolvido por um algoritmo cuja complexidade de tempo é da classe (b), ou seja polinomial.

Um problema que só pode ser resolvido por algoritmo cuja complexidade de tempo não pode ser limitada por um polinômio é chamado de problemas de tempo exponencial, da classe (d). Problemas onde não se consegue algoritmo de tempo polinomial que seja capaz de o solucionar, são denominados de problemas **intratáveis**. Portanto um problema inerentemente intratável é um problema onde se é incapaz de encontrar um algoritmo determinístico que o solucione em tempo polinomial.

Problemas intratáveis ou aparentemente intratáveis são difíceis de entendimento de suas propriedades cruciais, particularmente no ambiente usual da computação onde a notação de

algoritmo pressupõe que o resultado de cada operação seja unicamente definido. Estes algoritmos são chamados de **determinísticos** e eles proporcionam a maneira como os programas serão executados no computador.

Se um algoritmo admite conter operações não determinísticas, as quais não são unicamente definidas, mas são limitadas por um conjunto especificado de possibilades. A máquina que executaria tal tipo de operação, admitiria a escolha de qualquer uma destas soluções para uma certa condição de parada. Isto leva ao conceito de algoritmo **não-determinístico**.

Um algoritmo não-determinístico só terminará sem sucesso se não existir um conjunto de soluções que levem ao sucesso. Um algoritmo não-determinístico pode ser interpretado como constituído de dois estágios: o estágio de suposição e o de verificação ou checagem. É importante ressaltar que a principal propriedade de máquinas não-determinísticas é que elas são capazes de selecionar concretamente a solução no conjunto das soluções disponíveis (caso ela exista).

5.1.1 Os espaços P e NP e seu relacionamento

O termo espaço **NP** pode ser interpretado como a classe dos algoritmos não-determinísticos em tempo polinomial. E a classe de problemas solúveis por algoritmos determinísticos em tempo polinomial tem sido chamada de espaço **P**.

O relacionamento entre as classes **P** e **NP** é fundamental. Cada problema de decisão resolvido por um algoritmo determinístico em tempo polinomial é também resolvido por um algoritmo não determinístico de tempo polinomial, isto é **P** $\subseteq$ **NP**. Para se ver isto, basta observar que qualquer algoritmo determinístico pode ser utilizado como o estado de verificação (checagem) de um algoritmo não determinístico.

Se R pertence a **P** e AL é um algoritmo determinístico de tempo polinomial que resolve R, pode-se obter um algoritmo não determinístico de tempo polinomial para R, usando simplesmente o próprio AL como estágio de verificação e ignorando o estágio de suposição. Portanto, R pertence a **P**, implica que R pertence a **NP**.

Uma das questões mais importantes, em aberto, da Ciência da Computação é saber se as classes **P** e **NP** são iguais. Existem duas linhas de pesquisa tentando resolver esta questão:

a) Provando que os problemas da classe **NP** são intratáveis, isto é, que os problemas são tão difíceis que nenhum algoritmo de complexidade polinomial pode possivelmente resolvê-los. Infelizmente, provar que **P** é diferente de **NP** é tão difícel quanto provar que **P** é igual a **NP**;

b) Examinando o relacionamento entre os problemas **NP**. Por exemplo, na classe **NP**, a principal subclasse de problemas é provado ser dos problemas **NP-completos**. A prova que um problema é **NP-completo** é geralmente considerado um forte argumento para abandonar tentativas de providenciar um algoritmo eficiente para resolvê-los.

Problemas completos tem a propriedade que todos os problemas na classe **NP** (incluindo NP-completos) são polinomialmente redutíveis a qualquer um deles, ou seja resolver um significa resolver todos.

A demostração de que um problema é **NP-completo** é um processo constituído de duas partes, na primeira se tem que provar que o problema pertence a classe **NP** e, depois, demonstrar a redução (a transformação polinomial) a um problema **NP-completo** conhecido ao problema em pauta.

5.1.2 Problemas em NP

Entre os problemas contidos na classe **NP** destacam-se o problema da Caixeiro viajante, o da determinação do clique máximo, o da mochila e muitos outros. Apesar de nenhum destes problemas em **NP** ter, aparentemente cota inferior exponencial, não se conhece algoritmos polinomiais para resolvê-los e não se conhece, ainda, demonstrações de que qualquer um dos problemas tem cota inferior exponencial.

Formalmente, define-se a classe **NP** como a classe de problemas R, tais que, para algum M na classe P(polinomial) e para algum inteiro positivo k, x pertence a R se, e somente se, para algum y tal que:

$$\log y <= \log^k x, (x,y) \text{pertence a } M.$$

Intuitivamente, um problema R está em **NP** se qualquer instância x de R, possui uma demonstração y "curta" de que x pertence a R. Esta demonstração y deve ser verificável em tempo polinomial. Para ilustrar a definição, são apresentados alguns exemplos.

Seja G=(VG, aG) um grafo orientado com custos cij positivos associados às arestas (i,j); quando não existir uma aresta (i,j), por convenção cij será infinito. Suponha VG=n >1. Uma viagem em G é um circuito que contém cada vértice em VG uma e só uma vez. A soma dos custos das arestas na viagem é chamada custo da viagem. O problema do caixeiro viajante (PCV) consiste em achar uma viagem mínima, isto é, entre todas as viagens possíveis em G, uma viagem de custo mínimo. Uma instância deste problema é:

> Seja x = um conjunto de n cidades e um inteiro m, então será que existe uma solução mínima que seja menor ou igual a m?

Neste caso R é o conjunto das instâncias x que possuem SIM, isto é, existe uma viagem y candidata a solução e pode-se verificar em tempo polinomial se y tem custo menor ou igual a m e se y é realmente uma viagem visitando cada cidade uma só vez.

Outro exemplo é o problema denominado Sat-3 no qual é considerada uma expressão lógica com operandos booleanos E, OU, NÃO. As variáveis podem ter valor verdadeiro ou falso. Cada parte da expressão entre parênteses é chamado de cláusula e entre cada par de cláusulas ocorre um operador E. Dentro de cada cláusula, ocorrem no máximo dois operadores OU entre variáveis ou negações de variáveis. Diz-se que uma expressão assim definida, é satisfazível se existe uma atribuição de valores verdadeiro ou falso às variáveis na expressão que torne seu valor verdadeiro. Uma instância do Sat-3 é:

> Seja x = uma expressão lógica como definida acima, então será x satisfazível?

5.1.3 Problemas NP-completo e redução polinomial

No espaço NP, um problema R1 é chamado **polinomialmente transformável** ou **polinomialmente redutível** para o problema R2 se existe um algoritmo determinístico de tempo polinomial o qual transforma ou reduz R1 para R2. Isto será anotado por R1 [<=] R2.

Dois problemas R1 e R2 são chamados polinomialmente equivalentes se R1 [<=] R2 e R2 [<=] R1.

Um problema R é chamado **NP-completo** se R pertence a **NP** e cada problema em **NP** é polinomialmente redutível a R. Uma prova que um problema **NP** é **NP-completo** é uma demonstração que o problema não é polinomial, ou seja, que não exista um algoritmo determinístico de tempo polinomial, a menos que cada problema **NP** pertença a P.

A noção de problema **NP-completo** foi introduzida em 1971 por COOK, o qual provou o primeiro de todos problemas **NP-completos**, o então chamado problema da satisfatibilidade, também conhecido como Sat-3, descrito a pouco, ou por:

> Se o problema da satisfatibilidade pode ser resolvido em tempo polinomial por um algoritmo determinístico então a classe P é igual a classe **NP**, P = **NP**.

Todos os problemas considerados em termos de transformação de um para outro, são principalmente formulados como um problema de decisão, os quais tem por solução a resposta SIM ou NÃO.

A principal técnica usada para transformar ou reduzir um problema para outro é uma transformação construtiva que mapeie cada instância do primeiro problema em uma instância equivalente do segundo. Tal transformação provê o significado para converter

cada algoritmo que resolve o segundo problema em um algoritmo correspondente para resolver o primeiro problema.

O algoritmo geral para provar que o problema R é **NP-completo**, segue os seguintes passos:

a) Formular R como um problema de decisão;
b) Escolher um problema **NP-completo** R0 conhecido;
c) Transformar R0 para R;
d) Mostrar que a função de transformação é de complexidade polinomial.

Nota-se que quando R1 [<=] R2 e R2 [<=] R então R1 [<=] R, ou seja a relação [<=] é transitiva, o que permite dizer que se conseguirmos resolver um problema da classe, então todos os problemas serão resolvidos.

5.2 Classes de computação paralela

Para poder falar de classes de computação paralela é necessário, lembrar da formalização do conceito de algoritmo paralelo. Um algoritmo paralelo ou concorrente é um algoritmo que permite que operações sejam efetuadas simultaneamente ou em paralelo. Basicamente isto significa a utilização do fato que computações de larga escala e problemas de processamento de informação podem ser particionados de forma que várias partes do trabalho podem ser efetuadas independentemente e em paralelo e os resultados serão combinados juntos quando todas subcomputações forem completadas.

Um algoritmo sequencial especifica sequencialmente a execução de uma sequência de comandos, esta execução é chamada de processo. Um algoritmo paralelo ou concorrente especifica duas ou mais sequências de algoritmos que podem ser executados simultaneamente como processos paralelos. Portanto, um processo é uma coleção de controle sequencial de comandos com acesso local ou global de dados e que podem ser executados em paralelo com outras unidades de programa.

Para garantir que um algoritmo concorrente trabalhe corretamente, processos interagem, comunicam-se, são sincronizados e trocam dados. Os pontos onde os processos podem se comunicar com outros processos são denominados de pontos de interação. Esses pontos de interação dividem um processo em estágios. Ao final de cada estágio, um processo pode se comunicar com outros processos antes de iniciar novo estágio.

Sincronização é uma maneira de garantir que um processo só inicie após a conclusão de outro, ou depois que determinados dados lhe sejam transferidos. Em um algoritmo síncrono, processos podem ser bloqueados até que certos estágios de outros processos sejam concluídos. Este tipo de sincronização, onde processos aguardam até que dados lhe sejam fornecidos é chamada de sincronização explícita.

A sincronização implícita é causada quando processos necessitam utilizar recursos compartilhados, como variáveis globais ou até mesmo dispositivos. Um processo deve aguardar que o dispositivo seja liberado para que ele possa vir a utilizar.

Nos algoritmos paralelos assíncronos, excluindo as variáveis globais, nenhuma sincronização se faz necessária para garantir que os dados específicos estejem disponíveis aos processos em diferentes momentos. Os processos nunca esperam por dados mas continuam ou concluem seu processamento, dependendo da informação corrente contida nas variáveis globais.

Em um algoritmo sincronizado uma tarefa é decomposta em subtarefas, as quais, sempre que possível, do mesmo tamanho. Portanto cada subtarefa é resolvida por um conjunto explícito de restrições de precedência, dando a ordem de processamento delas.

Uma importante medida de paralelismo é a quantidade de cálculos que podem ser executados sem interrupção, a qual é chamada de **granularidade**. Granularidade é, portanto, a quantidade de trabalho computacional que pode ser realizado por um processador entre pontos de interação, pontos de sincronização.

Operações simultâneas em algoritmos paralelos requerem uma convenção especial de representação. Várias notações têm sido propostas como: **for-all** e **cobegin-coend**. Para expressar o processamento assíncrono de dois ou mais processos e o retorno dos resultados ao processo principal de controle, os comandos **fork** e **join** são utilizados.

5.2.1 Classes de problemas AC e NC

Um circuito booleano B com n entradas e m saídas é um grafo orientado acíclico com nós (chamados gates) rotulados da seguinte forma: o circuito B possui k nós de entrada com grau de entrada zero, rotulados $x_1, x_2, ..., x_k$ respectivamente. Todos os outros nós de grau de entrada um são rotulados com NÃO, enquanto que todos os outros nós têm grau de entrada dois e são rotulados com E ou OU. Exatamente m nós são de saída e rotulados com y_1, $y_2, ..., y_m$, respectivamente. Todo nó de entrada possui pelo menos um caminho dele mesmo para algum nó de saída.

O tamanho s(n) do circuito é o número de arestas; a profundidade d(n) é o comprimento do mais longo caminho de um nó de entrada até um nó de saída. O tamanho relaciona-se com o conteúdo de hardware e a profundidade com o tempo de computação.

Uma família de circuitos é **log-espaço uniforme** se dado um inteiro n, a descrição do n-ésimo circuito pode ser gerada em um espaço logarítmico por um algoritmo determinístico sequencial. O objetivo é, dado qualquer valor n, conseguir de forma eficiente gerar um circuito para resolver um problema.

A classe NC^k para k>=0 é a coleção de problemas solucionáveis por um circuito log-espaço uniforme com tamanho polinomial e profundidade log^k n. A classe **NC** é a união de todas as classes NC^k.

Se os nós com as funções E ou OU no circuito podem ter grau de entrada de tamnho variável, então se tem a classe **AC^k** e a classe **AC** de problemas é definida como a união de todas as classes **AC^k**.

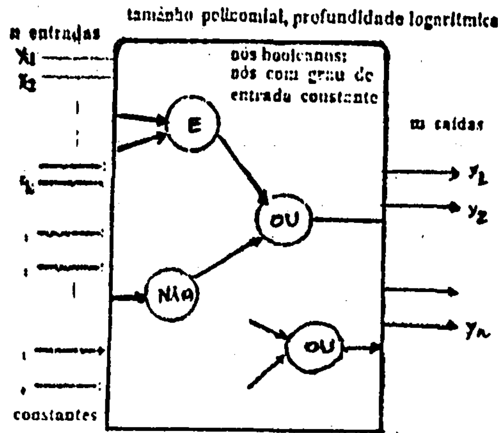


Fig. 5.1 Circuito NC ([TER90])

A seguir são enumerados alguns exemplos de problemas da classe **NC^1**: soma ou multiplicação de dois inteiros de n-bits; soma de n números de n-bits; Multiplicação de matrizes de inteiros ou booleanos e ordenação de n inteiros de n bits.

Exemplos de problemas em **NC^2** são: divisão de dois números, cada um com n bits; valor de expressão booleana; computação sobre matrizes e potencias; inversão de matrizes e cálculo de determinante; ordenação topológica de grafo acíclico e árvore espalhada mínima em grafo.

5.3 Relação entre as várias classes de complexidade de tempo

Considerando as classes de complexidade vistas nos itens anteriores, tem-se que a relação entre elas é dada pela inclusão a seguir e descrita na figura 5.2. Não se sabe se as inclusões são estritas ou não.

$NC^1 \subset NC^2 \subset \dots \subset NC \subset P \subset NP \subset$ P-espaço.

Tem-se, ainda, que dado um circuito em AC^k qualquer nó com grau de entrada n no circuito AC^k pode ser convertido para uma árvore de profundidade log n com funções do mesmo tipo, com entrada dois, tal que os nós de saída calculam a mesma função que a original. O resultado é um circuito com tamanho polinomial, com profundidade log^k+1 n e com grau de entrada dois e, portanto, em NC^k+1.

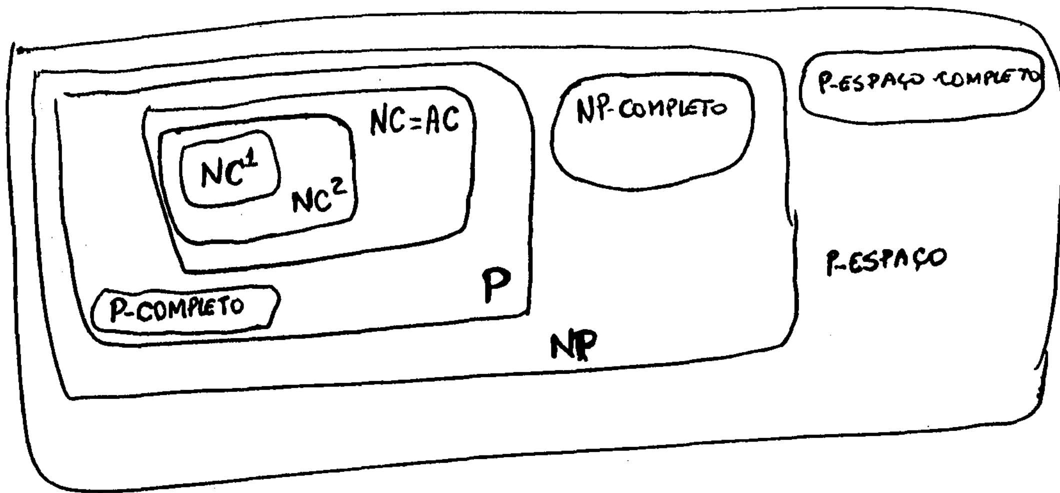


Fig. 5.2 Classes de complexidade ([TER90])

Por fim, recorda-se de forma resumida que a classe **NC^1** consiste de todos os problemas solucionáveis por uma família de circuitos uniformes de profundidade o(logn), onde n é o número de bits de entrada e a classe **NC^2** consiste de todos os problemas solucionáveis por uma família de circuitos uniformes de profundidade o(log^2 n) e tamanho polinomial.

5.4 Questões em aberto

Como vimos, a classe **NP-completa** é aquela constituída por problemas A tais que qualquer problema R em **NP** é polinomialmente redutível a A, isto é R [<=] A. Isto significa que A é pelo menos tão difícil de ser resolvido como qualquer problema em **NP**. Se alguém descobrir um algoritmo de rapidez polinomial para A, então pode-se resolver qualquer problema R de **NP** em tempo polinomial.

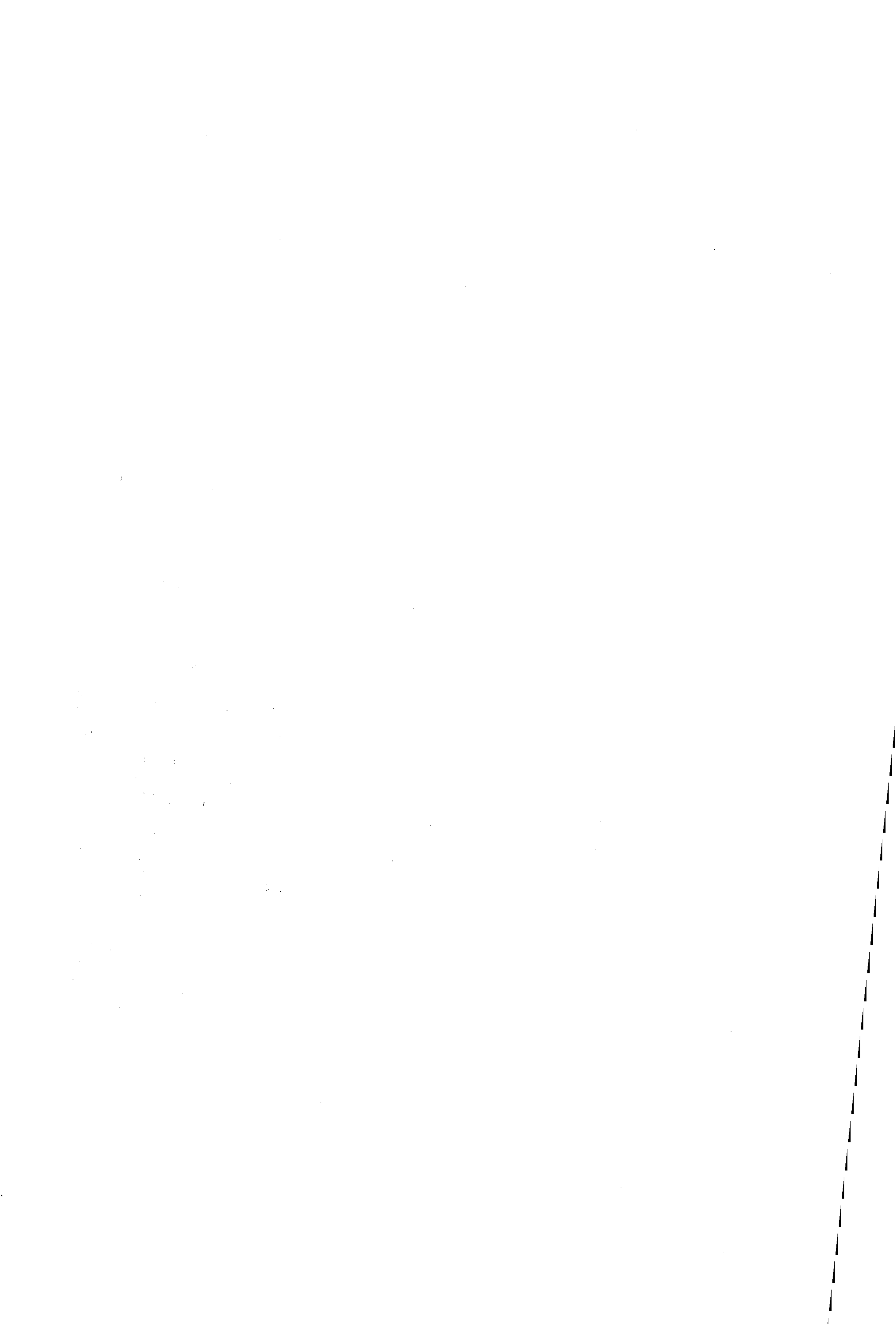
Devido a transitividade da redução polinomial, para se provar que um problema A é **NP-completo**, basta provar que o problema A está em **NP** e que existe outro problema B tal que **Sat-3** [<=] B e B [<=] A, onde Sat-3 é o problema da satisfatibilidade, demonstrado por Cook em 1971 como sendo o primeiro problema **NP-completo**.

Um dos problemas em aberto, e talvez seja o mais famoso em Ciência da Computação é que: Será **P = NP**? Isto implicaria na existência de algum problema L em **NP-completo** que esteja em P, o que por sua vez, resultaria que todos, por redução polinomial, também estivessem em **P**.

Mostrar que **P** está contido em **NP** é simples, uma vez que pode-se facilmente extender algoritmos determinísticos em não determinísticos. Portanto para se provar que **P=NP**, basta que alguém descubra um algoritmo polinomial para qualquer problema **NP-completo**. Mas, infelizmente, até hoje não se sabe se existe algum L nestas condições. Acredita-se que **P** seja diferente de **NP**, mas demonstrar isto é tão dificil quanto provar a igualdade.

Outro problema em aberto é decorrente da busca da definição de problemas que são inerentemente sequenciais, ou seja, problemas que podem ser facilmente resolvidos por algoritmos sequenciais em tempo polinomial, mas que não se sabe se eles admitem algum algoritmo paralelo rápido. Estes problemas são conhecidos como problemas **P-completos**.

Sabe-se que a classe **NC**, que consiste de problemas que são resolvidos por algoritmos paralelos num tempo o((log n)^k), utilizando um número polinomial de processadores para dado K>=0, está contido na classe **P**. Mas será que a classe **NC** é igual a classe **P**? Esta pergunta é tão dificil quanto a anterior, mas acredita-se fortemente que **NC** $\neq$ **P**, o que implica na existência de problemas inerentemente sequenciais, os quais não podem ser resolvidos por algoritmos paralelos eficientes, como no caso dos problemas **P-completos**.



6 TÉCNICA DE DIVISÃO E CONQUISTA

6.1 Formalização da técnica

O termo divisão e conquista é originado dos generais napoleônicos (1800 a 1814) que utilizavam a tática militar de dividir o exército inimigo para conquista-lo.

Esta é uma técnica para projetar algoritmos eficientes. É muito empregada para solucionar problemas sequenciais, mas também é importante, por ser uma técnica muito utilizada para o desenvolvimento de algoritmos paralelos.

Uma aplicação típica da técnica de divisão e conquista tem a estrutura descrita na figura 6.1.

```
1) Dado o problema P;

2) Se P é divisível em problemas menores

2.1)  Então
             - Dividir P em duas ou mais partes P1, P2,..., Pk;
             - Resolver P1;
             - Resolver P2;
             -  . . .
             - Resolver Pk;
             - Combinar as K soluções parciais na solução de P.
2.2)  Senão
             - Resolva P diretamente.
```

Fig. 6.1 Estrutura recursiva da técnica de divisão e conquista

Uma formalização da técnica pode ser feita, através de dada uma instância de um problema, de tamanho n, o método divide-a em k instâncias disjuntas (1 <= k <= n), que correspondem a k subproblemas disjuntos. Estes problemas são resolvidos separadamente e, então, acha-se uma forma de combinar as soluções parciais para se obter a solução para a instância original.

Quase sempre, combinar as soluções dos subproblemas pequenos do problema é mais simples do que resolver o problema diretamente. Esta técnica produz algoritmos eficientes e, portanto, é amplamente utilizada em muitas áreas da computação no desenvolvimento de algoritmos.

A aplicação sucessiva do método é muito usada, gerando algoritmos que chamam-se a si mesmo, direta ou indiretamente. Eles são chamados de recursivos. Os algoritmos são relativamente fáceis de serem escritos e compreendidos, são mais claros e concisos, além de serem fáceis de serem analisados quanto a certificação e rapidez.

O que viabiliza a implementação de rotinas recursivas é uma pilha, na qual são armazenados os dados utilizados por cada vez que a rotina é chamada, até que seja concluída. Isto significa que todos os dados não globais ficam armazenados na pilha. A pilha é dividida em pedaços, os quais são blocos de localizações (registradores). Cada chamada da rotina usa um destes pedaços da pilha de tamanho que depende da rotina particular chamada.

O tempo requerido para uma chamada da rotina é proporcional ao tempo requerido para avaliar o parametro atual e armazenar os ponteiros dos seus valores na pilha. O tempo de retorno não é superior ao da chamada.

De forma a avaliar o desempenho de tempo e espaço de um algoritmo recursivo, escreve-se uma expressão descrevendo a função de tempo ou espaço de todo problema em termos da função de tempo ou espaço das pequenas instâncias do problema. Ou seja, para calcular a complexidade de tempo de algoritmos recursivos, se faz uso de relações de recorrência, também chamadas simplesmente de recorrência.

Uma função Ti(n) é associada à i-ésima rotina e indica o tempo de execução da i-ésima rotina como uma função do parâmetro de entrada n. O conjunto resultante de equações simultâneas de recorrência pode, então, ser resolvido. Frequentemente, só uma rotina é envolvida e T(n) depende dos valores de T(m), para um conjunto finito de m menor que n.

Uma das razões desta técnica ser útil ao projeto de algoritmos paralelos é devido à estrutura da técnica, como vista na figura 6.1 de divisão do problema em k subproblemas e na resolução destes subproblemas separadamente. A resolução dos subproblemas pode ser feita em paralelo, apesar da necessidade de comunicação de dados entre eles. Para o sucesso do emprego da técnica de divisão e conquista é necessário que não se perca o tempo ganho na solução dos subproblemas em paralelo, na combinação ou construção da solução do problema global.

Existem muitas áreas da análise numérica onde métodos baseados na técnica de divisão e conquista são utilizados. Na álgebra linear eles são utilizados para solucionar sistemas tridiagonais e sistemas onde a matriz de coeficientes é do tipo banda, na obtenção de autovalores de matrizes simétricas tridiagonais, no cálculo de zeros de funções e na solução de equações diferenciais parciais. Vários problemas não numéricos também são solucionados eficientemente por algoritmos baseados por divisão e conquista.

Uma regra básica para projetar bons algoritmos recursivos, é a manutenção do balanceamento, ou seja subdividir o problema de tamanho n, em k subproblemas, cada um de tamanho aproximadamente igual, (n/k).

6.2 Caracterização via aplicações

Uma vez formalizada a técnica da divisão e conquista e seu emprego no projeto de desenvolvimento de algoritmos, serão mostrados alguns exemplos de aplicações de algoritmos baseados na divisão e conquista para a solução de alguns problemas, tais como: determinação do máximo e mínimo de uma lista linear; ordenação de listas lineares, multiplicação de dois inteiros de n-bits cada e multiplicação de duas matrizes nxn.

Juntamente com a caracterização do problema, com a descrição da solução, será analisado o algoritmo quanto à complexidade de tempo. Através disto, acredita-se que se terá uma caracterização prática da técnica de divisão e conquista e se poderá identificar qualidades que sirvam de orientação no emprego desta técnica no desenvolvimento de algoritmos paralelos.

6.2.1 Determinação do máximo e mínimo de uma lista

Considere o problema de achar tanto o elemento máximo, quanto o elemento mínimo de um conjunto ou uma lista, contendo n elementos. Para uma maior formalização do problema, seja a lista L = (M1,M2, ...,Mn), onde Mi são números que compoem a lista. (1 <= i <= n).

A solução mais simples, é aquela onde se supoem que o primeiro elemento da lista seja, a princípio, tanto o elemento máximo quanto o elemento mínimo. Então, compara-se com os demais elementos da lista. Ao final se terá o elemento máximo e o elemento mínimo. É fácil de verificar que esta solução exigiria (n-1) comparações para a determinação do elemento máximo e (n-1) comparações para a determinação do elemento mínimo, portanto seriam necessários 2(n-1) comparações.

A solução por um algoritmo baseado em divisão e conquista, consiste em dividir a lista L, em duas sublistas L1 e L2 de tamanhos aproximadamente iguais, ou seja:

L1=(M1,M2,...,Mk) e L2=(Mk+1, Mk+2,...,Mn) onde $k= \lfloor n/2 \rfloor$.

Os elementos máximo e mínimo são calculados pelo aplicar recursivo do algoritmo nas duas sublistas L1 e L2. O máximo e o mínimo de L são calculados pelo comparar os máximos e mínimos das sublistas, onde é gasto mais duas comparações. O algoritmo 6.1 é a formalização destas idéias.

```
ALGORITMO 6.1
        MaxMin - determinação do máximo e mínimo de uma lista.
Entrada: (n, M1, M2, ...,Mn)  n>=1 e L=(M1, M2, ...,Mn)
Saída: (max, min)

 1  Se n=1 então pare com saída <M1,M1> fim-se;
 2  Se n=2
 3     então Se M1 < M2 então pare com saída <M2,M1>
 4                       senão pare com saída <M1,M2>
 5     senão k = ⌊ n/2 ⌋
 6           <max1,min1> = MaxMin(k,M1,...,Mk);
 7           <max2,min2> = MaxMin(n-k,Mk+1,...,Mn);
 8           pare com saída (maximo (max1,max2),
 9                          minimo (min1,min2));
10  Fim-se.
```

Na figura 6.2 é mostrado esquematicamente as várias etapas da execução do algoritmo 6.1 para a entrada (n=5, L=(6, 11, -3, -4, 7)). Um diagrama como este na figura 6.2, foi denominado por Terada (em [TER91]) de pilha de execução recursiva. Cada quadro nesta pilha chama-se quadro de execução e corresponde à execução do algoritmo cujo nome e entrada ocorrem na primeira linha do quadro; e a saída resultante da execução é escrita na última linha do quadro, precedida pelo sinal de igual.

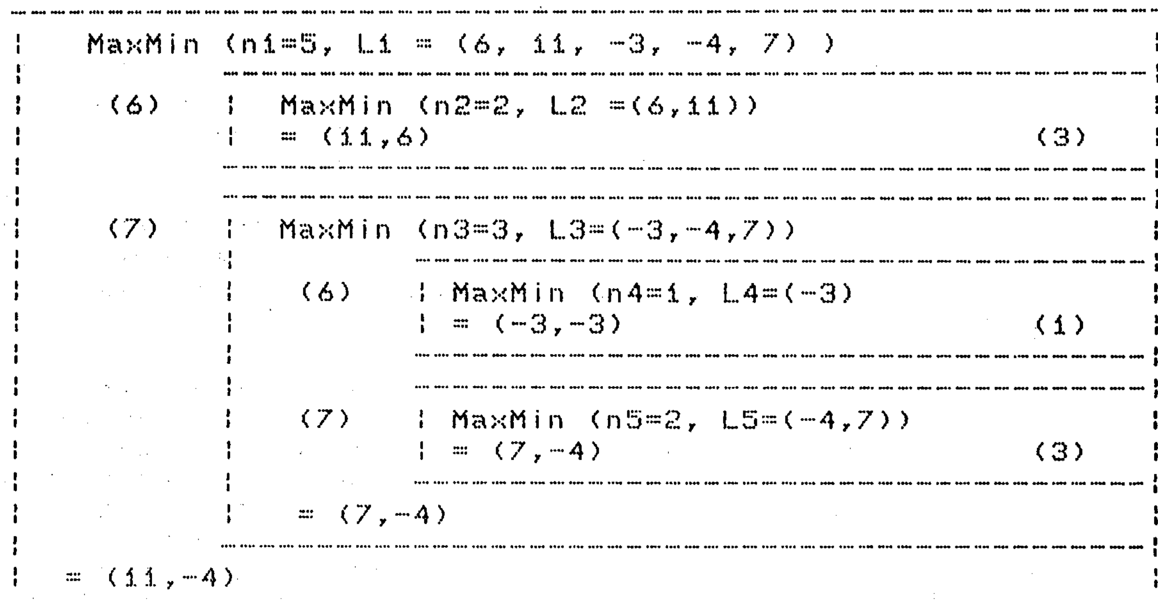


Fig. 6.2 Pilha de execução recursiva do algoritmo MaxMin

Seja T(n) o número de comparações realizados entre os elementos da lista L necessários pelo algoritmo 6.1 para achar os elementos máximo e mínimo da lista de n elementos. Se a lista tiver um elemento, nenhuma comparação será feita (T(1)=0); se a lista tiver dois elementos, só é necessário uma comparação

$(T(2)=1)$; se a lista tiver mais de dois elementos $n>2$, $T(n)$ é o número total de comparações utilizadas nas chamadas recursivas do algoritmo, com $n/2$ elementos cada, mais duas comparações finais, que relacionam os máximos e mínimos das sublistas.

A equação de recorrência que caracteriza o tempo de execução do algoritmo 6.1, é dado por:

$$T(n) = \begin{cases} 1, & \text{para } n=2 \\ 2T(n/2)+2, & \text{para } n>2, \end{cases}$$

a solução desta equação de recorrência é limitada por $\lceil 3n/2-2 \rceil$, o que pode ser demonstrado por indução matemática. Portanto, o algoritmo 6.1, baseado em divisão e conquista reduz o número de comparações por um fator constante.

Por outro lado, pode-se demonstrar que a cota inferior para o problema de determinação do elemento máximo e mínimo é também da ordem $| 3/2n -2 |$ comparações, portanto este algoritmo é ótimo, em termos de complexidade de tempo.

6.2.2 Ordenação de listas

Como já foi visto no capítulo 4, o problema de ordenar uma lista de n objetos em ordem não decrescente, resolvido pelo algoritmo 4.1, tem uma rapidez da ordem n^2.

Devido a grande necessidade de ordenação nos centros de processamento, que utilizam cerca de 1/4 do seu tempo em tarefas de ordenação, muitos algoritmos foram e vem sendo desenvolvidos, na tentativa de se conseguir algoritmos cada vez mais eficientes.

Em alguns algoritmos a operação base é a comparação e a principal questão é qual é o número mínimo de comparações que qualquer algoritmo de ordenação necessita para o problema de ordenação.

A execução de qualquer algoritmo de ordenação pode ser representada por uma árvore binária chamada árvore de decisão. Nas árvores de decisão, cada nó interno contém dois elementos, x e y, sendo comparados. A subárvore esquerda deste nó representa as comparações a serem feitas em seguida, se $x<y$, e a subárvore direita representa as ações a serem executadas se $x>=y$. Cada folha da árvore contém uma permutação dos elementos de acordo com a ordem estabelecida. Cada caminho da raiz até uma folha corresponde a uma sequência mínima de comparações necessária para se concluir uma ordem dos elementos. Cada uma das n! permutações de n elementos pode ocorrer num problema de ordenação, portanto a árvore contém n! folhas.

Com isto, pode-se demonstrar que o problema de ordenação de n elementos tem cota inferior de rapidez pessimista da ordem n log n, $\Omega(n \log n)$.

Pode-se, também, demonstrar que a rapidez média de qualquer algoritmo de ordenação (baseado em comparação) tem também cota inferior de $\Theta(n \log n)$. O número médio de comparações é o comprimento médio da tais caminhos, e é igual à profundidade média das n! folhas na árvore de decisão.

Para se ordenar uma lista L=(M1, M2,...,Mn) por intercalação, divide-se L em duas sublistas L1 e L2, onde L1=(M1,M2,...,Mk) e L2=(Mk+1,Mk+2,...,Mn) para $k=\lfloor n/2 \rfloor$.

No final combina-se as sublistas ordenadas, pegando o menor dos primeiros da lista. Toda a intercalação pode ser feita com n-1 comparações e, portanto, em tempo cn para um valor c constante maior que um. No algoritmo 6.2 é apresentado a ordenação por intercalação da lista L em ordem não decrescente.

ALGORITMO 6.2

OrdInter - Ordenação por intercalação

Entrada: (n,M1,M2,...,Mn) onde n>1 e L=(M1,...,Mn)
Saída: Lista L' ordenada em ordem não decrescente.

```
1  Se n=1 então pare com saída (M1);
2         senão  k = ⌊n/2⌋
3                L1= OrdInter (k,M1,M2,...,Mk);
4                L2= OrdInter (n-k,Mm+1,Mk+2,...,Mn);
5                "Intercalar L1 e L2 pegando o menor dos pri-
6                meiros das listas obtendo-se uma lista L' ";
7                pare com saída (L');
8  Fim-se.
```

Para melhor ilustrar o algoritmo 6.2, é mostrado na figura 6.3 a pilha de execução recursiva com a lista de entrada L=(6,11,-3,-4,7) com n=5 elementos.

Seja T(n) o tempo de execução do OrdInter (algoritmo 6.2) para uma entrada (n,M1, M2, ...,Mn), onde a operação elementar considerada é a comparação entre dois elementos. A relação de recorrência é

$$T(n) = \begin{cases} 0, & \text{Se } n=1 \\ T(\lfloor n/2 \rfloor)+T(\lceil n/2 \rceil)+cn, & \text{Se } n>1, \end{cases}$$

o que resulta que este algoritmo seja da ordem n log n e, portanto, o algoritmo é ótimo em termos de rapidez.

Caso no OrdInter não seja observado o princípio da equiparação, ou seja que os subproblemas sejam subdivididos em duas partes iguais, mas em duas partes de tamanho um e n-1, resulta uma equação de recorrência que tem por solução da ordem n^2.

```
-----------------------------------------------------------------------
| OrdInter (n1=5, (6,11,-3,-4,7))                                      |
|        --------------------------------------------------------------|
| (3) | OrdInter (n2=2, L2=(6,11))                                     |
|     |           -----------------------------------------------------|
|     | (3)    | OrdInter (n3=1, L3=(6))                               |
|     |        | = (6)                                       (1)       |
|     |           -----------------------------------------------------|
|     |           -----------------------------------------------------|
|     | (4)    | OrdInter (n4=1, L4=(11)                               |
|     |        | = (11)                                      (1)       |
|     |           -----------------------------------------------------|
|     | = (6,11)                                            (7)        |
|        --------------------------------------------------------------|
|        --------------------------------------------------------------|
| (4) | OrdInter (n5=3, L5=(-3,-4,7))                                  |
|     |           -----------------------------------------------------|
|     | (3)    | OrdInter (n6=1, L6=(-3))                              |
|     |        | = (-3)                                      (1)       |
|     |           -----------------------------------------------------|
|     |           -----------------------------------------------------|
|     | (4)    | OrdInter (n7=2, L7=(-4,7))                            |
|     |        |           --------------------------------------------|
|     |        | (3)    | OrdInter (n8=1, L8=(-4))                     |
|     |        |        | = (-4)                             (1)       |
|     |        |           --------------------------------------------|
|     |        |           --------------------------------------------|
|     |        | (4)    | OrdInter (n9=1, L9=(7))                      |
|     |        |        | = (7)                              (1)       |
|     |        |           --------------------------------------------|
|     |        | = (-4,7)                                    (7)       |
|     |           -----------------------------------------------------|
|     | = (-4,-3,7)                                         (7)        |
|        --------------------------------------------------------------|
| = (-4,-3,6,7,11)                                          (7)        |
-----------------------------------------------------------------------
```

Fig. 6.3 Pilha de execução recursiva do algoritmo OrdInter

6.2.3 Multiplicação de inteiros

Considere dois números x e y inteiros e não negativos, ambos expressos numa base b>=2 com n algoritmos (n>=1). Quer-se resolver eficientemente o problema de se obter o produto x.y.

O algoritmo tradicional de multiplicação para este problema requer $o(n^2)$ multiplicações algarismo a algarismo. Um algoritmo baseado na técnica de divisão e conquista, pode reduzir a rapidez $o(n^{\log 3})$ o que aproximadamente é $o(n^{1.59})$, como é mostrado no método Karatsuba e Ofman, que é descrito a seguir.

Seja x e y dois números de n-bits. Por medida de simplicidade, será assumido que n é uma potência de dois. Divide-

se x e y em duas metades de n/2 algarismos cada (mantendo assim o princípio da equiparação), obtendo:

$$x = x_1 . b^{(n/2)} + x_2 \quad e \quad y = y_1 . b^{(n/2)} + y_2,$$

$$x.y = x_1.y_1 \;.\; b^n + (x_1.y_2 + x_2.y_1).b^{(n/2)} + x_2.y_2$$

No produto acima, temos quatro multiplicações de dois números com n/2 algarismos cada, três somas e duas multiplicações da forma $c.b^l$, que consiste na prática de uma shift, ou seja, colocar l zeros a direita de c, podendo ser efetuada em tempo proporcional a l. O algoritmo que se pretende, deve reduzir de quatro para três multiplicações, através do reescrever a fórmula acima.

Como, $x_1.y_2+x_2.y_1 = (x_1+x_2).(y_1+y_2)-x_1.y_1 - x_2.y_2$,
temos que,

$$x.y=(x_1.y_1)b^n + ((x_1+x_2).(y_1+y_2) - x_1.y_1 - x_2.y_2)b^{n/2} + x_2.y_2.$$

Portanto, pode-se calcular x.y, através de três multiplicações, quatro somas e duas subtrações. Observa-se que os produtos $x_1.y_1$ e $x_2.y_2$ são utilizados duas vezes. Observa-se ainda, que as adições são efetuadas mais rapidamente do que as multiplicações.

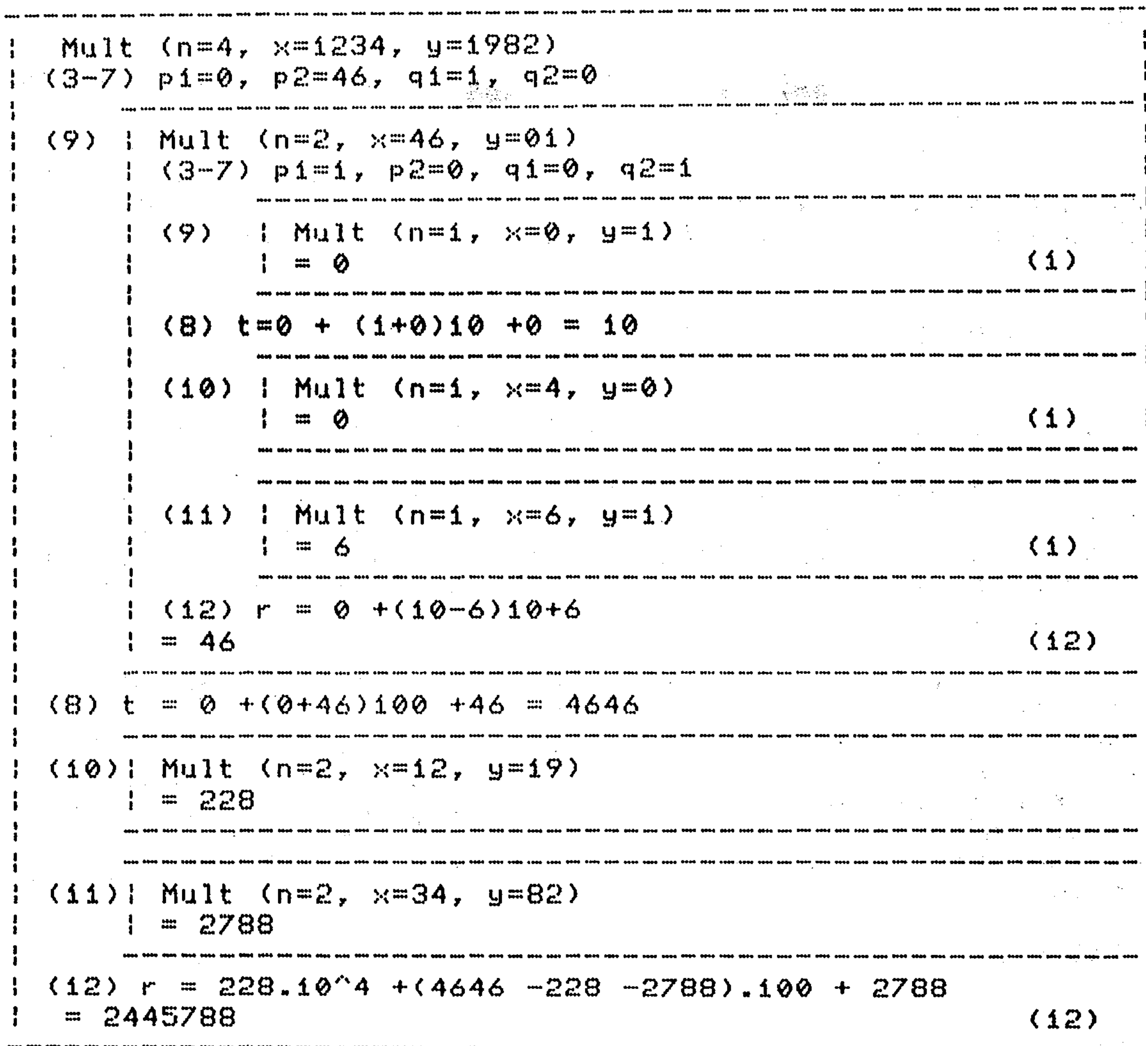
O algoritmo 6.3 implementa a multiplicação de dois inteiros de n-bits. Chama-se a atenção para o fato de que p, calculado na linha 3 tem 1+n/2 algarismos, pois se x_1 e x_2 tem n/2 algarismos e o algarismo mais à esquerda de p é chamado de p_1 pelo algarismo. Uma notação análoga é usada para o q calculado na linha 6 e, em consequência, no cálculo de t na linha 8, p_1 e q_1 são de apenas um algarismo, enquantoque p_2 e q_2 são de n/2 algarismos.

ALGORITMO 6.3
Mult - multiplicação de dois inteiros

Entrada: (n,x,y) onde x e y são inteiros, não nulos de n-bits;
Saída: (r) onde r = x.y

```
1 Se n=1 então r = x.y;
2         senão
3           p = x1+x2;
4             (p tem no máximo 1+n/2 algarismos, ou seja
5              p=p1b^n/2 +p2 e p1 tem um algarismo);
6           q = y1+y2;
7             (analogamente, seja q=y1b^n/2 +q2);
8           t = p1.q1.b^n +(p1.q2 + p2.q1)b^n/2 +
9                          + Mult (n/2, p2, q2);
10          u = Mult (n/2, x1, y1);
11          v = Mult (n/2, x2, y2);
12          r = u.b^n + (t-u-v).b^n +v
13 fim-se
14 pare com saída (r).
```

A seguir é exemplificado a execução do algoritmo 6.3 como o produto dos números x = 1234 e y = 1982, portanto com n=4, através da figura 6.4, onde se tem a pilha de execução recursiva do algoritmo Mult.



```
Mult (n=4, x=1234, y=1982)
(3-7) p1=0, p2=46, q1=1, q2=0

(9)  | Mult (n=2, x=46, y=01)
     | (3-7) p1=1, p2=0, q1=0, q2=1
     |
     | (9)  | Mult (n=1, x=0, y=1)
     |      | = 0                                   (1)
     |
     | (8) t=0 + (1+0)10 +0 = 10
     |
     | (10) | Mult (n=1, x=4, y=0)
     |      | = 0                                   (1)
     |
     | (11) | Mult (n=1, x=6, y=1)
     |      | = 6                                   (1)
     |
     | (12) r = 0 +(10-6)10+6
     | = 46                                       (12)

(8) t = 0 +(0+46)100 +46 = 4646

(10)| Mult (n=2, x=12, y=19)
    | = 228

(11)| Mult (n=2, x=34, y=82)
    | = 2788

(12) r = 228.10^4 +(4646 -228 -2788).100 + 2788
 = 2445788                                    (12)
```

Fig. 6.4 Pilha de execução recursiva do algoritmo Mult

O algoritmo 6.3 tem sua rapidez expressa por uma equação de recorrência T(n), que é limitada pela expressão abaixo, onde c é uma constante.

$$T(n) <= 3T(n/2) + cn,$$

Pode-se generalizar esta tipo de equação de recorrência, uma vez que elas são muito utilizadas em algoritmos baseados em divisão e conquista, elas dependem do tamanho do problema.

Sejam a,b e c constantes não negativas. A solução de recorrência T(n) é dada por:

$$T(n) = \begin{cases} b, & \text{para } n=1 \\ a.T(n/2)+bn, & \text{para } n>1. \end{cases}$$

para n uma potência de c é

$$T(n) = \begin{cases} o(n) & \text{se } a<c \\ o(n \log n) & \text{se } a=c \\ o(n^{\log_c a}) & \text{se } a>c. \end{cases}$$

Pode-se concluir da generalização acima que dividindo um problema em dois subproblemas a metade do tamanho resulta em dois subproblemas a metade do tamanho resulta em problemas da ordem n log n. Se o número de subproblemas for 3, 4 ou 8, então o algoritmo poderá ter ordem $n^{\log 3}$, n^2 ou n^3, respectivamente.

Concluindo, pode-se verificar que a rapidez da multiplicação de dois inteiros de n-bits, foi reduzida de $o(n^2)$ para $o(n^{\log 3})$. Mas esta não é a cota superior corrente. O melhor desempenho assintótico de tempo, para o produto de dois inteiros é de o(n.log n.log log n) como demonstrado por Shonhague e Strassem em [SHO71].

6.2.4 Multiplicação de matrizes

Considere, agora, o problema da multiplicação de duas matrizes A e B, ambas de dimensão nxn, resultando a matriz produto C. A matriz resultante C = AB é, por definição uma matriz nxn cujo elemento c_{ij}, da linha i e coluna j é obtido a partir da i-ésima linha de A e da j-ésima coluna de B, através do somatório:

$$c_{ij} = \sum_{k=1}^{n} a_{ik}.b_{kj} \quad . \quad (1 <= i,j <= n)$$

cada elemento c_{ij} requer n multiplicações e (n-1) adições e como, a matriz C possui n^2 destes elementos, são necessárias n^3 multiplicações para se obter C. Este algoritmo trivial para a multiplicação de matrizes tem, portanto, rapidez $o(n^3)$.

Muitas das operações matriciais básicas, como inversão de matrizes, cálculo de determinantes, estão diretamente relacionadas com a multiplicação de matrizes. A descoberta de um algoritmo eficiente para a multiplicação implica diretamente na rapidez assintótica para as demais operações.

Este problema também pode ser resolvido por um algoritmo baseado na técnica da divisão e conquista. Sabe-se até que a rapidez é da ordem de n elevado ao logaritmo de sete, o que aproximadamente $o(n^{2.81})$. A idéia do algoritmo consiste em particionar as matrizes A e B em quatro submatrizes quadradas, sendo cada uma delas com dimensão n/2xn/2 (esta sendo suposto que n seja uma potência de dois) conforme o esquema abaixo:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

onde $C_{ij} = A_{i1}.B_{1j} + A_{i2}.B_{2j}$ para i e j pertencentes a {1,2}

Se n=2, as submatrizes tem dimensão 1x1 e o produto AB pode ser calculado diretamente pela fórmula de cij, por multiplicação de elementos.

Para n>2, os cálculos dos cij necessitam de multiplicações de submatrizes n/2xn/2. Como n/2 também é uma potência de dois, a multiplicação das submatrizes pode ser feita pela aplicação recursiva do particionamento em quatro submatrizes (de dimensão n/4xn/4). Recursivamente, estes particionamentos sucessivos resultarão em vários casos de multiplicação de matrizes 2x2, que podem ser efetuados diretamente.

O algoritmo recursivo descrito acima requer oito multiplicações e quatro somas de matrizes n/2xn/2. A rapidez T(n) deste algoritmo pode ser expressa por,

$$T(n) = \begin{cases} a, & \text{Se } n=2 \\ 8T(n/2) + cn^2 & \text{Se } n>2 \quad \text{(sendo a e c constantes)} \end{cases}$$

o que tem por solução $o(n^3)$. As multiplicações cij podem ser substituídas por adições, dadas por:

$D_1 = A_{11} (B_{12}-B_{22})$
$D_2 = A_{22} (B_{21}-B_{11})$
$D_3 = (A_{11} +A_{12}) B_{22}$
$D_4 = (A_{21} +A_{22}) B_{11}$
$D_5 = (A_{11} +A_{22}) (B_{11} +B_{22})$
$D_6 = (A_{12} -A_{22}) (B_{21} +B_{22})$
$D_7 = (A_{21} -A_{11}) (B_{11} +B_{12})$

resultando nas novas fórmulas cij, dadas por:

$C_{11} = D_5 +D_2 -D_3 +D_6$
$C_{12} = D_1 +D_3$
$C_{21} = D_4 +D_2$
$D_{22} = D_5 +D_1 -D_4 +D_7$

o que resulta em uma rapidez T(n) expressa por

$$T(n) = \begin{cases} a, & \text{se } n=2 \\ 7T(n/2) +bn^2, & \text{se } n>2 \quad \text{(sendo a e b constantes)} \end{cases}$$

que tem por solução uma rapidez da ordem $n^{\log 7}$, o que é aproximadamente $o(n^{2.81})$.

Para um algoritmo ótimo paralelo, é necessário n^3 processadores de forma que sejam executadas n multiplicações em um único tempo, para todos os produtos internos cij simultâneos. Baseado no princípio da dupla recursividade, as n^2 somas de

produtos podem ser efetuadas com n.n/2 processadores pelo duplo adicionar em computações paralelas com as duplas adições paralelas das somas intermediárias recursivas, portanto isto garante que cada produto interno seja efetuado em $\lceil \log n \rceil$ passos de tempo, resultando numa complexidade de tempo $\lceil \log n + 1 \rceil$, ou seja da ordem log n para n^3 processadores.

6.3 Potencialidades da Divisão e conquista na paralelização

Para que um usuário possa se valer das instruções vetoriais de hardware ou do paralelismo, existem três alternativas, três formas, que são:

a) Recompilar o programa fonte desenvolvido e testado para máquinas sequenciais, utilizando um compilador que otimiza, vetoriza e paraleliza o código, trocando as instruções escalares em vetoriais e executando partes identificadas e marcadas pelo compilador em paralelo. Isto ocorre nos supercomputadores Convex, IBM 3090 e Cray;

b) Pode ser obtido um maior benefício do compilador se primeiro for reestruturado o código fonte sequencial de modo a auxiliar o compilador a reconhecer e marcar mais oportunidades para gerar código vetorial e paralelo. No IBM 3090, o compilador faz um diagnóstico do código fonte, aconselhando alterações para se ter um maior grau de vetorização e paralelização. Este diagnóstico leva em conta vários critérios, incluindo gastos com carga de operandos e tempo estimado paralelo e sequencial;

c) Pode ser recodificado completamente a aplicação pelo desenvolvimento de um novo algoritmo que utilize as vantagens da máquina paralela, independente da solução sequencial existente.

A técnica de divisão e conquista, quando empregada em observância ao princípio de equiparação, produz algoritmos sequenciais eficientes (muitas vezes até ótimos), como foi observado nos itens anteriores, especialmente nos exemplos de determinação do máximo e mínimo de uma lista linear, na ordenação de listas, multiplicação de inteiros e de matrizes.

O emprego desta técnica é de grande valia no processamento paralelo. Isto é verificado em exemplos práticos e na própria idéia do método, que subdivide o problema original em subproblemas independentes que podem ser resolvidos separadamente. Entretanto é importante perceber que esta técnica explora, principalmente, a paralelização a nível de execução e não a nível de projeto de desenvolvimento de algoritmos paralelos.

Isto significa que a técnica atinge, principalmente, as duas primeiras formas de paralelização e produzem um ganho de tempo de execução para solucionar o problema. Isto não quer dizer

que a técnica não seja utilizada no projeto de algoritmos paralelos, antes pelo contrário, pois são encontrados algoritmos paralelos baseados em divisão e conquista como por exemplo para seleção do k-ésimo elemento, descrito e analisado em [AKL89] e para várias áreas da análise numérica como referenciado por Duff em [DUF87].

Portanto, uma das razões da técnica de divisão e conquista ser muito útil na paralelização de algoritmos é devido à estrutura da técnica, que consiste da divisão do problema em k-subproblemas menores, os quais são resolvidos separadamente, e, muitas vezes em paralelo. Apesar da necessidade de comunicações de dados entre eles.

Para o sucesso do emprego desta técnica é necessário observar o princípio da equiparação na divisão do problema em subproblemas de tamanho aproximadamente iguais, minimizar a comunicação entre eles e que não se perca o tempo ganho na solução dos subproblemas em paralelo, na combinação ou na construção da solução global do problema.

Outra questão importante é o tipo de máquina paralela onde os algoritmos baseados em divisão e conquista tem um melhor desempenho. Segundo Zorat (em [ZOR83]) e Laira Toscani (em [TOS87]) são as máquinas que possuem uma arquitetura de árvore.

Segundo seu estudo de caso, em [TOS88], Laira Toscani analisa a técnica de divisão e conquista aplicada em algoritmos paralelos em arquiteturas de árvore. Neste estudo, ela conclui que a complexidade de um programa paralelo é independente do número de subproblemas que o problema é subdividido, o que não aconteceria no caso de algoritmos sequenciais. No caso paralelo, o que varia com o número de subproblemas é o número de processadores necessários, como no problema de ordenação por intercalação (OrdInter - algoritmo 6.2), onde m=c^k; e nos problemas de multiplicação de inteiros (Mult -algoritmo 6.3) e de matrizes onde m > c^k.

Outro resultado importante encontrado em [TOS88] foi que a complexidade do algoritmo de divisão e conquista, no caso em que o número de processadores não é suficiente para permitir a resolução do problema totalmente em paralelo, resulta na mesma complexidade dos algoritmos sequenciais, não havendo, portanto um ganho significativo na ordem de complexidade do algoritmo, mas apenas um ligeiro fator de aceleração no tempo de computação.

Por fim, ela comenta que, por não se conhecer as constantes que aparecem nas funções de recorrência primitivas e sendo elas diferentes nas versões sequencial e paralela (no qual dependem inclusive da tecnologia utilizada na construção da máquina paralela) de cada algoritmo, ser de pequena utilidade uma comparação mais detalhada da complexidade dos algoritmos sequenciais e paralelos.

Conclui-se que, se existir uma máquina com uma arquitetura adequada(estrutura de árvore) com processadores suficientes para que o problema seja resolvido totalmente em paralelo, quase sempre haverá um ganho na ordem de complexidade. Entretanto, para que problemas sejam resolvidos totalmente paralelizados, são necessários um grande número de processadores e, por outro lado, como a cada instante apenas um nível da árvore está ativo, apenas uma pequena fração dos processadores está ativo. Isto resulta na necessidade de uma máquina com muitos processadores ou numa nova filosofia, onde os processadores são realocados dinamicamente.

6.4 Exemplos de algoritmos paralelos

Como já visto, ordenação é definido como o problema de rearranjar uma sequência de valores em ordem crescente ou decrescente. O limite inferior para o problema (sequencial) é de o(n.log n) para ordenar n elementos.

Neste item é, inicialmente, apresentado o exemplo do algoritmo paralelo de ordenação por intercalação par-impar. Este algoritmo pertence a classe NC^2 e utiliza n log^2 n processadores, em tempo o(log^2 n) e é atribuído a Batcher ([BAT68]).

A seguir é apresentado o algoritmo paralelo de ordenação por intercalação bitônica, também atribuído a Batcher.

6.4.1 Ordenação por intercalação Par-Impar

A idéia básica do algoritmo é intercalar duas sequências A=(a1,a2,...,an) e B=(b1,b2,...,bn) já ordenadas recursivamente (por uma estratégia de divisão e conquista). Cada sequência possui n elementos, sendo m um inteiro e potência de dois.

Esta intercalação é feita via circuitos comparadores C. O circuito comparador C possui duas entradas a e b e duas saídas min e max. Por definição as saídas de C, min e max contém o mínimo e máximo de a e b, respectivamente.

No caso n=1, um comparador é suficiente para intercalação. O caso n=2, que é a base de recursão, necessita três comparadores: C, C' e D. São efetuadas duas etapas. Na primeira C e C' efetuam comparações em paralelo, com as entradas a(C)=a1, b(C)=b1 e a(C')=a2, b(C')=b2. O comparador C recebe os índices ímpares e o comparador C' os pares. Na segunda etapa, tem-se que as entradas de D são a(D)=max(C) e b(D)=min(C'). Por fim, tem-se que a sequência final, por definição, e1=min(c), e2=min(D), e3=max(D) e e4=max(C').

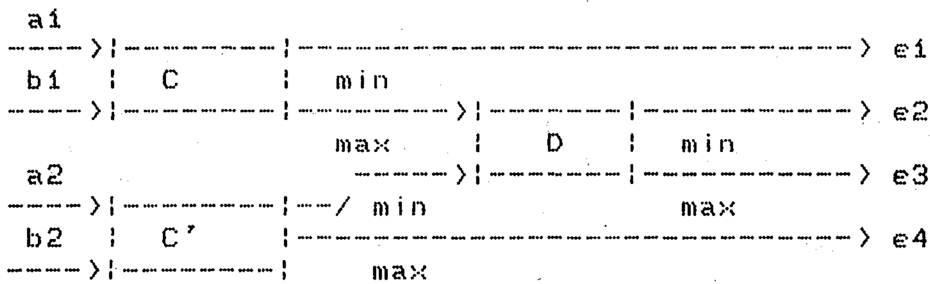


Fig. 6.5 Diagrama do método par-ímpar para n=2

No caso n>2, têm-se A=<a1,a2,...,an> e B=<b1,b2,...,bn> já ordenados recursivamente, faz-se a intercalação das duas para gerar a sequência final ordenada de 2n elementos E=<e1,e2,...,en>, a qual também é feita em duas etapas.

Etapa 1: Utilizando um circuito intercalador de n entradas e n saídas, os elementos de índice par de A e os de B são intercalados para se obter C=<c1,c2,...,cn>, em paralelo; os índices ímpar de A e os de B são intercalados utilizando uma cópia do mesmo circuito intercalador de n entradas e n saídas, obtendo D=<d1,d2,...,dn>.

Etapa 2: Após a obtenção de C e D em paralelo, obtem-se a sequência final E por:

$e_1 = c_1$
$e_{2i} = \min(c_{i+1}, d_i)$ para i=1,2,..,n-1
$e_{2i+1} = \max(c_{i+1}, d_i)$ para i=1,2,..,n-1
$e_{2n} = d_n$

Para melhor ilustrar o algoritmo de intercalação Par-Ímpar, é apresentado na figura 6.6 o circuito para ordenar a sequência s=(8,7,6,5,4,3,2,1).

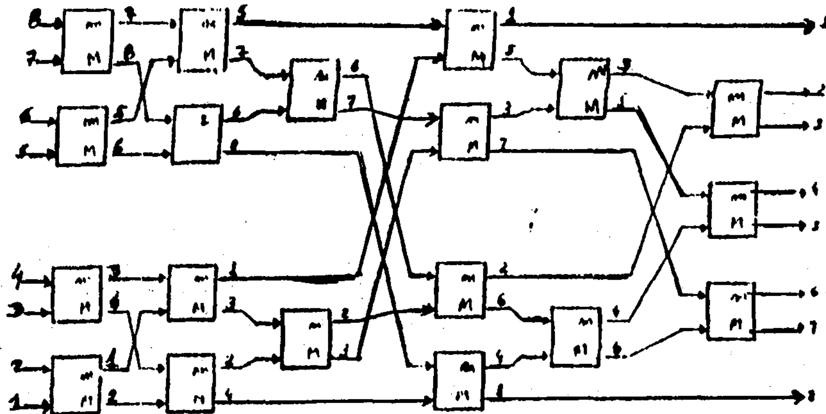


Fig 6.6 Circuito para ordenar a sequência S

Os cálculos de e2i e e2i+1 acima são feitos por n-1 circuitos comparadores. A sequência ordenada com os elementos de A e B, pode ser conseguida através da construção de uma concatenação de circuitos intercaladores, I2, I4,...,I2^k, onde

I2 é constituído por n intercaladores de duas entradas, em paralelo, que recebem 2n elementos de entrada e geram n sequências ordenadas com dois elementos cada uma;

I4 é constituído por n/2 intercaladores de quatro entradas, em paralelo, que recebem as saídas de I2 e geram n/2 sequências ordenadas com quatro elementos cada uma;

e assim por diante, obtendo-se sequências ordenadas cada vez mais longas, até que na saída de I2^k, onde 2^k = 2n (I2^k é constituido de um só circuito intercalador de 2n entradas) obtém-se uma só sequência ordenada com 2n elementos.

Se T(.) denota o número de comparadores em I2^k, tem-se a equação de recorrência:

$$T(2^k) = 2T(2^{(k-1)}) + 2^{(k-1)} - 1, \quad \text{para } k>1,$$

a qual tem por solução: $T(2^k) = (k-1)2^{(k-1)} + 1$.

O total de camparadores no circuito ordenador é dado pela soma de n cópias de T(2) em I2; n/2 cópias de T(4) em I4 e assim por diante, até uma cópia de T(2^K) em I2^k, o que resulta em $2^{(k-2)}(k^2 - k + 4) - 1$. Portanto, como $2^k = 2n$ o número total de comparadores é da ordem de $n \log^2 n$.

6.4.2 Ordenação por intercalação Bitônica

Uma sequência S=(a1,a2,...,a2n) é dita ser **bitônica** se existe um inteiro j (1 <= j <= 2n), tal que:

$$a_1 \le a_2 \le \dots \le a_j \ge a_{j+1} \ge a_{j+2} \ge \dots \ge a_{2n},$$

ou a sequência S não satisfaz a condição inicialmente, mas pode ser deslocada ciclicamente até que satisfaça a condição.

Pode-se ordenar uma sequência bitônica S=(a1,...,a2n) em ordem crescente em duas etapas:

ETAPA 1: Usando n comparadores, cria-se duas sequências conforme definido a seguir,
min(a1,an+1), min(a2,an+2),...,min(an,a2n)
max(a1,an+1), max(a2,an+2),...,max(an,a2n)

ETAPA 2: Cada uma dessas duas sequências, se forem bitônicas, podem ser ordenadas recursivamente pelo mesmo processo.

Como nenhum elemento da primeira subsequência é maior que qualquer elemento da segunda, a primeira subsequência produzirá os n primeiros elementos da sequência ordenada e a segunda subsequência produzirá os n últimos elementos.

A figura 6.7 mostra o intercalador bitônico para uma sequência S={a1,a2,a3,a4} (com quatro elementos). Na figura m indica o mínimo e M o máximo.

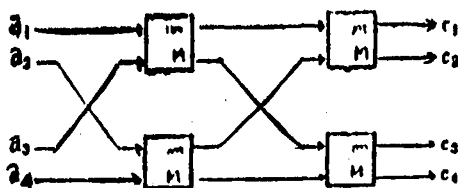


Fig 6.7 Intercalador bitônico para uma sequência com 4 elementos

Na figura 6.8 temos um intercalador bitônico para uma sequência com 2n elementos.

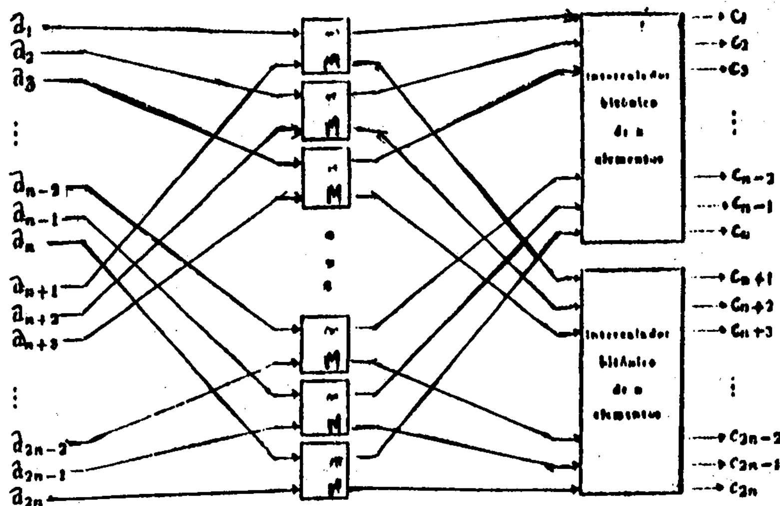


Fig.6.8 Intercalador Bitônico (m=min, M=max)

A figura 6.9 apresenta o circuito para ordenar a sequência S={4,8,1,3,2,7,5,6}. Observa-se que para produzir a parte decrescente da sequência bitônica as linhas de saída de alguns comparadores foram invertidas.

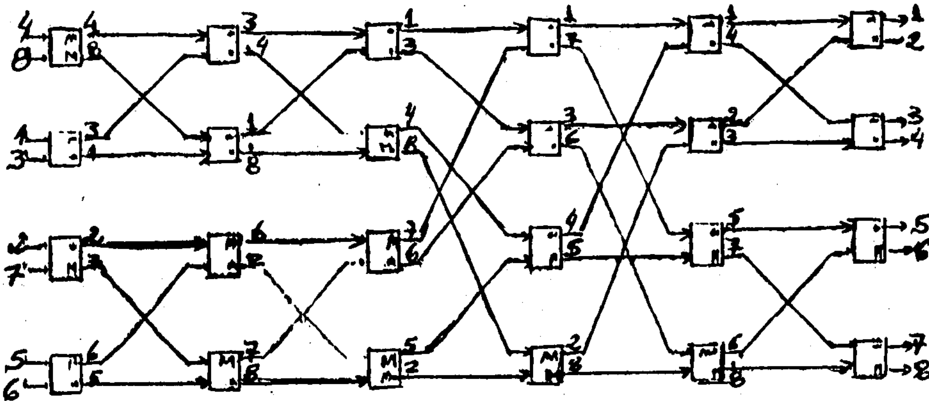


Fig.6.9 Circuito para ordenar {4,8,1,3,2,7,5,6}

O número total de comparadores e de comparações em paralelo necessários para ordenar uma sequência com n elementos (sendo $n=2^m$), utilizando ordenação bitônica é dado pela relação de recorrência,

$$\begin{cases} q(2)=1 & \text{para } i=1 \\ q(2^i) = 2^{(i-1)} + 2q(2^{(i-1)}) & \text{para } i>1 \end{cases}$$

a qual resolvida, determina que o número total de comparadores necessários para ordenar uma sequência com $n = 2^m$ elementos, $2^{(m-1)} \cdot m(m+1)/2$, ou seja, o número de comparadores necessários é da ordem de $n.\log^2 n$.

Observa-se que tem a mesma ordem de complexidade que o algoritmo anterior, entretanto, este algoritmo pode ser aperfeiçoado pela técnica do **embaralhamento perfeito**, através de processadores interconectados, resultando num algoritmo para ordenação onde o custo é menor.

A figura 6.10 consiste de um resumo da comparação destes algoritmos de ordenação por intercalação, quanto ao número de processadores e quanto a complexidade de tempo.

ALGORITMO	PROCESSADORES	COMP TEMPO
Par-Impar	n log^2 n	o(log^2 n)
Bitonico	n log^2 n	o(log^2 n)
Embar.Perf	n/2	o(log^2 n)

Fig.6.10 Comparação das complexidade

7 CONCLUSÕES

7.1 Considerações finais

Este relatório é o resultado do estudo de um exame de qualificação de doutorado do Curso de Pós-Graduação em Ciência da Computação da UFRGS. Achou-se importante documentar não só as conclusões, mas também pensamentos ou propostas de estudos, que surgiram durante o desenvolvimento deste trabalho, mas que vão além do objetivo e da sua abrangência.

Antes de entrar nas conclusões cabe ressaltar a dificuldade encontrada no estudo de algoritmos paralelos e de complexidade de algoritmos, uma vez que no PGCC da UFRGS não existe tradição e "knowhow" de pesquisa nestas áreas; não se tem ainda máquinas paralelas instaladas; livros sobre o assunto são escassos e o assunto envolve não só mudança na forma de pensar - paralelismo, como também, envolve problemas matemáticos bastante complexos.

Com este trabalho se pretendeu aprofundar os conhecimentos sobre projeto de desenvolvimento de algoritmos paralelos, em especial, o uso da técnica de divisão e conquista.

Para tanto, foi necessário a caracterização do paralelismo, dos tipos de arquiteturas e máquinas paralelas e de metodologias de obtenção de algoritmos paralelos.

Foram necessários, também, estudos que abrangeram a análise e a complexidade de algoritmos sequenciais e paralelos; as classes de complexidade de tempo e espaço e seu relacionamento. Estes estudos serviram para introduzir o leitor ao ambiente de análise e projeto de algoritmos paralelos.

Uma das considerações importantes, que merece ser recordada aqui, é o fato de que o desenvolvimento de algoritmos paralelos depende do problema a ser resolvido, do número de processadores disponíveis para a resolução do problema e do modelo computacional ou da arquitetura onde o problema será resolvido.

Dentro deste ambiente e desta percepção do projeto de desenvolvimento de algoritmos paralelos, formalizou-se a técnica de divisão e conquista e seus princípios para que seja utilizada no desenvolvimento de algoritmos de uma forma eficiente, gerando algoritmos ótimos, do ponto de vista de rapidez ou complexidade de tempo.

Acredita-se que com este trabalho, conseguiu-se abrir um caminho para compreensão do projeto de desenvolvimento e análise de algoritmos paralelos. Este trabalho sintetiza uma grande quantidade de idéias e conceitos e pretende proporcionar opções para a continuidade deste estudo. Algumas destas opções foram formalizadas e propostas no item 7.3 propostas para novos estudos.

7.2 Constatações

Verificou-se que os algoritmos baseados na técnica de divisão e conquista não acrescentam grandes progressos na ordem de complexidade de tempo para a solução de problemas, mas sim um ganho no tempo de processamento, no tempo de execução do programa para solucionar o problema.

Para que algoritmos baseados em divisão e conquista produzam um ganho na ordem de complexidade, é necessário que a máquina tenha uma arquitetura adequada à técnica, ou seja, uma estrutura de árvore. É necessário ainda, que existam processadores suficientes para que o problema seja executado totalmente em paralelo. Isto significa dizer que o problema é dividido em subproblemas até o nível de "problemas simples", os quais são resolvidos diretamente. Este nível corresponderia as folhas da árvore, onde todos os subproblemas seriam resolvidos ao mesmo tempo, em paralelo. Ora, isto implica que, como a cada passo somente um nível da árvore está ativo, que muitos processadores estariam ociosos grande parte do tempo, em que o problema estaria sendo resolvido. Nestes moldes, a máquina seria ineficiente e cara. Estudos vem sendo desenvolvidos para definir uma arquitetura de árvore, com processadores realocados dinamicamente.

Verificou-se, ainda, que a paralelização dos algoritmos baseados na divisão e conquista atinge mais o nível de execução do que o nível de projeto de desenvolvimento de algoritmos paralelos, isto significa, que algoritmos sequenciais baseados em divisão e conquista, podem ser executados em paralelo ganhando tempo de execução, de processamento.

Apesar disto, Duff (em [DUF87]) apresenta referências de vários trabalhos onde citam e abrangem algoritmos baseados em divisão e conquista e dupla-recursividade (um caso particular de divisão e conquista) para várias áreas não numéricas e numéricas, ou, ainda, na bibliografia consultada existem exemplos destes algoritmos.

Nas caracterizações de modelos computacionais estudados, encontrou-se alguns baseados em características ideais ou utópicas, sobre os quais foram e são feitas análises dos algoritmos. Entre estas características utópicas ou ideais, está a suposição do paralelismo livre, no qual existem infinitos processadores disponíveis; o problema pode ser representado por uma sequência de operações lógicas e aritméticas binárias; cada operação pode ser executada em uma unidade de tempo; cada processador poder executar qualquer operação a qualquer momento; os demais processos envolvidos (como transferência de dados e controle de processamento) na solução do problema não requerem nenhum tempo para serem executados e não existir nenhum conflito de acesso aos dados.

Na prática vê-se que as análises baseadas em tais modelos, levam a resultados que não correspondem à realidade, pois como foi visto no capítulo quatro, a complexidade de algoritmos paralelos não depende apenas da quantidade de trabalho necessário para resolver o problema, mas também do número de processadores disponíveis e do modelo de computação, da arquitetura da máquina e da forma como é feita a comunicação entre os processadores.

Quando se estuda complexidade de algoritmos sequenciais, logo são identificadas as medidas de complexidade, onde o tempo e o espaço são as principais. A quantificação destas medidas é feita através de um formalismo, que procura descrever o relacionamento do tamanho do problema ao tempo necessário para resolver. Este relacionamento é feito através de funções, denominadas de funções de complexidade de problemas, que calculam o número de operações elementares necessárias para resolver o problema, independente da máquina. Para tanto, usa-se a notação de ordem, ou o conceito de ordem máxima(mínima ou exata). Esta idéia tem sido utilizada para descrever a complexidade de algoritmos paralelos, apesar desta depender dos modelos computacionais (ou máquinas) e do número de processadores disponíveis. Portanto, há a necessidade de se criar uma notação mais expressiva, mais eficiente para se expressar esta complexidade.

Devido, ainda, a esta dependência da complexidade de algoritmos paralelos a fatores como quantidade de trabalho, arquitetura/modelo computacional e número de processadores; tem-se para cada tipo de máquina(teórica) uma cota superior de complexidade. Um trabalho interessante a ser feito, é identificar os principais modelos computacionais (modelos nos quais as máquinas reais se baseiam), e determinar a cota superior para cada problema em cada modelo; determinar qual o número ótimo de processadores para resolver cada problema em cada modelo e a implicação da existência de um número menor na complexidade do algoritmo paralelo.

As considerações anteriores implicam na criação de um formalismo capaz de expressar a complexidade de algoritmos paralelos em função dos seguintes parâmetros: tamanho do problema, tipo de máquina, número de processadores; e que expresse a quantidade de trabalho necessário na máquina com o determinado número de processadores, de forma que variando os parâmetros, seja identificado o efeito da mesma na complexidade.

Sabe-se que o primeiro supercomputador foi instalado em 1976 e, a partir de então, com o aumento da demanda e disponibilidade de máquinas paralelas, o desenvolvimento de algoritmos paralelos passou do campo teórico para o prático, resultando numa grande quantidade de diferentes algoritmos paralelos para a solução de problemas, em vários tipos de máquinas paralelas. Com isto surgiram algumas questões, que nos levam a refletir sobre o assunto.

a) Quais as Classes de problemas que são resolvidos de forma eficiente em cada máquina?;

b) Qual o melhor tipo de algoritmo para cada máquina e qual sua complexidade?;

c) Para um determinado problema, qual a cota superior de complexidade dos algoritmos em cada máquina e em qual delas o problema é mais eficiente. Quão próximo está esta cota superior da cota inerente do problema e qual a relação entre elas?;

d) Qual a variação que resulta na complexidade do algoritmo paralelo uma variação do número de processadores e qual o número ideal de processadores para resolver o problema?

Por fim, sugere-se que sejam analisadas as soluções existentes para as máquinas, com o objetivo de identificar as técnicas de paralelização utilizadas, para que se possa desenvolver algoritmos paralelos eficientes.

7.3 Propostas para novos estudos

Neste item são apresentados algumas propostas de estudo decorrentes deste trabalho. Alguns destes estudos chegaram a ser iniciados e o resultado parcial encontra-se documentado no anexo B Considerações sobre algoritmos numéricos paralelos.

Equações de recorrência e a determinação da cota superior de problemas

Estudar a solução de equações de recorrência e sua importância na determinação da cota superior de algoritmos, uma vez que elas são uma das principais metodologias para a determinação de limites superiores. Este estudo deve identificar e caracterizar os principais problemas abordados e determinar a equação de recorrência do algoritmo.

Bibliografia: [AKL89], [KRO86], [WEI77]

Linguagens e notações de algoritmos paralelos e distribuídos

Identificar abordagens das linguagens vetoriais e paralelas, as quais se caracterizam pelo compilador identificando comandos vetorizáveis e paralelizáveis e pelo programador utilizando comandos paralelos da linguagem. O estudo deveria caracterizar as principais linguagens paralelas e distribuídas, identificando os principais conceitos de comandos paralelos para cada tipo de máquina paralela.

Bibliografia: [PER79], [PER90], [HOL81], [KRO86]

Problemas NP-Completos e Redução polinomial

Caracterizar os problemas NP-Completos; Definir a redução polinomial; demonstrar a equivalência de problemas e considerações sobre a questão em aberto: P = NP ? Identificar os problemas inerentemente sequenciais (P-completos), caracterizá-los e descrever a forma de abordagem na tentativa de solução. Fazer considerações sobre a questão em aberto: P=NC?

Bibliografia: [AHO74], [CAR88], [PIP79a], [TRA76b], [TER90], [TER91], [MIY88], [KRO86], [WEI77].

Algoritmos paralelos iterativos

Estudar a grande classe de problemas resolvidos por métodos iterativos e as vantagens e consequências da paralelização; Identificar modelos de métodos iterativos paralelos e o uso de algoritmos sincronos e assincronos; determinar a ordem de convergência dos algoritmos iterativos paralelos e critérios de parada que podem ser empregados.

Bibliografia: [AKL89], [DUF87], [HOL81], [JAM85], [SAM77], [KRO86]

Problemas de comparação

Os problemas de comparação consistem em problemas de seleção, ordenação por intercalação e concatenação e pesquisa. O estudo teria por objetivo identificar as cotas inferiores dos problemas e os melhores algoritmos sequenciais que resolvem estes problemas. Estudar algoritmos paralelos para estes problemas e compará-los com os sequenciais.

Bibliografia: [AKL89], [BAT68], [CAR88], [KRO86]

Operações com matrizes - algoritmos paralelos

Problemas envolvendo matrizes aparecem em vários contextos numéricos e não numéricos. Exemplos do uso de matrizes são encontrados na solução de sistemas e na representação de grafos.

Neste estudo, deveria caracterizar operações com matrizes que são computadas em paralelo, como cálculo da matriz transposta, multiplicações entre matrizes e matrizes com vetores, bem como o cálculo da matriz inversa. Este estudo deve identificar, para cada operação, a versão do algoritmo mais eficiente para cada arquitetura.

Bibliografia: [AKL89], [DUF87], [GEN79], [HEL78], [MOD88], [KRO86], [TER90], [TER91], [AHO74]

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

Resolução de Problemas numéricos -algoritmos paralelos

Em aplicações científicas e da engenharia que se utilizam de computadores para resolver problemas matemáticos. Elas abrangem um grande intervalo de aplicações, como modelagem da atmosfera para previsão do tempo, para modelar o plasma na física teórica, para projetar estações espaciais e aeronaves, para controlar tráfego aéreo entre outros. Nestas aplicações computadores são utilizados para calcular zeros de funções (raízes de polinômios), resolver sistemas de equações, calcular autovalores e resolver equações diferenciais.

Portanto, se faz necessário o estudo da resolução de problemas numéricos via algoritmos paralelos e do impacto da computação vetorial e paralela sobre tais problemas.

Outra questão importante a ser estudada é que algoritmos numéricos envolvem um grande número de operações elementares, ocasionando problemas de erro de arredondamento e de sua propagação, além de questões de estabilidade ou instabilidade numérica.

Entre as vantagens do processamento paralelo estão: o aumento da velocidade de processamento; a possibilidade de resolução de problemas muito complexos para máquinas sequenciais e a solução de problemas de natureza paralela.

Os problemas computacionais mais comuns em álgebra linear são a solução de equações lineares e o problema algébrico de autovalores para vários tipos de matrizes. O problema das características afetam qualquer implementação computacional, incluindo matrizes quadradas ou retangulares; simétricas ou não; densas, esparsas ou tridiagonais; explicitamente representada pelas suas entradas ou implicitamente representada pelas suas ações em vetores. Tais propriedades determinam qual algoritmo ou família de algoritmo é apropriada para resolver o problema em qual ambiente computacional, sequencial ou paralelo, podendo sofrer maior impacto no ambiente paralelo. Por exemplo, a necessidade de linhas ou colunas intercambiarem-se para estabilidade numérica pode ser uma complicação mais séria ao ambiente paralelo do que ao sequencial. Outra questão importante é o tamanho da matriz, o que determina a quantidade de recursos computacionais que serão necessários, assim como que classes de algoritmos e estrutura de dados deve ser mais apropriada.

Os algoritmos paralelos para solução de sistemas de equações lineares podem ser por métodos diretos ou iterativos.

Fatoração de matrizes densas é um caso interessante de estudo para implementação paralela devido ao fato de existirem duas características que tendem a inibir a eficiência paralela. Elas são: restrições de precedência sequenciais, devido a sucessivas linhas, colunas e submatrizes da matriz fatorada,

necessitam ser calculadas em ordem consecutiva e; comunicação global ser requerida porque cada sucessiva linha, coluna ou submatriz da matriz fatorada depender de todas precedentes linhas, colunas ou submatrizes.

Métodos iterativos para solução de sistemas de equações lineares também se apresentam como um caso interessante de estudo para implementação em paralelo. Eles diferem dos métodos diretos, nos quais um conjunto fixo de cálculos precisam ser realizados em ordem para chegar a solução correta, a natureza autocorretiva dos métodos iterativos permite que se altere os cálculos para herdar paralelismos. O speedup de iterações individuais pelo acrescentar da concorrência não se beneficia, entretanto, facilita o cálculo da próxima iteração.

Métodos iterativos são frequentemente recomendados para implementações em paralelo por limitarem mais os requisitos de comunicação, em relação a comunicação global necessária em métodos diretos por fatoração.

Métodos para solução de equações diferenciais parciais (PDE) podem ser classificados em métodos diretos e métodos iterativos. Métodos diretos resolvem PDEs analiticamente; métodos iterativos iniciam com valores estimados em certa localização específica e, então, convergem para estimativas mais exatas. Algoritmos iterativos para resolver PDEs podem consumir um tempo de processamento bastante grande, por isto é interessante olhar para computações paralelas. Quando a utilidade de um método iterativo é verificada, o intervalo de convergência dos valores para se ter a solução é de grande importância.

A resolução de PDEs pode ser feita através de algoritmos paralelos baseados no uso de três rotinas: particionamento, discretização e iteração.

Particionamento - As variáveis no problema são divididas em grupos. O critério de agrupamento pode ser geométrico, ou uma propriedade da matriz, ou uma propriedade física. Diferentes particionamentos podem ser usados em diferentes vezes e partições podem ser mais particionadas. Particionamento geralmente conduz ao uso de paralelismo.

Discretização - O problema contínuo PDE é substituído por um problema finito com um conjunto de números reais como variáveis. As duas técnicas mais comuns são diferenças finitas e funções bases (elementos finitos, polinômios e expansões em séries). Métodos de diferenças finitas e funções bases com suporte local são muito bons para métodos paralelos na fase de discretização, por serem esses cálculos altamente independentes uns dos outros. Discretizações são também divididos em métodos de alta ordem e de baixa ordem. Métodos de alta ordem são vantajosos para paralelização pois mais de um trabalho pode ser feito na fase de discretização dos cálculos.

Iteração - Iterações é uma técnica de propósito geral que têm dificuldades (não linearidade, problemas muito grandes, dependência de tempo). Iteração é inerentemente sequencial e portanto, não é muito propícia para exploração de paralelismo. Ainda que iteração seja essencial para resolver muitos PDEs, é bom só introduzi-la quando o número de iterações for pequeno e o trabalho em cada iteração bem grande e que se possa dividi-lo em componentes paralelos.

Bibliografia: [AKL89], [DUF87], [GAL77], [MAE87], [SAM77], [SHE67], [KRO86], [VOI85], [CAP85], [HEL78], [RUM90].

BIBLIOGRAFIA

[AHO74] AHO,A.V.; HOPCROFT,J.E.; ULLMAN,J.D. **The design and analysis of computer algorithms.** Reading: Addilson Wesley, 1974.

[AKL89] AKL, S.G. **The design and analysis of parallel algorithms.** Englewood Cliffs: Prentice Hall, 1989. 401p.

[ALL83] ALLEN, J.R. **Dependence analysis for subscripted variables and its applications to program transformations.** Houston: Rice University, 1983.(Ph.D dissertation).

[BAT68] BATCHER,K.E. Sorting networks and theirs applications. In: AFIP SPRING JOINT COMPUTER CONFERENCE, Apr.30-May 2, 1968, Atlantic City. **Proceedings...** Montvale: AFIP Press, 1968. p.307-314.

[BAE80] BAER,J.L. **Computer Systems Architecture.** Maryland: Potomak, 1980.

[BER89] BERTSEKAS,D.P; TSITSIKLIS,J.V. **Parallel and distributed computations numerical methods.** Englewood Cliff: Prentice Hall, 1989.

[BON91] BONDELI,S. Divide and conquer: a parallel algorithm for the solution of a tridiagonal linear system of equations. **Parallel computing**, Amsterdam, v.17,n.4&5, p.419-434, July, 1991.

[CAP85] CAPPELLO, P.R. A mesh automation for solve dense linear systems. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, St Charles, Aug. 20-23, 1985 **Proceedings.** New York, IEEE,1985. p.418-425.

[CAR89] CARISSIMI,A.S. **Implementação de arquitetura SIMD.** Porto Alegre, PGCC da UFRGS, 1989. (dissertação de mestrado)

[CAR88] CARVALHO,M. **Introdução a algoritmos paralelos e suas complexidades.** São Paulo: IME-USP, 1988.70p. (dissertação de mestrado).

[DEM82] DEMINET,J. Experience with multiprocessor algorithms. **IEEE Transactions on Computer**, New York, v.C-31, n.4, p.278-288, Apr. 1982.

[DIV91] DIVERIO,T.A. **Processamento vetorial e vetorização na máquina CONVEX C210.** Porto Alegre: PGCC da UFRGS, 1991. RP.160. 100p.

[DUF87] DUFF,I.S The influence of vector and parallel processors on numerical analysis. In: THE STATE OF THE ART IN NUMERICAL ANALYSIS. Oxford: Clarendon Press, 1987.

[ENS74] ENSLOW,P.H. **Multiprocessors and Parallel Processing.** New York: Wiley-Intersciense, 1974.

[FLY72] FLYNN,M.J. Some organizations and their effectiveness. **IEEE Transactions Computer.** New York,v.C-21,n.9, p.948-960, Sep, 1972.

[GAL77] GAL,S.; MIRANKER, W.L. Optimal sequential and parallel seach for finding a root. **Journal of Combinatorial Theory (A)**, v.23, p.1-14, 1977.

[GEN79] GENTLEMAN,W.M. Some complexity results for matrix computations on parallel processor. **Journal ACM**,New York, v.25, n.1, p.112. 1978.

[HAY78] HAYES,J.P. **Computer architecture and organization.** New York: McGraw Hill, 1978.

[HEL78] HELLER,D. A survey of parallel algorithms in numerical linear algebra. **SIAM Review**, New York, v.20, n.4, p.740-777, Oct, 1978.

[HOC81] HOCKNEY,R.W.;JESSHOPE,C.R. **Parallel computer: architecture, programmming and algorithms.** Bristol: Adam Hilger, 1981.

[HOS84] HOSSFELD,F. Parallel algorithms: the impact of communication complexity. In: COLLOQUIA MATHEMATICS SOCIETATIS JANOS BOLYAI. Pecs, Hwngary, 1984. p.207-232.

[HWA84] HWANG,K; BRIGGS,F.A. **Computer architecture and parallel processing.** New York: McGraw Hill, 1984.

[JAM85] JAMIESON,L; GANNON,D; DOUGLASS,R. **The Characteristics of Parallel Algorithms.** Cambridge: Mit Press, 1985.

[KNU81] KNUTH,D.E.; GREENE,D.H. **Mathematics for the analysis of algorithms.** Boston, Birkhauser, 1981.

[KOO81] KOOGGE,P.M. **The Architecture of Pipelined Computers.** New York, McGraw Hill, 1981.

[KRO86] KRONJO,L. **Computational compexity of sequential and parallel algorithms.** Chichester: John Wiley, 1986.

[KUC78] KUCK,D.J. **The structure of computer and computations.** New York: John Wiley, 1978. v.1.

[KUN76] KUNG,H.T. Synchronized and asynchronous parallel algorithms for multiprocess. In: SYMPOSIUM ON NEW DIRECTIONS AND RECENT RESULTS IN ALGORIHMS AND COMPLEXITY, 1976, Carnegie-Mellon University. **Algorithms and complexity.** New York: Academic Press, 1976. 523p. p.153-200.

[KUN80] KUNG,H.T. The structure of parallel algorithms. **Advances in Computers.** New York, v.19, p.65-112, 1980.

[LAM74] LAMPORT,L. The parallel executor of DO loops. **Communications of the ACM**,New York, V.17, N.2, fev.1974.

[LEE88] LEE,P.; KEDEM,Z.M. Synthesizing linear array algorithms from mested for loop algorithms. **IEEE Transactions on Computers.** New York, Dez.1988.

[LIN81] LINT,B.; AGERWALA, I. Communications Issues in the design and analysis of parallel algorithms. **IEEE Transactions on Software Engineering.** V. SE 7, N.2, p.174-188, Mar, 1981.

[MAE87] MAEDER,A.J; WYNTON,S.A. Some parallel methods for polynomial root-finding. **Journal of Computational and applied Mathematics**, Antwerp, p.71-81, 1987.

[MIY88] MIYANO,S. Parallel complexity and P-complete problems. In: INTERNATIONAL CONFERENCE ON FIFTH GENERATION COMPUTER SYSTEMS, 1988. **Proceedings** New York, ICOT, 1988. p. 532-541.

[MOD88] MODI,J.J **Parallel algorithms and matrix computations.** New York: Oxford, 1988.

[MOR90] MORKAZEL,F.C.;PANETTA,J. Representação automática de programas sequenciais para processamento paralelo. In: SIMPOSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES E PROCESSAMENTO PARALELO, 3, 1990, nov.7-9, Rio de Janeiro,**Anais...** Rio de Janeiro: SBC/PUC-RJ, 1990. 340p. p.5.B.1.1-7.

[NAV90] NAVAUX, P.O.A **Processadores Pipeline e processamento vetorial.** Escola de Computação, VII,USP São Paulo, 12-20, Jul, 1990.

[PER79] PERROTT,R.H. A language for array and vector processing. **ACM Transaction on programming languages and systems.** New York, 1(2): 177-195, Oct, 1979.

[PER90] PERROTT,R.H. Parallel languages and parallel programming. **Parallel computing.** New York, v. ,n. ,p.47-58, 1990.

[PET81] PETERS,F Tree machines and divide and conquer algorithms. **Lecture Notes in computer science.** New York, v.111, p.25-35, 1981.

[PIP79a] PIPPENGER,N; FISCHER,M.J. Relations among complexity measures. **Journal ACM**, New York,v.26, n.2, p.361-381, 1979.

[PIP79b] PIPPENGER,N. On simultaneous resource bounds. In: SYMPOSIUM ON FUNDATIONS OF COMPUTER SCIENCE, 20, Apr. 1979, Los Angeles. **Proceedings...** Los Angeles: IEEE Computer Society, 1979. p.307-311.

[QUI88] QUINN, J.M. **Designing efficient algorithms for parallel computers.** New York: McGraw Hill, 1988.

[RAB91] RABHI,F.A; MANSONG,G.A Divide and conquer and parallel graph reduction. **Parallel computing**, Amsterdam, v.17, n.2&3, p.189-205, Apr, 1991.

[RAM83] RAMAKRISHNAN,I.V; BROWNE,J.C. A paradigm for the design of parallel algorithms with applications. **IEEE Transactions Software Engineering**, New York, v.SE-9, n.4, p.411-415, July, 1983.

[RAM86] RAMAKRISHNAN,I.V et alli. Mapping homogenesus graphs on linear array. **IEEE Transactions on Computers**, New York, Mar.1986.

[ROD80] RODRIQUE,G.;GIROUX,E.D.;PRATT,M. Pespective on large-scale scientif computations. **Computer**, New York, v.13, n.10, p.65-80, Oct, 1980.

[RUM90] RUMP,S.M Rigorous sensivity analysis for systems of linear and nonlinear equations. **Mathematics of Computation**, Providence, v.54, n.190, p.721-736, Apr, 1990.

[SAM77] SAMEH,A.H; **Numerical Parallel algorithms -a survey.** In: HIGH SPEED COMPUTER AND ALGORITHM ORGANIZATION. New York: Academic Press, 1977. p.207-228.

[SCH84] SCHENDEL, U. **Introduction to numerical methods for parallel computers.** Chichester: Ellis Horwood, 1984.

[SHE67] SHEDLER, G.S. Parallel numerical methods for the solution of equations. **Communications ACM**, New York, v.10, n.1, p.286-291, 1967.

[STO80] STONE,H.S.et al. **Introduction to computer architecture.** Chicago: Sciense Research Associations, 1980.

[SUG80] SUGARMAN,R. Superpower computer. **IEEE Spectrum**, New York, v.17, n.4, p.28-34, Apr, 1980.

[TER90] TERADA,R. Introdução à complexidade de algoritmos paralelos. In: ESCOLA DE COMPUTAÇÃO, VII,1990, São Paulo, **Curso**. São Paulo, IME-USP, 1990. 104p.

[TER91] TERADA,R. **Desenvolvimento de algoritmos e estruturas de dados.** São Paulo: McGraw Hill-Makron, 1991.

[TOS87] TOSCANI, L.V Análise da complexidade de algoritmo em arquiteturas paralelas. Estudo de caso: a técnica de divisão e conquista. **Pesquisa operacional**, v.7, n.2, p.66-86, dez, 1987.

[TOS88] TOSCANI,L.V. **Métodos de desenvolvimento de algoritmos: especificação formal, análise comparativa e de complexidade.** Rio de Janeiro, PUC RJ, 1988. (tese doutorado).

[TRA73] TRAUB,J.F. **Complexity of sequential and parallel numerical algorithms.** New York: Academic Press, 1973.

[TRA76a] TRAUB,J.F **Analytic computational complexity.** New York: Academic Press, 1976, 239p.

[TRA76b] TRAUB,J.F. **Algorithms and complexity- new directions and recents results.** New York: Academic Press, 1976.

[TRA80] TRAUB,J.F.; WOZNIAKOWSKI,H. **A general theory of optimal algorithms.** New York: Academic Press, 1980.

[VOI85] VOIGT,R.G.; ORTEGA,J.M. Solution of partial differential equations on vector and parallel computers. **SIAM REVIEW**, New York, v.27, n.2, p.149-240, June, 1985.

[WEI77] WEIDE,B. A survey of analysis Techniques for discrete algorithms. **Computing Surveys.** New York, v.9, n.4,p.291-313, Dec, 1977.

[ZOR83] ZORAT,A.; HOROWITZ,E. Divide and conquer for parallel processing. **IEEE Transaction on Computers**, New York, v.C-32, p.582-585, 1983.

ANEXO A

REVISÃO MATEMÁTICA

LOGARITMO

Para todo x maior que zero ($x > 0$), logaritmo na base b de x é um número y tal que b elevado na y seja igual a x. Na linguagem matemática, temos:

$$(\forall x \in \Re \ \ e \ \ x > 0) \ \ log_b x = y \iff b^y = x$$

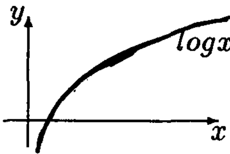
Um logaritmo na base dois, binário é definido por:

$$log_2 x = y \iff 2^y = x$$

para facilitar a notação, será adotada a convenção que o logaritmo na base 2, será anotado por:

$$log_2 x = logx$$

Cabe ainda relembrar o gráfico da função logarítmica



- $(\forall x \in \Re, x > 0)$
- $log1 = 0$, pois $2^0 = 1$
- o logaritmo de zero não é definido pois não existe valor k que elevado a dois seja zero. $[(\nexists k \in \Re)/\ 2^k = 0)]$

Propriedades:

(i) Se $x_1 > x_2$ então $logx_1 > logx_2$ – função extritamente crescente;

(ii) Se $logx_1 = logx_2$, então $x_1 = x_2$.
– função ionivoca. (1 para 1)

(iii) $log1 = 0$, pois $2_0 = 1$;

(iv) $log_b b^a = a$, particularmente $log2^a = a$;

(v) $log_b x = \frac{logx}{logb} \Longrightarrow logx = log_b x \,.\, logb$

(vi) $log(x_1.x_2) = logx_1 + logx_2$;

(vii) $log(x_1/x_2) = logx_1 - logx_2$;

(viii) $logx^a = alogx$;

(ix) $x_1^{logx_2} = x_2^{logx_1}$;

(x) Se $n = 2^k$, $logn = k$;

(xi) Se $2^k < n < 2^{k+1}$, $\lfloor logn \rfloor = k e \lceil logn \rceil = k + 1$;

(xii) $n \leq 2^{\lceil logk \rceil} < 2n$; e $\frac{n}{2} \leq 2^{\lfloor logn \rfloor} < n$

PROGRESSÃO ARITMÉTICA (PA)

É uma sucessão numérica que, a partir do segundo, cada termo é igual ao anterior somado com a razão da Progressão Aritmética.
formúla do termo geral

$$a_k = a_1 + (n-1).r$$

soma de n termos de uma P.A.

$$S_n = n.(\frac{a_1 + a_n}{2})$$

PROGRESSÃO GEOMÉTRICA (PG)

Uma sucessão de números não nulos em que o quociente de cada um deles, a partir do segundo, pelo seu antecessor é sempre o número, chama-se Razão, denotada por q.
formúla do termo geral

$$a_n = a_1.q^{n-1}$$

formúlas de somas de PG's

$S_n = \frac{a_1(q^n-1)}{q-1}$, usada quando se tem a_1, q, n

$S_n = (\frac{a_nq-a_1}{q-1})$, usada em PG's finitas

$S_n = \frac{a_1}{1-q}$, usada em PG's infinitas $|q| < 1$

Produto de n termos de uma PG
$P_n = \sqrt{(a_1.a_n)^n}$

COMBINATÓRIA

Arranjos simples – agrupamentos de n elementos p a p. São diferenciados pela ordem de cada agrupamento.

$$A_{n,p} = \frac{n!}{(n-p)!}$$

Combinações simples – agrupamentos de n elementos p a p. Não são diferenciados pela ordem de cada agrupamento.

$$C_{n,p} = \frac{n!}{(n-p)!p!} \qquad C_{n,p} = \binom{n}{p}$$

Permutações simples – agrupamentos de n elementos n a n, importando a ordem de cada agrupamento.

$$P_n = n! = n(n-1)(n-2)\ldots(2)1$$

SOMATÓRIOS

i) $$\sum_{i=1}^{n} i = \frac{n(1+n)}{2}$$

ii) $$\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$$

iii) $$\sum_{i=0}^{k} 1/2^i = \sum_{i=0}^{k} 2^{-i} = 2 - \frac{1}{2^k}$$

vi) $$\sum_{i=0}^{k} a^i = \frac{a^{k+1} - 1}{a - 1}$$

v) $$\sum_{i=1}^{n} i^2 = \frac{1}{6} n.(n+1)(2n+2)$$

ANEXO B

CONSIDERAÇÕES SOBRE ALGORITMOS NUMÉRICOS PARALELOS

Este anexo consiste de um relatório parcial do estudo sobre algoritmos numéricos paralelos que vem sendo desenvolvido pelo subgrupo de Matemática Computacional - Algoritmos paralelos, coordenado pélo prof Tiaraju A Diverio.

Indice:

B1 Algoritmos numéricos paralelos
B2 Princípios para construçào de algoritmos paralelos
B2.1 Resolução de relações de recorrência
B2.2 Resolução de equações algébricas
B2.2.1 Método da Bissecção
B2.2.2 Método da Regula-falsi
B2.3 Solução de equações diferenciais parciais
B2.4 Solução de sistemas de equações

B1 Algoritmos numéricos paralelos

O modelo usualmente utilizado para descrever métodos numéricos no contexto de computadores digitais é o modelo de Von Neumann, que consiste de um processador composto de uma unidade de controle e uma unidade aritmética e acumuladora. Na memória são armazenados os dados e o programa.

Os algoritmos de análise numérica envolvem um grande número de operações elementares. Problemas de erros de arredondamento e sua propagação e questões de estabilidade numérica têm assumido grande importância.

Apartir de 1960, começou-se a estudar a real possibilidade do emprego de computadores paralelos na solução de problemas numéricos. Iniciou-se a estudar o aspecto teórico da existência de paralelismo máximo para uma determinada classe de problemas. Este tipo de questão é estudada na Teoria da Complexidade.

Alguns autores estabelecem prós e contras para o uso de computadores paralelos. Alguns itens favoráveis são:

a) O aumento da velocidade de processamento, uma vez que as estruturas convencionais tem suas limitações físicas e que um computador com n processadores é mais barato que n computadores com um só processador;

b) A possibilidade de resolução de problemas muito complexos para máquinas sequenciais;

c) A solução de problemas, os quais são de natureza paralela, como por exemplo operações vetoriais e simulações discretas;

Dois dos itens desfavoráveis ao uso são:

a) A utilização pobre das máquinas paralelas (questões de gerenciamento);

b) O acesso paralelo aos dados que ocasiona uma complicada organização de dados.

Um exemplo ilustrativo do uso de computadores paralelos, explorando o paralelismo das operações é na soma de n termos de uma sequência (ou um conjunto de números reais) (ai) (para i=1, ...,n). Para melhor ilustrar, vamos assumir uma sequência com 16 termos, ou seja n = 16. Então temos,

$$A = a1 + a2 + ... + a15 + a16$$

Para efetuar esse somatório são necessários 15 adições. Se considerarmos como uma unidade o tempo de cada adição, teremos que o tempo necessário para efetuar tal somatório é de 15 unidades de tempo.

Se usarmos dois processadores, somando as parcelas pares e as impares nos processadores, usaremos apenas sete adições e, mais uma para obter o soma total, ou seja,

$$B1 = a1 + a3 + a5 + a7 + a9 + a11 + a13 + a15$$
$$B2 = a2 + a4 + a6 + a8 + a10 + a12 + a14 + a16$$

$$A = B1 + B2$$

Portanto, temos que o tempo T2 para efetuar a soma usando dois processadores é de 8 unidades de tempo. Aumentando o número de processadores e seguindo a mesma linha de raciocínio, obtemos a relação número de processadores e unidades de tempo necessários para somar 16 números, descrita pela figura b1.

p	Tp
1	15
2	8
3	4
4	5
8	4

Fig.b1 Unidades de tempo necessárias para somar 16 números utilizando p processadores

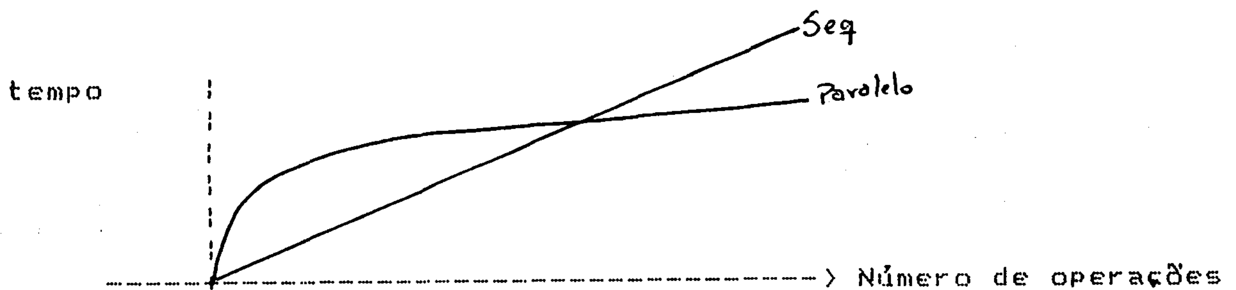


Fig.b2 Comparação entre Processamento Sequencial e Paralelo

B2 Princípios para construção de algoritmos paralelos

Neste item são enumerados alguns dos princípios para a construção de algoritmos paralelos. O primeiro princípio na construção de algoritmos paralelos é iniciar com um algoritmo serial (sequencial) e, então, convertê-lo em rotinas com operações com vetores. O propósito disto é que operações vetoriais podem ser executadas em paralelo. Esse princípio pode ser exemplificado com a solução de sistemas de equações lineares de ordem nxn, pelo método de eliminação de Gauss (triangularização da matriz de coeficientes).

O segundo princípio é o método do vetor iteração, onde é feita uma substituição do algoritmo sequencial direto por um algoritmo paralelo iterativo. Outro princípio empregado é o método da dupla recursividade.

A qualidade numérica de algoritmos paralelos envolvem os fatores da estabilidade, do erro de arredondamento e da propagação dos erros.

Na análise numérica a solução de um problema é muitas vezes, expresso como uma sequência $x1, x2, ..., xn$ de reais, onde xi $(i=1,2,...,n)$ pode depender de um xj de índice inferior, ou seja, $j < i$. Equações que possuem estas propriedades são chamadas equações de recorrência. Estas soluções para dada constituição de limites se constitui um problema de recorrência.

Alguns exemplos de problemas de recorrência são o produto interno de dois vetores, a avaliação de polinômios pelo método de Horner e o cálculo da sequência de Fibonacci de segunda ordem. A paralelização desse tipo de problema, geralmente depende da decomposição de uma expressão A em duas expressões A1 e A2 similares, onde cada uma delas pode ser calculada simultaneamente em um processador independente. Para assegurar isto, é necessário que exista uma função f tal que, $A=f(A1,A2)$; que o cálculo de A1 e A2 sejam processados independentemente e possuam a mesma complexidade computacional e, por fim, que A1 e A2 requeram a mesma sequência de operações para seus cálculos.

Neste relatório parcial, são descritos algoritmos paralelos para a solução de relações de recorrência, resolução de equações algébricas, equações diferenciais e sistemas de equações lineares. Uma relação de recorrêcia é uma equação que expressa o valor da função em um ponto em termos de valores de outros pontos. Estudos mostram um número de instâncias na qual equações de recorrência ocorrem em análise numérica. Estas instâncias incluem a solução de equações lineares através da eliminação de Gauss; solução de equações diferenciais ordinárias no tempo e métodos que levam para a solução de equações diferenciais no espaço.

Equações diferenciais parciais (PDE) aparecem frequentemente na engenharia, na física, na química e em outras ciências físicas. O caso de estudo considerado aqui, é o simples, o problema intuitivo de achar o estado constante de distribuição de temperatura em uma placa fina de metal retangular com condições fixas de limites.

O outro problema considerado são cálculos matriciais, como a solução de sistemas de equações lineares através do algoritmo de eliminação de Gauss.

B2.1 Resolução de relações de recorrência

Uma relação de recorrência geral de primeira ordem tem a forma:

$$x_j = a_j . x_{j-1} + d_j, \text{ para } j=1,...,n$$

onde os valores $x_0, a_1, a_2, ..., a_n$ e $d_1, ..., d_n$ são dados. Pode ser assumido, sem perda de generalidade, que $x_0 = a_1 = 0$, portanto se os valores não são nulos, d_1 pode ser redefinido como igual à soma $d_1 + a_1 x_0$. Assumiu-se isto para facilitar a análise e subsequente simplificação no algoritmo para entendimento.

Dada uma sequência de valores $d_1, d_2, ..., d_n$ a **soma parcial** x_i é definida pelo somatório dos d_j, com $j=1$ a i. Achados todas as somas parciais $x_1, ..., x_n$ da sequência de valores é na realidade um caso especial de recorrência linear de primeira ordem, o caso onde todos os a_j são iguais a um.

Um algoritmo B1, serve para calcular as somas parciais da sequência de valores, sua execução é ilustrada na figura b3. O algoritmo é chamado de método da soma parcial por cascata. Assumindo que $n=2^k$, os valores $d_1, ..., d_n$ são armazenados nos elementos processadores P0, P1, ..., Pn-1. Em outras palavras, a variável d(0) no elemento processador P0 contém d_1 e assim por diante. Quando o algoritmo termina, x(i) irá conter a soma parcial x_{i+1}. A complexidade do algoritmo é o(log n) com n elementos processadores no modelo.

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

ALGORITMO B1: Método da soma parcial por cascata

```
BEGIN
  FOR ALL Pi, where 0 <= i <= n-1
    DO x(i)=d(i);
  FOR i=0 to log n-1
    DO FOR ALL Pj, where 2^i +1 <= j <= n
       DO BEGIN
          t(j)= x(j-2^i)
          x(j)= x(j)+t(j)
       ENDFOR;
END
```

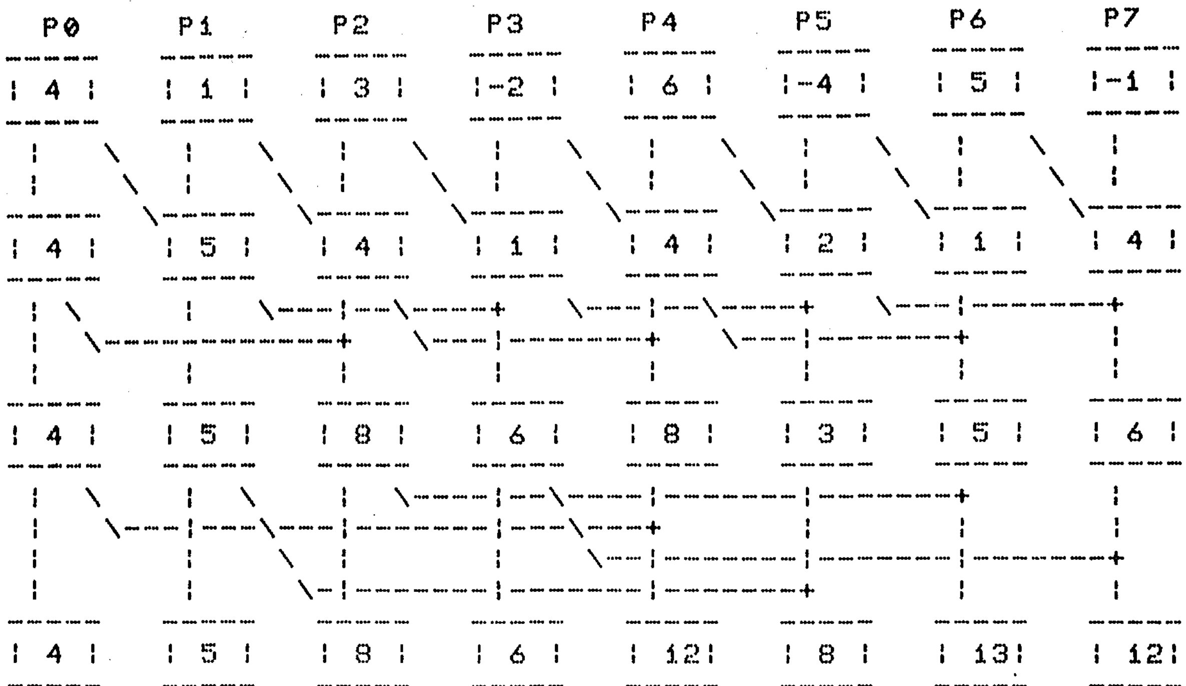


Fig.b3 Execução do algoritmo da Soma Parcial por cascata (modelo com 8 processadores)

Um modelo bastante similar pode ser desenvolvido para resolver uma relação de recorrência linear de primeira ordem geral. O algoritmo é chamado de **resolução cíclica** e ele inicia pela redução de relações entre termos adjacentes na sequência com relações entre cada outro termo na sequência. No segundo passo, relações entre dois termos na sequência são reduzidos para relações entre quatro termos na sequência. Depois de um número logaritmico de passos de reduções, os termos são tomados no seu valor final, pois eles são usados para valores constantes e não para outros termos.

Considerando dois sucessivos termos na relação de recorrência, temos:

$$x_j = a_j . x_{j-1} + d_j$$
$$x_{j-1} = a_{j-1} . x_{j-2} + d_{j-1}$$

aplicando a redução, resulta em :

$$x_j = a_j . a_{j-1} . x_{j-2} + a_j . d_{j-1} + d_j$$

que com substituição de variável se tem:

$$x_j = a_j^{(1)} . x_{j-2} + d_j^{(1)}$$

Esta equação é uma relação de recorrência de primeira ordem e, representa um progresso em relação a relação inicial, pois envolvem termos alternativos da sequência original. Este processo pode ser repetido log n vezes. A cada passo são gerados um conjunto de relações de recorrência:

$$x_j = a_j^{(l)} . x_{j-2^l} + d_j^{(l)} \quad , \text{ onde } l=0,1,...,\log n \text{ e } j=1,..,n$$

e onde

$$\left.\begin{aligned} a_j^{(l)} &= a_j^{(l-1)} . a_{j-2^{(l-1)}}^{(l-1)} \\ d_j^{(l)} &= a_j^{(l-1)} . d_{j-2^{(l-1)}}^{(l-1)} + d_j^{(l-1)} \end{aligned}\right\} (*)$$

com a condição inicial,

$$a_j^{(0)} = a_j \quad \text{e} \quad d_j^{(0)} = d_j$$

Se o valor da variável subscrita i de a_i, d_i e x_i estiver fora do intervalo de 1 a n, então o valor referenciado é nulo. Quando $l=\log n$, a variável subscrita de cada referência para x_{j-2^l} é menor que um. Portanto, a solução para relação de recorrência pode ser achada em log n reduções, sendo,

$$x_j = d_j^{\log n}$$

A chave para o algoritmo paralelo é a paralelização da avaliação das equações (*), para todos a_j e d_j. A equação de $d_j^{(l)}$ pode ser vista como uma relação de recorrência de primeira ordem, mas não é. Os valores calculados em cada iteração l dependem somente de valores calculados na iteração l-1. Entretanto, todos os valores na l-ésima iteração podem se calculados simultaneamente. O algoritmo paralelo para redução cíclica é dado a seguir.

ALGORITMO B2: Algoritmo da redução cíclica

```
BEGIN
  FOR i=0 to log n-1
     DO FOR ALL Pj, where 2^i  +1 <= j <= n-1
       DO BEGIN
          IF i >=1
            THEN BEGIN
               t(j)= a(j-2^i)
               a(j) = a(j)+t(j)
          ENDIF
          t(j)=d(j-2^i)
          d(j) = a(j)*t(j) +d(j)
       ENDFOR
  FOR ALL Pj DO x(j)=d(j);
END.
```

B2.2 Resolução de equações algébricas

A resolução de equações algébricas ou, como é mais conhecido, o problema do cálculo de zeros de uma função não linear, ou cálculo das raízes de um polinômio, geralmente envolvem um intervalo que contém um zero da função e se aplica algum método. Esses métodos variam em sua complexidade, número de operações, na convergência ou não para o zero da função.

São exemplificados aqui métodos de quebra, que são métodos onde a idéia básica consiste de, a partir de um intervalo fechado que contenha um zero de uma função contínua no intervalo, determina-se um ponto xm que "quebre" o intervalo em dois, de modo que o zero pertença a um dos subintervalos formados.

Na abordagem paralela, a solução é obtida pelo uso de algoritmos paralelos sobre máquinas paralelas. Nessa versão inicia-se com um intervalo fechado inicial que contenha um zero z. Algoritmos são providenciados para gerar uma sequência de intervalos tal que:

$$\{ I^{\langle k \rangle} \}, \quad \text{onde para cada } I^{\langle k \rangle}, \; z \in I^{\langle k \rangle}$$

O objetivo é obter algoritmos monotônicos. A cada iteração n valores da função são calculados em paralelo, usando n processadores. Mais formalmente, temos a função real f, tal que:

$$f: [a^{\langle 0 \rangle}, b^{\langle 0 \rangle}] \longrightarrow \mathbb{R},$$

que possui zeros z no intervalo $I^{\langle 0 \rangle} := [a^{\langle 0 \rangle}, b^{\langle 0 \rangle}]$.

A procura pela solução da equação f(x)=0, pode ser expressa pela geração de uma sequência de intervalos:

$\{ I^{\langle k \rangle} \}$, onde $I^{\langle k \rangle}$ é gerado a partir de um intervalo inicial, segundo a regra:

i) $z \in I^{\langle k \rangle}$, para $k >= 0$;

ii) $I^{\langle k \rangle} \subset \ldots \subset I^{\langle 2 \rangle} \subset I^{\langle 1 \rangle} \subset I^{\langle 0 \rangle}$;

iii) $I^{\langle k \rangle} \longrightarrow z$, quando k tende ao infinito;

A regra anterior garante que o algoritmo gere uma sequência monotônica de limites inferiores e superiores para z e, ainda, a convergência pode ser assegurada independentemente de quão bem foi escolhido o intervalo inicial. Na prática o item iii) é introduzido no algoritmo devido ao acúmulo de erro de arredondamento e na forma:

iii) $I^{\langle k \rangle} = I, \quad k >= k0$

O intervalo atua como uma espécie de ponto-fixo, que não interfere sobre o método considerado.

Define-se o diâmetro d do intervalo I:= [a ,b] por:

$$d(I) := b - a.$$

As condições de convergência para a sequência de intervalos são estabelecidas através de condições baseadas na distância.

São descritos a seguir, dois métodos de quebra sequênciais e suas versões paralelas, onde se considera a função y = f(x), da qual se quer calcular seus zeros.

B2.2.1 Método da Bisecção

Versão sequencial

O método da Bissecção é um método de quebra e é útil para o cálculo de zeros de funções, desde que para a função f(x) em um dado intervalo I = [a, b] que contenha um zero (simples), ou seja, f(a)*f(b) < 0, divide-se o intervalo ao meio e, calcula-se o valor da função nesse ponto intermediário xm. Caso xm não seja a raiz, verifica-se em qual dos dois subintervalos o zero da função permaneceu, através do produto de f(a)*f(xm) ser menor ou não que zero, tendo assim, outro intervalo menor, para o qual se repete o processo, até que um dos critérios de parada seja satisfeito ou que se encontre a raiz real.

Na figura b4 temos a interpretação geométrica do método da bissecção com o valor xm intermediário como o ponto médio do intervalo (xm = (a+b)/2).

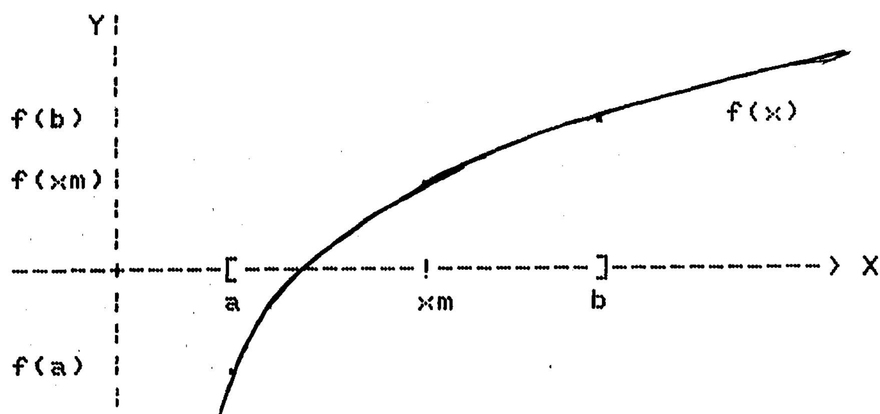


Fig.b4 Interpretação geométrica do Método da Bissecção

Algumas observações se fazem necessárias sobre o método como por exemplo, que ele não pode ser utilizado para calcular zeros muito próximos e nem para zeros com multiplicidade par, pois o teste f(a)*f(b) < 0 não será satisfeito.

O método, quando satisfeita a condição inicial converge, mas esta convergência é muito lenta; ganha-se a cada iteração um dígito binário de exatidão, o que resulta que tenhamos que fazer 3.3 iterações para se ganhar um dígito decimal.

Depois de k repetições do processo, o intervalo inicial I será reduzido de tamanho para (b - a) / 2^k. Se for tomado como valor aproximado do zero da função o ponto médio, este terá um erro máximo de (b - a) / 2^(k+1).

O algoritmo B3 é a versão sequencial para o método da Bissecção. Nele é assumido que f(x) é uma função definida previamente. Os dados de entrada são os extremos a e b do intervalo inicial e o valor requerido de algarismos significativos corretos (ASCREQ). A variável K representa o número de iteração, enquanto que MENOR é um valor pequeno dependente da máquina.

ALGORITMO B3: Método da Bissecção sequencial

```
INICIO
  LEIA A,B,ASCREQ;
  K:=0;
  FM:=10;
  FA:=f(A);
  FB:=f(B);
  ENQUANTO FA*FB>0
   FAÇA INICIO
     ESCREVA"Intervalo não satisfaz a condição de uso do método";
     LEIA A,B;
     FA:=f(A);
     FB:=f(B);
   FIM;
  ESCREVA "Valores iniciais:",A,B;

  ENQUANTO K<30 & ASC<ASCREQ & Abs(FM)>MENOR
   FAÇA INICIO
     XM:=(A+B)/2;
     K:=K+1;
     FM:=f(XM);

     SE Abs(FM)<MENOR
       ENTÃO INICIO
         A:=XM; B:=XM;
         FA:=FM; FB:=FM;
        FIM
       SENÃO SE FA*FM <0
               ENTÃO FAÇA INICIO
                 B:=XM; FB:=FM;
                FIM
               SENÃO FAÇA INICIO
                 A:=XM;  FA:=FM;
                FIM;
     ASC:=-(0.3+LOG(my+Abs((B-A)/B)));
   FIM %ENQUANTO;
  SE K=30 & ((ASC< ASCREQ) OU (Abs(FM)>MENOR))
     ENTÃO ESCREVA "Não convergiu",XM,ASC,K
     SENÃO ESCREVA "Raiz =",XM,ASC,K;
FIM.
```

Para melhor ilustrar o método, sua convergência e para melhor compreensão de seu algoritmo é dado um exemplo prático, o cálculo da raiz do polinômio Pn(x) abaixo,no intervalo I=[3,4], com no mínimo cinco algarismos significativos corretos.

$$Pn(x) = x^4 + 2x^3 - 7.5x^2 - 20x - 11$$

Teste inicial: Verificar se no intervalo I=[3,4] existe uma raiz, ou seja, se $f(3)*f(4) < 0$. Como f(3)= -3.5000 e f(4)= 173.0000 estão satisfazendo a condição de utilização, então escolhemos o ponto médio xm= 3.5 e verificamos se o valor da função no ponto xm é nulo ou não. f(xm)= 62.9375.

Como f(3)*f(3.5) < 0, temos o novo subintervalo como sendo [3.0 ; 3.5] e, assim, repetimos o processo, o qual está descrito na figura b5.

K	A	B	XM	F(XM)	ASC
0	+3.0000000	+4.0000000	+3.5000000	+62.9375000	0.5
1	+3.0000000	+3.5000000	+3.2500000	+25.0039063	0.8
2	+3.0000000	+3.2500000	+3.1250000	+9.6604004	1.1
3	+3.0000000	+3.1250000	+3.0625000	+2.8178864	1.4
4	+3.0000000	+3.0625000	+3.0312500	-0.4053345	1.7
5	+3.0312500	+3.0625000	+3.0468750	+1.1900482	2.0
6	+3.0312500	+3.0468750	+3.0390625	+0.3883209	2.3
7	+3.0312500	+3.0390625	+3.0351563	-0.0095139	2.6
8	+3.0351563	+3.0390625	+3.0371094	+0.1891479	2.9
9	+3.0351563	+3.0371094	+3.0361328	+0.0897598	3.2
10	+3.0351563	+3.0361328	+3.0356445	+0.0400925	3.5
11	+3.0351563	+3.0356445	+3.0354004	+0.0152893	3.8
12	+3.0351563	+3.0354004	+3.0352783	+0.0028915	4.1
13	+3.0351563	+3.0352783	+3.0352173	-0.0033188	4.4
14	+3.0352173	+3.0352783	+3.0352478	-0.0002136	4.7
15	+3.0352478	+3.0352783	+3.0352631	+0.0013428	5.0
16	+3.0352478	+3.0352631	+3.0352554	+0.0005646	5.3

Fig.b5 Valores parciais do método da Bissecção para o polinômio Pn(x)

Versão Paralela

Seja p o número de processadores. Para implementação paralela o intervalo é subdivididos em **p+1** subintervalos, através da escolha de **p** pontos interiores do intervalo. Para os quais são calculados, em paralelo, os valores da função. Entre os p+1 intervalos, existe um intervalo particular Ij = [aj, bj] tal que o produto dos extremos é menor que zero. Caso em nenhum dos intervalos seja satisfeita esta condição, é devido a que algum dos p pontos intermediários deve ser o zero da função.

Pode-se assumir que o tempo de pesquisa do novo subintervalo seja desprezível em relação ao tempo requerido para calcular o valor da função. Esta afirmativa é justificada na visão de que a rotina converge e que a velocidade pode ser estimada como quociente:

$$Sp = \frac{Ki}{Kp},$$

onde Ki é o número de subdivisões de intervalos requeridos quando i processadores são usados.

Para melhor ilustrar, consideraremos uma máquina paralela com dois processadores, ou seja p = 2. Dado um intervalo inicial I=[a, b], determina-se dois pontos intermediários (pois p=2) de I, de modo a termos três subintervalos de tamanho T, dado

por T := (b-a)/(p+1), que no caso específico seria T= (b-a)/3. Os pontos intermediários seriam p1= a+T e p2=a+2T.

Uma vez determinados os pontos intermediários, têm-se três subintervalos [a, p1], [p1, p2] e [p2, b], dos quais um contém a raiz. Para esse intervalo, seria então, repetido o processo. A figura b6 contém a interpretação geométrica do método da bissecção para uma máquina com dois processadores.

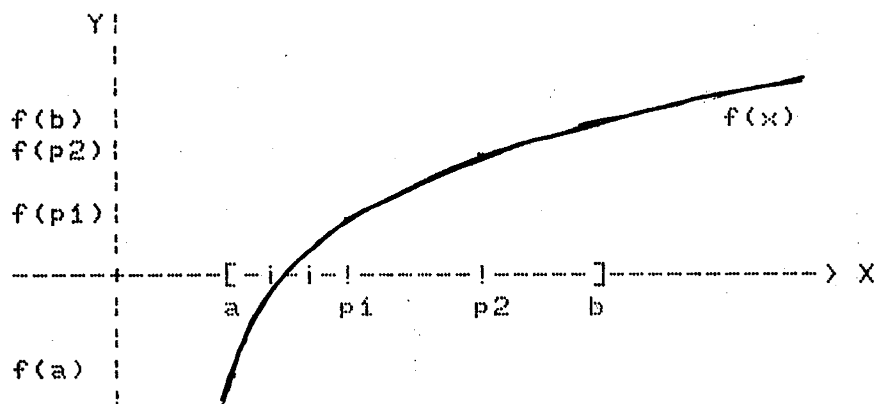


Fig.b6 Interpretação geométrica do Método da Bissecção para uma máquina com dois processadores

Usando o algoritmo sequencial, acha-se $\hat{z}$ tal que $|z-\hat{z}| < \Theta$, onde z é o valor real da raiz e quando o intervalo final for menor que 2Θ em tamanho.

Seja d o tamanho (diâmetro) do intervalo I=[a, b], então

$$\frac{d}{2^{k1}} <= 2\Theta,$$

entretanto, a versão paralela requer p+1 passos de redução no tamanho do intervalo e, portanto

$$\frac{d}{(p+1)^{kp}} <= 2\Theta,$$

$$2^{k1} = (p+1)^{kp},$$

donde temos,

$$k1 = kp \, Log2\,(p+1)$$

e portanto

$$Sp = Log2\,(p+1),$$

ou seja, Sp é da ordem Log2 (p).

O algoritmo B4 é a versão simulada do algoritmo paralelo que implementa o método da Bissecção. Nele os valores de a e b, extremos do intervalo são representados por X(0) e X(P+1)

e o diâmetro do intervalo é representado por T. As demais variáveis são análogas ao algoritmo sequencial. As partes grifadas indicam ações que podem ser calculadas em paralelo.

ALGORITMO B4: Versão paralela do Método da Bissecção

```
INICIO

  P:= <Nro processadores>;

  RESPOSTA="FALSE";
  K:=0; ASC:=0;

  LEIA X(0),X(P+1), ASCREQ;
  F(0):=f(X(0));
  F(P+1):=f(X(P+1));

  ENQUANTO F(0)*F(P+1) >0
    FAÇA INICIO
     ESCREVA "INTERVALO NÃO SATISFAZ CONDIÇÃO DE USO DO MÉTODO";
     LEIA X(0),X(P+1);

     F(0):=f(X(0));
     F(P+1):=f(X(P+1));
    FIM;

  ENQUANTO  K < 30 & ASC < ASCREQ & RESPOSTA = "FALSE"
    FAÇA INICIO
     K:=K+1;
     T:=(X(P+1)-X(0))/(P+1);

     PARA I:=1 ATÉ P
       FAÇA INICIO
         X(I):= X(0) + I * T;
         F(I):= f(X(I));
       FIM;

     PARA I:=1 ATÉ P
       FAÇA SE Abs(F(I)) < MENOR
               ENTÃO INICIO
                 RESPOSTA:="TRUE"
                 X(0):=X(I);
                 X(P+1):=X(I);
                 F(0):=F(I);
                 F(P+1):=F(I);
               FIM;
```

```
    SE RESPOSTA="FALSE"
       ENTÃO INICIO

          PARA I:=1 ATÉ P+1
            FAÇA SE F(I-1)*F(I)<0
                     ENTÃO INICIO
                        X(0):=X(I-1);
                        F(0):=F(I-1);
                        X(P+1):=X(I);
                        F(P+1):=F(I);
                     FIM; %fim do para

            ASC:=-(0.3 + Log(my + Abs((X(P+1)-X(0))/X(P+1))));

       FIM; %fim do se-então

   FIM; % fim enquanto

  SE K=30 & ((ASC<ASCREQ) OU RESPOSTA="FALSE")
     ENTÃO ESCREVA"não convergiu",X(0),X(P+1),ASC,K;
     SENÃO ESCREVA X(0),X(P+1),ASC,K;
FIM.
```

A seguir uma simulação do algoritmo para o mesmo exemplo da versão sequencial, com os mesmos valores iniciais, ou seja, I = [3, 4] com ASC >5 é fornecida pela figura b7.

K	A	F(A)	B	F(B)	ASC
0	+3.0000000	-3.5000000	+4.0000000	+173.0000000	0.0
1	+3.0000000	-3.5000000	+3.3333333	+36.5308600	0.7
2	+3.0000000	-3.5000000	+3.1111111	+8.0934360	1.1
3	+3.0000000	-3.5000000	+3.0370370	+0.1818056	1.6
4	+3.0246914	-1.0654140	+3.0370370	+0.1818056	2.1
5	+3.0329218	-0.2361798	+3.0370370	+0.1818056	2.6
6	+3.0342936	-0.0970954	+3.0356653	+0.0422178	3.0
7	+3.0352080	-0.0042632	+3.0356653	+0.0422178	3.5
8	+3.0352080	-0.0042632	+3.0353605	+0.0112356	4.0
9	+3.0352080	-0.0042632	+3.0352589	+0.0008929	4.5
10	+3.0352419	-0.0008131	+3.0352589	+0.0008929	5.0
11	+3.0352476	-0.0002393	+3.0352532	+0.0003497	5.4

Fig.b7 Resolução pelo método da bissecção paralelo por uma máquina de dois processadores

B2.2.2 Método da Regula-Falsi

O método da Regula-Falsi, também conhecido como o método da falsa-posição, é semelhante ao método da Bissecção. Temos que ter um intervalo J = [a, b] que contenha uma raiz da equação f(x) = 0, que tem que ser contínua no intervalo.

Traçamos a reta secante aos pontos (a, f(a)) e (b, f(b)), sendo o ponto de intersecção da reta secante com o eixo x, denominado xm. Analiticamente temos:

$$xm = a - \frac{(b-a)\ f(a)}{f(b) - f(a)} = \frac{a\ f(b) - b\ f(a)}{f(b) - f(a)}$$

O novo intervalo é determinado de acordo com o sinal da função em [a,b], e xm, escolhendo-o de modo que a raiz esteja contida dentro do novo intervalo. Se f(a) f(xm) < 0 então o novo intervalo será [a, xm], caso contrário será [xm, b]. Repetindo o processo até que seja satisfeito algum critério de parada ou que se tenha encontrado a raiz.

Na figura b8 temos a interpretação geométrica do método Régula-Falsi nos casos convexo e côncavo.

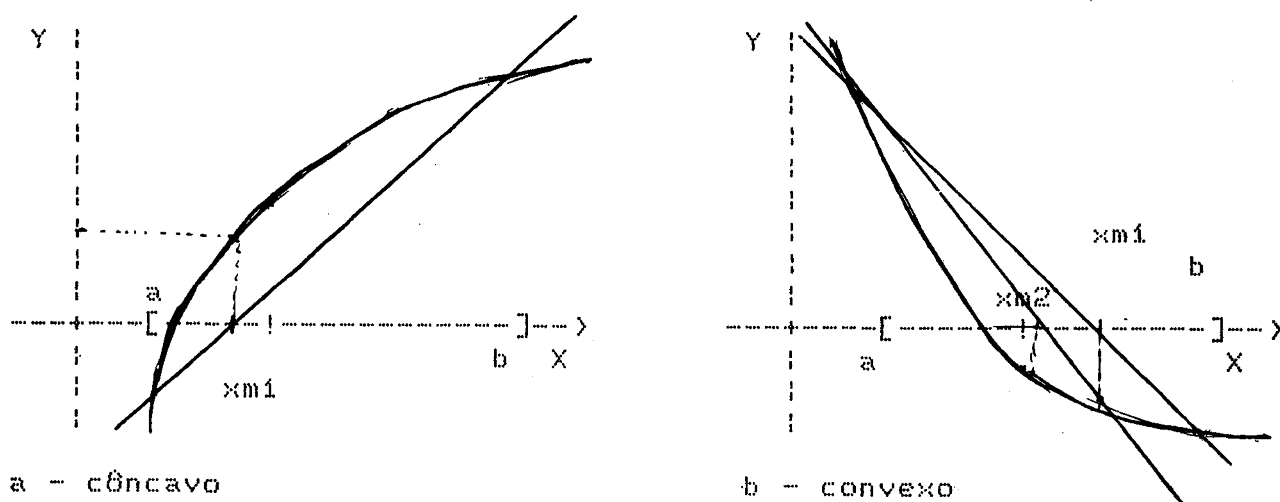


Fig.b8 Interpretação geométrica do método Regula-Falsi

Como o método necessita duas estimativas iniciais, uma de cada lado da raiz, torna-se extremamente difícil utilizá-lo quando não se tem nenhuma idéia da localização da raiz, ou se há raízes quase iguais. Além disso, raízes múltiplas de ordem par simplesmente não podem ser achadas.

O método da regula-falsi sempre converge uma vez satisfeita a condição inicial de uso, mas não é muito rápido. No caso de funções convexas ou côncavas no intervalo [a, b], temos que uma das extremidades do intervalo permanece sempre a mesma, ocasionando uma convergência linear. No caso em que ambas as extremidades mudam, a convergência é super linear, sendo o valor de ordem de convergência p = 1,618.

A seguir á apresentado o algoritmo sequencial do método Regula-Falsi. Nele é assumido que f(x) é uma função definida préviamente, assim como a função ASC, que calcula o número de algarismos significativos corretos. Os dados de entrada são os extremos a e b do intervalo inicial e o valor requerido de algarismos significativos corretos (ASCREQ).

ALGORITMO B5: Algoritmo doMétodo Regula-Falsi sequencial

```
INICIO

  FXM:=1;
  K:=0;
  ASC:=0;

  LEIA "Entre com os valores de A, B, ASCREQ", A,B,ASCREQ;
  FA:=f(A);
  FB:=f(B);

  ENQUANTO ASC < ASCREQ OU abs(FXM) > 10^(-(ASCREQ-1)) OU K<>30;
   FAÇA INICIO
     XM:= A- ((B-A) * FA / (FB-FA));
     FXM:= f(XM);
     K:=K+1;
     SE FA*FXM < 0;
        ENTÃO INICIO
           B:=XM;
           FB=FXM;
          FIM
        SENÃO INICIO
           A:=XM;
           FA:=FXM;
          FIM
     ASC:=ASC(A,B);
   FIM

  SE ASC < ASCREQ OU K > 30;
     ENTÃO ESCREVA "Nao convergiu"
     SENÃO ESCREVA XM,ASC,K;
FIM
```

Para melhor ilustrar o método, sua convergência e para melhor compreensão de seu algoritmo é dado um exemplo prático, o cálculo da raiz do polinômio Pn(x) abaixo no intervalo I=[3,4], com no mínimo cinco algarismos significativos corretos.

$$Pn(x) = x^4 + 2x^3 - 7.5x^2 - 20x - 11$$

Teste inicial: Verificar se no intervalo I=[3,4] existe uma raiz, ou seja, se f(3)*f(4) < 0 . Como f(3)= -3.5000 e f(4)= 173.0000 estão satisfazendo a condição de utilização. A figura b9 apresenta os valores parciais do método Regula Falsi.

K	A	B	XM	F(XM)	ASC
0	3.0000000	4.0000000	3.0198300	-1.5510119	0.3
1	3.0198300	4.0000000	3.0285395	-0.6787926	0.3
2	3.0285395	4.0000000	3.0323363	-0.2954692	0.3
3	3.0323363	4.0000000	3.0339862	-0.1283094	0.3
4	3.0339862	4.0000000	3.0347021	-0.0556530	0.3
5	3.0347021	4.0000000	3.0350126	-0.0241114	0.3
6	3.0350126	4.0000000	3.0351470	-0.0104664	0.3
7	3.0351470	4.0000000	3.0352054	-0.0045384	0.3
8	3.0352054	4.0000000	3.0352307	-0.0016235	0.3
9	3.0352307	4.0000000	3.0352417	-0.0008537	0.3
10	3.0352417	4.0000000	3.0352464	-0.0003614	0.3
11	3.0332464	4.0000000	3.0352484	-0.0001423	0.3
12	3.0332484	4.0000000	3.0352492	-0.0000743	0.3
13	3.0332492	4.0000000	3.0352496	-0.0000139	0.3
14	3.0332496	4.0000000	3.0352497	-0.0000154	0.3
15	3.0332497	4.0000000	3.0352498	-0.0000172	0.3
16	3.0332498	4.0000000	3.0352499	+0.0000038	7.2

Fig.b9 Valores parciais do método da Regula-Falsi para o polinômio Pn(x)

Como se pode observar no exemplo acima, um dos extremos permaneceu constante, o que pode ser melhorado através de uma adaptação, como veremos em outros métodos. Outra observação é que o ASC permaneceu nulo, em virtude do extremo fixo. O valor final do ASC é devido a troca de sinal da função na última iteração.

Versão paralela

Seja p o número de processadores. Para implementaçãp paralela I:=[a;b] é novamente subdividido em p+1 subintervalos, através da escolha de p pontos equidistantes inferiores de I. Para os quais são calculados, simultaneamente em paralelo, os valores da função f(mi)(para i- 1 até p). O método Regula-Falsi é, então aplicado nos intervalos [a, mi] ou [mi, b] dependendo da condição ser verdadeira, isto é, f(a)*f(mi) <0 ou f(mi)*f(b)<0 e, se nenhum dos pontos mi é zero da função.

Com isto, produz-se uma sequência de p pontos zi (para i=1 até p). Finalmente, a partir de a, b, mi, zi (i=1 até p) é escolhido um par de modo que seja definido o menor subintervalo Ik+1 = [ak+1; bk+1].

Observa-se que se os pontos estiverem ordenados numa sequência de 2p+2 elementos, bastará verificar o sinal do produto da função do ponto com seu vizinho próximo a direita. Determinando desta forma o novo intervalo, para o qual se repete o processo, até que se encontre o zero da função ou que se esteja suficientemente próximo dele.

A figura b10 apresenta o esboço gráfico da aplicação do método Regula-Falsi nos intervalos onde a condição é satisfeita, para a determinação dos p pontos zi.

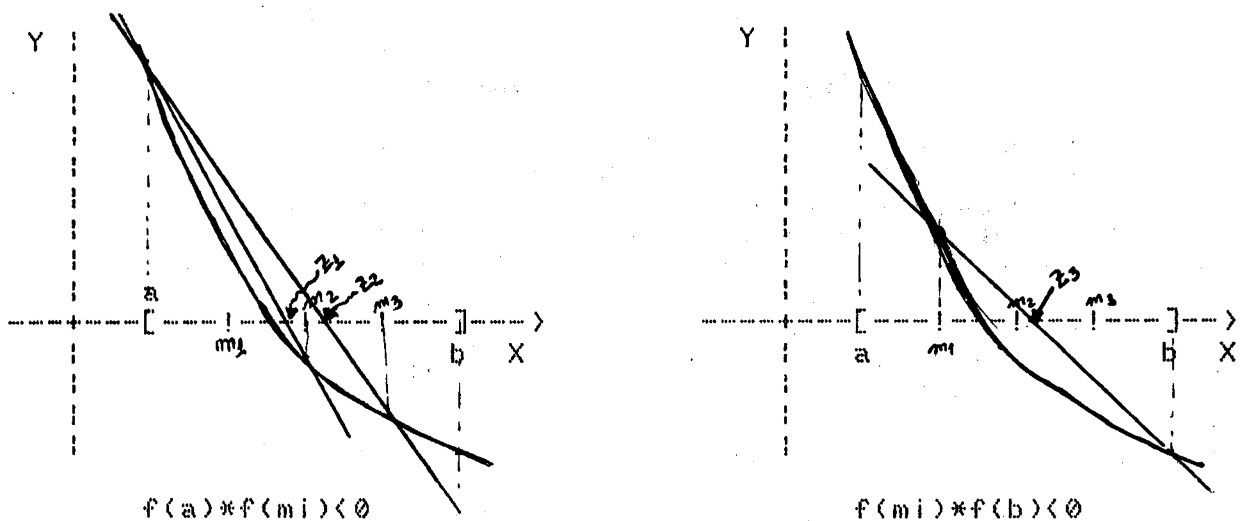


Fig.b10 Gráficos de determinação dos zi

A seguir é apresentado o algoritmo da versão paralela do Método Regula-Falsi.

ALGORITMO B6: Algoritmo do método regula-falsi paralelo

```
INICIO

  P:= <Nro processadores>;
  RESPOSTA="FALSE";
  K:=0; ASC:=0;
  LEIA X(0),X(P+1), ASCREQ;
  F(0):=f(X(0));
  F(P+1):=f(X(P+1));
  ENQUANTO F(0)*F(P+1) >0
    FAÇA INICIO
     ESCREVA "INTERVALO NÃO SATISFAZ CONDIÇÃO DE USO DO MÉTODO";
     LEIA X(0),X(P+1);
     F(0):=f(X(0));
     F(P+1):=f(X(P+1));
    FIM;
  ENQUANTO  K < 30 & ASC < ASCREQ & RESPOSTA = "FALSE"
    FAÇA INICIO
     K:=K+1;
     T:=(X(P+1)-X(0))/(P+1);
     PARA I:=1 ATÉ P
      FAÇA INICIO
        X(I):= X(0) + I * T;
        F(I):= f(X(I));
      FIM;
```


UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

```
     PARA I:=1 ATÉ P
       FAÇA SE Abs(F(I)) < MENOR
               ENTÃO INICIO
                 RESPOSTA:="TRUE"
                 X(0):=X(I);
                 X(P+1):=X(I);
                 F(0):=F(I);
                 F(P+1):=F(I);
               FIM;
     SE RESPOSTA="FALSE"
        ENTÃO INICIO
          PARA I:=1 ATÉ P
             FAÇA
              INICIO
               SE F(0)*F(I)<0
                ENTÃO
                 Z(I):=X(0)-((X(I)-X(0))*F(0))/(F(I)-F(0))
                SENÃO
                 Z(I):=X(I)-((X(P+1)-X(I))*F(I))/(F(P+1)-F(I));
               FZ(I):=f(Z(I));
            FIM;
           PARA I:=1 ATÉ P
            FAÇA SE Abs(FZ(I)) < MENOR
               ENTÃO INICIO
                 RESPOSTA:="TRUE"
                 X(0):=Z(I);
                 X(P+1):=Z(I);
                 F(0):=FZ(I);
                 F(P+1):=FZ(I);
               FIM;
      FIM;
      SE RESPOSTA="FALSE"
       ENTÃO INICIO

         ordena (2P+2:X(I),F(I),Z(I),FZ(I);O(J),FO(J));

         PARA J:=1 ATÉ 2P+1
          FAÇA SE FO(J-1)*FO(J) < 0
           ENTÃO INICIO
                 X(0):=O(J-1);
                 X(P+1):=O(J);
                 F(0):=FO(J-1);
                 F(P+1):=FO(J);
               FIM;
           ASC:=-(0.3 + Log(my + Abs((X(P+1)-X(0))/X(P+1))));
        FIM; %fim do se-então
    FIM; % fim enquanto

   SE K=30 & ((ASC<ASCREQ) OU RESPOSTA="FALSE")
      ENTÃO ESCREVA"não convergiu",X(0),X(P+1),ASC,K;
      SENÃO ESCREVA X(0),X(P+1),ASC,K;
FIM.
```

A rotina ordena tem como parâmetros o número de elementos a serem ordenados, os pontos x_i com os pontos gerados pela intersecção da secante z_i (e suas respectivas imagens da função f_i e fz_i). A lista ordenada é no vetor O_i (e as imagens em FO_i). O algoritmo B7 é composto por duas etapas. Na primeira é feita a concatenação dos valores x_i e z_i e, na segunda, é a ordenação propriamente dita.

```
ALGORITMO B7: Ordena -Rotina de ordenação
INICIO
 O(0):=X(0);
 FO(0):=F(0);
 O(2P+1):=X(P+1);
 FO(2P+1):=FO(P+1);
 PARA I:=1 até P
  FAÇA INICIO
     O(I):=X(I); FO(I):=F(I);
     O(I+P):=Z(I); FO(I+P):=FZ(I);
  FIM;
 PARA J:=2P até 0 passo -1
  PARA I:=1 até J
   FAÇA SE O(I) > O(I+1)
    ENTÃO INICIO
      AUX:=O(I); AUF:=FO(I);
      O(I):=O(I+1); FO(I):+FO(I+1);
      O(I+1):=AUX; FO(I+1):=AUF;
    FIM;
FIM;
```

A primeira parte da rotina pode ser suprimida, desde que na hora de geração dos z_i, eles sejam armazenados no próprio vetor x_i e suas imagens em f_i. Para tanto, a dimensão dos vetores deve ser ampliada para $2p+1$ (variando de 0 a $2p+1$).

Outra observação, é que esta ordenação também poderá ser feita explorando recursos se simultaniedade, diminuindo o tempo necessário para o cálculo de cada iteração.

Para melor compreensão, é dado na figura b11, a seguir os valores parciais do Método Regula Falsi versão paralela para o mesmo exemplo da versão sequencial, com os mesmos valores iniciais, ou seja, I= [3, 4] com ASC >5.

K	A	F(A)	B	F(B)	ASC
0	+3.0000000	-3.5000000	+4.0000000	+173.0000000	0.0
1	+3.0291443	-0.6178645	+3.3333333	+36.5308600	0.7
2	+3.0348861	-0.3694970	+3.1305406	+10.2930000	1.2
3	+3.0352426	-0.0007492	+3.0667710	+3.2685390	1.7
4	+3.0352426	-0.0007492	+3.0352499	+0.0000042	5.3

Fig.b11 Valores parciais do método da Regula-Falsi versão paralela para o polinômio Pn(x), usando uma máquina com dois processadores

Observa-se que a versão paralela determinou o zero da função em quatro iterações, enquanto que a versão sequencial obteve a solução em dezesseis iterações, quatro vezes mais. Outra comparação que pode ser feita é com a vesaão paralela do método da Bissecção, onde a solução foi calculada em onze iterações, três vezes mais.

Estas observações estão sendo feitas para um caso particular, portanto não se pode generalizar.

B2.3 Solução de equações diferenciais parciais

Métodos para solução de equações diferenciais parciais (PDE) podem ser classificados em métodos diretos e métodos iterativos. Métodos diretos resolvem PDEs analiticamente; métodos iterativos iniciam com valores estimados em certa localização específica e, então, convergem para estimativas mais exatas. Algoritmos iterativos para resolver PDEs podem consumir um tempo de processamento bastante grande, por isto é interessante olhar para computações paralelas. Quando a utilidade de um método iterativo é verificada, o intervalo de convergência dos valores para se ter a solução é de grande importância.

J.Rice apresenta uma taxonomia para métodos paralelos, baseada no uso de três rotinas: particionamento, discretização e iteração.

Particionamento - As variáveis no problema são divididas em grupos. O critério de agrupamento pode ser geométrico, ou uma propriedade da matriz, ou uma propriedade física. Diferentes particionamentos podem ser usados em diferentes vezes e partições podem ser mais particionadas. Particionamento geralmente conduz ao uso de paralelismo.

Discretização - O problema contínuo PDE é substituído por um problema finito com um conjunto de números reais como variáveis. As duas técnicas mais comuns são diferenças finitas e funções bases (elementos finitos, polinômios e expansões em séries). Métodos de diferenças finitas e funções bases com suporte local são muito bons para métodos paralelos na fase de discretização, por serem esses cálculos altamente independentes uns dos outros. Discretizações são também divididos em métodos de alta ordem e de baixa ordem. Métodos de alta ordem são vantajosos para paralelização pois mais de um trabalho pode ser feito na fase de discretização dos cálculos.

Iteração - Iterações é uma técnica de propósito geral que têm dificuldades (não linearidade, problemas muito grandes, dependência de tempo). Iteração é inerentemente sequencial e portanto, não é muito propícia para exploração de paralelismo. Ainda que iteração seja essencial para resolver muitos PDEs, é bom só introduzi-la quando o número de iterações for pequeno e o trabalho em cada iteração bem grande e que se possa dividi-lo em componentes paralelos.

Considere o estado constante de temperatura bi-dimensional no problema de distribuição na figura 5.3. A placa fina retangular é cercada por três lados por corrente de condensação (temperatura a 100 graus centígrados). O quarto lado é tocado por uma barra de gelo (temperatura a zero grau centígrado). Um cobertor isolante cobre de cima a baixo a placa. O problema é achar o estado constante de distribuição da temperatura em 100 pontos espaçados formando uma malha na placa de 10x10 pontos.

Este problema é um exemplo de equação diferencial parcial linear de segunda ordem. Quando o estado constante de distribuição da temperatura é encontrado, o conjunto de equações diferenciais relatam os valores das variáveis nos pontos vizinhos na malha:

$$\phi = \frac{\phi_{x-1,y} + \phi_{x,y-1} + \phi_{x+1,y} + \phi_{x,y+1}}{4}$$

aqui, as variáveis subescritas x e y referem-se as coordenadas dos pontos na malha. Este problema é bastante simples e poderia ser resolvido analiticamente, entretanto, PDEs mais complicados necessitam ser resolvidos iterativamente. Por esta razão será explorada a solução iterativa. A rotina iterativa inicia pelo assumir um valor inicial estimado para cada variável $\phi_{x,y}$. A equação diferencial é então, usada para calcular valores sucessivos. Se os valores das variáveis convergirem para a solução, a rotina terá sucesso.

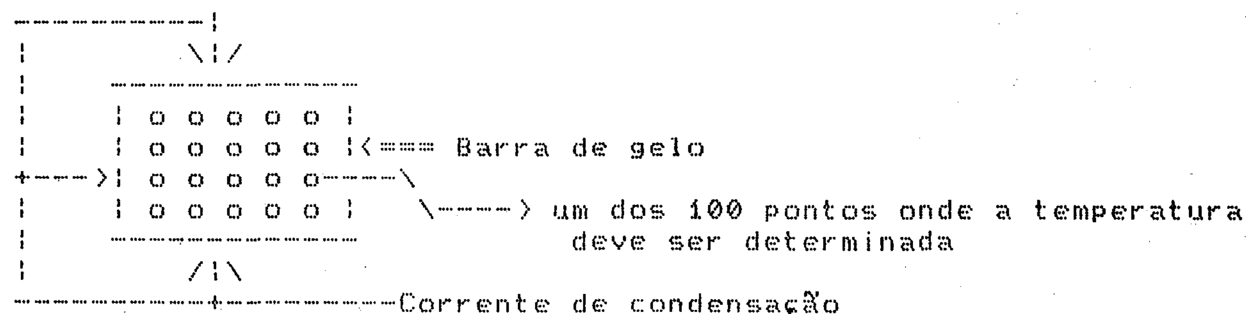


Fig.b12 Problema do estado cte de distribuição de temperatura

Uma rotina simples iterativa para o cálculo dos valores das variáveis ϕ pode ser desenvolvida, onde todos os valores dos pontos da malha são calculados simultaneamente pela fórmula:

$$\phi' = (\phi_{x-1,y} + \phi_{x,y-1} + \phi_{x+1,y} + \phi_{x,y+1})/4$$

os valores ϕ do lado direito da equação são os valores antigos, os valores ϕ' representam os novos valores.

O método proposto por Jacobi (em 1845) é bastante propício à paralelização, pois todas as variáveis que necessitam ser acessadas por um processador estão disponíveis em um processador adjacente.

O mais popular método iterativo para a solução de PDEs em computadores sequenciais é o sobrerelaxação sucessivas (SOR) por pontos. O SOR difere do método de Jacobi em dois importantes aspectos:

i) Um peso médio de $\phi'_{x,y}$ e $\phi^{old}_{x,y}$ é tomado para determinar um novo valor da variável ϕ;

ii) Novos valores substituem os antigos assim que calculados.

Num algoritmo sequencial pontos da malha são processados um por vez, coluna por coluna. A substituição dos valores antigos pelos novos é feita assim que os novos são calculados, aumentando a velocidade de convergência. Entretanto, este método sofre pela desvantagem que somente uma variável é calculada por vez.

Felizmente, uma variante paralela deste algoritmo retém esta velocidade de convergência. O algoritmo paralelo é referenciado como ordenação ímpar-par com aceleração de Chebyshev. Imagine os elementos processadores como se formando uma tábua de checagem. Cada iteração tem duas fases, na primeira, todos os processadores ímpar recebem novos valores para suas respectivas variáveis. Na segunda fase, todos os processadores pares geram novos valores para suas variáveis. O algoritmo paralelo também muda o peso entre $\phi'_{x,y}$ e $\phi^{old}_{x,y}$ em cada meia iteração.

Observa-se que estes métodos iterativos de solução de PDEs são convenientes para máquinas do tipo processador matricial. Em 1982, Deminet implementou alguns algoritmos para solução de PDEs em máquinas multiprocessadoras. Ele resolveu a equação de Laplace com as condições de contorno de Dirichlet pelo método das diferenças finitas. A equação:

$$\frac{d^2 z}{dx^2} + \frac{d^2 z}{dy^2} = 0$$

é resolvida por uma malha de 150x150 pontos, usando técnicas iterativas similares as já descritas. Em cada iteração um novo valor por elemento é transmitido a média dos valores aos seus vizinhos.

Cada processo executa em sua própria CPU, achando os valores para uma subseção contínua da malha. Para fazer o cálculo dos índices de uma forma mais simples, à cada processo é atribuído um número completo de colunas e iterativamente

calculados os valores das variáveis nessas colunas até os valores se estabilizarem.

A primeira providência feita no algoritmo por Deminet foi um ótimo trabalho de seleção de tarefa. O algoritmo original tanto atribuia subseções por processadores randômicos ou baseados na atribuição de números de módulos computacionais. O algoritmo proposto adimite subseções que podem necessitar mais iterações, todas no meio da malha, para processadores rápidos, todas no mesmo conjunto de dados.

A segunda modificação é para distribuir dados entre o grupo contendo processadores ativos, com o objetivo de providenciar localização de referência.

B2.4 Solução de sistemas de equações

Cálculos matriciais estão entre as pedras angulares de computações científicas. Problemas de álgebra linear, envolvem sistemas de equações lineares, problemas lineares dos mínimos quadráticos e, problemas algébricos de autovalores são fundamentais para o cálculo da solução de equações diferenciais, problemas de otimização e de análise de várias estruturas discretas. O uso efetivo de computadores de arquiteturas paralelas em computações científicas é, entretanto, criticamente dependente da exploração do paralelismo em cálculos matriciais.

Algoritmos matriciais têm sido a vanguarda de desenvolvimento de algoritmos em multiprocessadores, não somente porque eles vão construindo blocos nos quais muitos outros cálculos científicos são baseados, mas também porque eles servem como protótipos reais que apresentam muitos dos desafios fundamentais de computações paralelas na forma pura. Por isso, o desenvolvimento de algoritmos paralelos para cálculo matricial tem recebido forte ênfase dos pesquisadores em processamento paralelo, tanto como uma ferramenta, quanto um paradigma para computação científica em geral em arquiteturas paralelas.

Os problemas computacionais mais comuns em álgebra linear são a solução de equações lineares e o problema algébrico de autovalores para vários tipos de matrizes. O problema das características afetam qualquer implementação computacional, incluindo matrizes quadradas ou retangulares, simétricas ou não, densas ou esparsas, explicitamente representadas pelas suas entradas ou implicitamente representada pelas suas ações em vetores. Tais propriedades determinam qual algoritmo ou família de algoritmos é apropriada para resolver o problema em qual ambiente computacional, sequencial ou paralelo, podendo sofrer maior impacto no ambiente paralelo. Por exemplo, a necessidade de linhas ou colunas intercambiarem-se para estabilidade numérica pode ser uma complicação mais séria ao ambiente paralelo do que ao sequencial. Outra questão importante é o tamanho da matriz, o que detemina a quantidade de recursos computacionais que serão necessários, assim como que classes de algoritmos e estrutura de dados deve ser mais apropriada.

Os algoritmos paralelos para solução de sistemas de equações lineares podem ser por métodos diretos ou iterativos.

Fatoração de matrizes densas é um caso interessante de estudo para implementação paralela devido ao fato de existirem duas características que tendem a inibir a eficiência paralela. Elas são: restrições de precedência sequenciais, devido a sucessivas linhas, colunas e submatrizes da matriz fatorada, necessitam ser calculadas em ordem consecutiva, e comunicação global ser requerida porque cada sucessiva linha, coluna ou submatriz da matriz fatorada depender de todas precedentes linhas, colunas ou submatrizes.

Muitos projetistas de algoritmos têm mostrado na prática que os cálculos podem ser suficientemente gerenciados e a comunicação suficientemente rápida de forma a atender a elevada utilização de processadores e eficiente paralelização.

As principais distinções entre algoritmos paralelos para fatoração de matrizes densas são a concorrência e a comunicação. O grau de concorrência atendida e o custo de comunicação são determinados pela escolha de granularidade. Uma implementação de granularidade fina, com subtarefas de complexidade de ordem um (o(1)), como em algoritmos sistólicos e "topo de onda" potencialmente atendem o máximo de concorrência possível, mas este potencial não é alcançado, a menos que o gerenciamento de comunicação seja muito pequeno, com custo de comunicação correspondente ao custo de uma operação aritmética. Uma implementação com granularidade média, com subtarefas de complexidade de ordem n (o(n)), é semelhante para providenciar um ótimo equilíbrio entre concorrência e custo de comunicação na maioria das arquiteturas multiprocessadoras de propósito geral.

Fatoração de matrizes reduz o problema de solução de um sistema de equações geral em um problema de solução de um sistema triangular de equações. Um sistema de equações lineares AX=B pode ser resolvido, por fatorização, em três passos:

i) Decomposição da matriz dos coeficientes A no produto de uma matriz triangular inferior L e outra triangular superior U;

ii) Resolvendo o sistema LY=B;

iii) Resolvendo o sistema UX=Y.

Destes três passos, a rotina que triangulariza (i) tem alta complexidade computacional. O Algoritmo completo é dado no Relatório de pesquisa: Processamento vetorial e vetorização de algoritmos na máquina Convex C210 ([DIV91]), no capítulo seis, sob o nome de método de Banachievicz. Outro método lá existente é o de Cholesky.

Problemas esparsos tem tanto vantagens quanto desvantagens em relação ao caso denso. Sua principal vantagem é que a potencialidade esparsa leva a grande concorrência e menos comunicação devido a independência de dados de algumas partes do problema em relação a outras.

Métodos iterativos para solução de sistemas de equações lineares também se apresentam como um caso interessante de estudo para implementação em paralelo. Eles diferem dos métodos diretos, nos quais um conjunto fixo de cálculos precisam ser realizados em ordem para chegar a solução correta, a natureza autocorretiva dos métodos iterativos permite que se altere os cálculos para herdar paralelismos. O speedup de iterações individuais pelo acrescentar da concorrência não se beneficia, entretanto, facilita o cálculo da próxima iteração.

Métodos iterativos são frequentemente recomendados para implementações em paralelo por limitarem mais os requisitos de comunicação, em relação a comunicação global necessária em métodos diretos por fatoração.

Algoritmos para cálculo de autovalores de matrizes são menos desenvolvidos para máquinas paralelas do que dos para solução de sistemas de equações lineares. Uma das áreas que tem apresentado resultados satisfatórios para algoritmos paralelos para problemas de autovalores é no projeto de array sistólicos. Arrays sistólicos têm sido desenvolvidos para calcular autovalores e valores singulares de vários tipos de matrizes. Outra área de estudo que vem se desenvolvendo são os algoritmos paralelos baseados na divisão-e-conquista para cálculo de autovalores.

Algoritmos numéricos paralelos: Bibliografia

ANEXO C

Levantamento Bibliográfico nos periódicos da biblioteca do PGCC da UFRGS

C1 PERIÓDICOS ANALISADOS

ACM Transactions on Mathematical Software
Advances in Computers
Communications of the ACM
Computer & Educations
Computer & Mathemaics with applications
Computer Journal
Computing
Computing Surveys
IEEE Software
IEEE Transactions on Computers
IEEE Transactions on Software Engineering
International Journal of Parallel Programming
Journal of Computational and Applied Mathematics
Journal of the ACM
New Generation Computing
Parallel Computing
Proceedings of the IEEE
Software Practice & Experience
Science of Computer Programming
SIGNUM News Letter
Theoretical Computer Science

C2 TÍTULOS LEVANTADOS

- C2.1 Processamento vetorial e paralelo
- C2.2 Modelos de arquiteturas e máquinas
- C2.3 Paralelização (ambiente, técnicas, eficiência e recursividade)
- C2.4 Complexidade de algoritmos (sequenciais e paralelos)
- C2.5 Algebra Linear (sistemas de equações e matrizes)
- C2.6 Equações algébricas (Zeros de funções)
- C2.7 Equações Diferenciais
- C2.8 Problemas de comparação (sorting)
- C2.9 Ponto-flutuante e aritmética intervalar
- C2.10 Aplicações
- C2.11 Software matemático

C3 Artigos selecionados

C3.1 PROCESSAMENTO VETORIAL E PARALELO

COWELL, W. E; THOMPSOM, C. P. Transforming FORTRAN DO loops to improve performance on vector architectures. **ACM Transactions on the Mathematical Software**, New York, v.12, n.4, p.324-353, Dec, 1986.

COWELL, W. R.; THOMPSON, T. P. Tools do aid in discovering parallelism and localizing arithmetic in FORTRAN programs. **Software: Practice & Experience**, v.20, n.1, p.25-48, Jan, 1990.

EVANS, D. J.; HATZOPOULOS, M. The parallel calculation of the eigenvalues of a real matrix A. **Computers & Mathematics**, Oxford, v.4, n.3, p.211-218, 1978.

HWANG, K.; SU, SHUAN-PIAO; NI, M. L. Vector computer architecture and processing techniques. **Advances in Computers**, New York, v.20, p.116-197, 1981.

NICOL, D. M.; D'HALLARON, D. Z. Improved algorithms for mapping pipeline and parallel computations. **IEEE Transactions on Computers**, New York, v.C-40, n.3, p.295-306, Mar, 1991.

PETERSEN, W. P. Vector FORTRAN for numerical problems on CRAY-1. **Communication of the ACM**, New York, v.26, n.11, p.1008-1021, Nov, 1983.

RAMANDORTHY, C. V.; LI, H. F. Pipeline architecture. **ACM Computing Surveys**, New York, v.9, n.1, p.61-102, Mar, 1977.

C3.2 MODELOS DE ARQUITETURAS E MAQUINAS

DEKKER, E. The Cray-2 architecture. **Parallel computing** Amsterdam, p.575-580, 1990.

DETERT, U.; HOFEMANN, G. Crau X-mp and Y-mp memory performance. **Parallel computing**, Amsterdam, v.17, n.485, p.579-590, July, 1991.

ENSLOW Jr, P. H. Multiprocessor organization - a survey. **ACM Computing Surveys**, New York, v.9, n.1, p.103-129, Mar, 1977.

GALIL, Z.; PAUL, W.J. An efficient general purpose parallel computer. **Journal of the ACM** New York, v.30, n.2, p.360-387, Apr, 1983.

GOTO, A.; TANAKA, H.; MOTO-OKA, T. Highly parallel inference engine Pic-Goal rewriting model and machine architecture. **New Generation Computing**, v.2, n.1, p.37-58, 1984.

HILLS, W.D.; STEELE, G.L. Data parallel algorithms. **Communications of the ACM**, New York, v.29, n.12, p.1170-1183, Dec, 1986.

HWANG, K.; SU, SHUAN-PIAO; NI, M. L. Vector computer architecture and processing techniques. **Advances in Computers**, New York, v.20, p.116-197, 1981.

JAROSZ, J.; JAWOROWSKI, J. R. Computer tree - the power of parallel computation. **Computer Journal**, London, v.29, n.2, p.100-108, Apr, 1986.

JONES, A. K.; SHUWARZ, P. Experience using multiprocessor systems - a status report. **ACM Computing Surveys**, New York, v.12, n.12, p.121-166, June, 1980.

KUCK, D. J. A survey of parallel machine organization and programming. **ACM Computing Surveys**, New York, v.9, n.1, p.29-59, Mar, 1977.

MAPLES, C. Analysing software performance in a multiprocessor environment. **IEEE Software**, New York, v.2, n.4, p.50-63, July, 1985.

MISRA, J.; CHANDY, K. M. Asynchronous distributed simulation on sequence of parallel computation. **Communication of the ACM**, New York, v.24, n.4, p.198-206, Apr, 1981.

NATARASAN, N. A destributed synchronisation scheme for communicating processes.**Computer Journal**, London, v.29, n.2, p.109-117, Apr, 1986.

OLSON, R. Parallel processing in a message-based operation system. **IEEE Software**, New York, v.2, n.4, p.50-63, July, 1985.

RAMANDORTHY, C. V.; LI, H. F. Pipeline architecture. **ACM Computing Surveys**, New York, v.9, n.1, p.61-102, Mar, 1977.

ROSEMBERG, J. B.; BECHER, J. D. Mapping massive SIMD parallelism onto vector architecture for simulation. **Software: Practice & Experience**, v.19, n.8, p.739-756, Aug, 1989.

STEWART, G. W.; O'LEARY, D. P. Data-flow algorithms for matrix computations. **Communication of the ACM**, New York, v.28, n.8, p.840-853, Aug, 1985.

VENN, A. Dataflow machine architecture. **ACM Computing Surveys**, New York, v.18, n.4, Dec, 1986.

YAU, S. S.; FUNG, H. S. Associative processor architecture - a survey. **ACM Computing Surveys**, New York, v.9, n.1, p.3-27, Mar, 1977.

ZAKHAROV,V Parallelism and array processing. **IEEE Transactions on computers.** New York, v.C-33, n.1, p.45-78, Jan, 1984.

C3.3 PARALELIZAÇÃO (AMBIENTE, TECNICAS, EFICIENCIA, RECURSIVIDADE)

AGRAWAL,R.;JAGADISH,H.V. Partitioning techiniques for large grained parallelism. **IEEE Transactions on computers.** New York, v.37, n.12, p.1627-1634, Dec, 1988.

ALLEN, J.R.; KENNEDY, K. A parallel programming environment. **IEEE Software**, New York, v.2, n.4, p.21-29, July, 1985.

ANDREWS, G. R. Parallel programs: proof, principles and practice. **Communication of the ACM**, New York, v.24, n.3, p.140-146, Mar, 1981.

BENTLEY, J. L. Multidimensional divide-and-conquer. **Communication of the ACM**, New York, v.23, n.4, p.214-229, Apr, 1980.

BOKHARI,S.H.Partitioning problems in parallel, pipelined and distributed computing. **IEEE Transactions on computers**, New York, v.37, n.1, p.48-57, Jan, 1988.

BONDELI,S. Divide and conquer: a parallel algorithm for the solution of a tridiagonal linear system of equations. **Parallel computing**, Amsterdam, v.17, n.4&5, p.419-434, July, 1991.

DASGOPTA, S.; TARTAR, J. The identification of maximal parallelism in straight line microprograms. **IEEE Transactions on Computers**, New York, v.C-25 , n.10, p.986-992, Oct, 1976.

Developments and software requirenments of the emerging National Supercomputer Research Center. **IEEE Software**, New York, v.2, n.6, p.55-67, Nov, 1985.

EAGER, D. L.; ZAHORJAN, S.; LAZOWSKA, E. D. Speed versus efficiency in parallel algorithms. **IEEE Transactions on Computers**, New York, v.C-38, n.3, p.408-423, Mar, 1989.

EBENSTEIN, S. E.; McDERMOTT, T. L. Optimization techniques for parallel processing. **Software: Practice & Experience**, v.20, n.8, p.833-849, Aug, 1990.

FRIEDMAN, D. P.; WISG, D. S. Aspects of applicative programming for parallel processing. **IEEE Transaction on Computers**, New York, v.C-27, n.4, p.289-296, Apr, 1978.

GANNON, O; ROSENDALE,J.V. On the impact of communication complexity on the design of parallel numerical algorithms. **IEEE Transactions on computers.** New York, v.C-33, n.12, p.1180-1194, Dec, 1984.

GIVE'ON, Y. S. Is recursion well defined? **Computer & Educations**, v.14, n.1, p.35-41, 1990.

GIVE'ON, Y. S. Teaching recursive programming using parallel multiturtle graphics. **Computer & Educations**, v.16, n.3, p.267-280, 1991.

HEATH,L; ROSENBERG,A; SMITH,B.T. The physical mapping problem for parallel architectures. **Journal of the ACM** New York,v.35,n.3, p.603-634,July, 1988.

HOROWITZ, E.; ZORAT, A. Divided-and-conquer for parallel processing.**IEEE Transactions on Computers**, New York, v.C-32, n.6, p.582-585, June, 1983.

KUMAR,M. Measuring parallelism im computation intensive scientific/engieneering applications. **IEEE Transactions on computers.** New York, v.37, n.9, p.1088-1098, Sep, 1988.

KUNG, H. T. The structure of parallel algorithms. **Advances in Computers**, New York, v.19, p.65-112, 1980.

LEE,S.Y; AGGARWAL,J.K A mapping strategy for parallel processing. **IEEE Transactions on computers**, New York, v.C36, n.4,p.433-442, Apr, 1987.

LUNDSTROM,S.P.;Lawrie, D.H. Experiences with distributed systems. **IEEE Software.** New York, p.5-6, May, 1985.

McKEAG, R. M.; MILLIGAN, P. An experiment in parallel program design. **Software: Practice & Experience**, v.10, n.9, p.687-696, Sep, 1980.

MOITRA, A; IYENGAR, S. S. Parallel algorithms for some computational problems. **Advances in Computers**, New York, v.26, p.93-153, 1987.

NICOL, D. M.; D'HALLARON, D. Z. Improved algorithms for mapping pipeline and parallel computations. **IEEE Transactions on Computers**, New York, v.C-40, n.3, p.295-306, Mar, 1991.

PADUA, D.A;WOLFE,M.J. Advanced computer optimizations for supercomputers. **Communications of the ACM**, New York, v.29, n.12,p.1184-1201, Dec, 1986.

POLYCHRONOPOULOS,C.D; BANERJEE,U. Processor allocation for horizontal and vertical parallelism and related speedup bounds. **IEEE Transactions on computers.** New York, v.C-36, n.4,p.410-420, Apr, 1987.

PRATT, T. W. PISCES: An environment for parallel scientific computation. **IEEE Software**, New York, v.2, n.4, p.7-20, July, 1985.

RABHI,F.A; MANSONG,G.A Divide and conquer and parallel graph reduction. **Parallel computing**, Amsterdam,v.17,n2&3,p.189-205, April, 1991.

RADUE, J. E.; MULLINS, J. M. Solving synchronization problems using semaphores. **Software: Practice & Experience**, v.5, n.1, p.54-64, Jan/Mar, 1975.

RAMAKRISHNAN,I.V; BROWNE,J.C A paradigm for the design of parallel algorithms with applications. **IEEE Transactions on Software engieneering** New York, v.SE-9,n.4, p.411-415, July, 1983.

SEGAL, Z.; RUDOLPH, L. PIE: A programming and instrumentation environment for parallel processing. **IEEE Software**, New York, v.2, n.6, p.22-37, Nov, 1985.

SMITH, D. R. The design of divide-and-conquer algorithm. **Science of Computer Programming**, Amsterdam, v.5, n.1, p.37-58, Feb, 1985.

SPECIAL ISSUE ON Parallelism. **Communication of the ACM**, New York, v.29, n.12, p.1165-1239, Dec, 1986.

WELCH, P. H.Parallel assignment revisited. **Software: Practice & Experience**, v.13, n.12, p.1117-1180, Dec, 1983.

WETTSTEIN, H. The implementation of synchronizing operations in various environments. **Software: Practice & Experience**, v.7, n.1, p.115-126, Jan/Mar, 1977.

C3.4 COMPLEXIDADE DE ALGORITMOS PARALELOS

CHUNG, K.; LUCCIO, F.; WONG, C. K. On the complexity of sorting in magnetic bubble memory systems. **IEEE Transactions on Computers**, New York, v.C-29, n.7, p.553-563, July, 1980.

COOK, S. A. An overview of computational complexity. **Communication of the ACM**, New York, v.26, n.6, p.400-408, June, 1983.

GATI,G. The complexity of Solving polynomial equations by quadrature. **Journal of the ACM**, New York, v.30,n.3, p.637-640, July, 1983.

HOPCROFT,J; PAUL,W.; VALIANT,L. On time versus space. **Journal of the ACM** New York,v.24,n.2, p.332-337,Apr, 1977.

HYAFIL,L; KUNG,H.T. The complexity of parallel evaluation of linear recurrences. **Journal of the ACM** New York,v.24,n.3, p.513-521,July, 1977.

JONES,N.; MUCHNICK,S.S The complexity of finite memory programs with recursion. **Journal of the ACM** New York,v.25,n.2, p.312-321,Apr, 1978.

KARP, R. M. Combinatories, complexity and randomness. **Communication of the ACM**, New York, v.29, n.2, p.97-109, Feb, 1986.

KARP, R. M. Complexity and parallel processing. **Communication of the ACM**, New York, v.29, n.2, p.110-117, Feb, 1986.

KARP, A. H.; FLATT, H. P. Measuring parallel processor performance. **Communication of the ACM**, New York, v.23, n.5, p.539-543, May, 1990.

KRUSCAL, C. P.; RUDOLPH, L.; SNIR, M. A Complexity theory of efficient parallel algorithms. **Theoretical Computer Science**, v.71, n.1, p.95, Mar, 1990.

LAWRIE, D. H. ; SAMEH A. H. The computation and communication complexity of a parallel banded system solve. **ACM Transactions on the Mathematical Software**, New York, v.10, n.2, p.185-195, June, 1984. (v.11, n.2, p.188, June, 1985)

LEIGHTON,T. Tight bounds on the complexity of parallel sorting. **IEEE Transactions on computers**, New York, v.C-34, n.4, p.344-354, Apr, 1985.

LOGRIPPO,L. Renamings, maximal parallelism, and space-time tradeoff in program schemata. **Journal of the ACM** New York,v.26,n.4, p.819-833,Oct, 1979.

MAHER,D. The complexity of the some problems on subsequences and supersequences. **Journal of the ACM** New York,v.25,n.2, p.322-336,Apr, 1978.

MATE,A Nondeterministic polynomial-time computations and models of arithmetic. **Journal of the ACM** New York,v.37,n.1, p.175-193,Jan, 1990.

PIPPENGER,N.; FISCHER,M.J. Relations among complexity measures. **Journal of ACM** New York, v.26, n.2,p.361-381, Apr, 1979.

SAVITCH,W.J; STIMSON,M.J. Time bounded random access machines with parallel processing. **Journal of the ACM** New York,v.26,n.3, p.103-118,JulY, 1979.

VITTER, J.S; SIMONS, R.A New Class for parallel complexity: a study of unification and other complete problems for P. **IEEE Transactions on computers**, New York, v.C-35, n.5, p.403-418, May, 1986.

WEIDE, B. A survey of analysis techniques for discrete algorithms. **Computing surveys** v.9, n.4, p.291-313, Dec, 1977.

WELDE, B. A survey of analysis techniques for discrete algorithms.**ACM Computing Surveys**, New York, v.9, n.4, p.291-313, Dec, 1977.

C3.5 ALGEBRA LINEAR (SISTEMAS DE EQUACOES & MATRIZES)

ARNOLD, L. P.; PARR, M. I.; DEWE, M. B. An efficient parallel algorithm for the solution of large sparse linear matrix equations. **IEEE Transactions on Computers**, New York, v.C-32, n.3, p.265-273, Mar, 1983.

AYKANAT,C; OZGUNER,F;ERCAL,F; SADAYAPPAN.P Iterative algorithms for solution of large sparse systems of linear equations on Hypercube. **IEEE Transactions on computers** New York, v.37,n.12, p.1554-1568,Dec, 1988.

BAMPIS,E; KONIG,J.C Impact of communication on the complexity of the parallel Gaussian elimination. **Parallel computing**, Amsterdam, v.17, n.1,p.55-61, April, 1991.

BARLOW, R. H.; EVANS, D. J. Parallel algorithms for the iteractive solution to linear systems. **Computer Journal**, London, v.25, n.1, p.56-60, Feb, 1982.

BAR-ON, I. A practical parallel algorithm for solving band simetric positive and definite systems of linear equations. **ACM Transactions on the Mathematical Software**, v.13, n.4 , p.323-332, Dec, 1987.

BONDELI,S. Divide and conquer: a parallel algorithm for the solution of a tridiagonal linear system of equations. **Parallel computing**, Amsterdam, v.17, n.4&5, p.419-434, July, 1991.

BRUGNANO.L A parallel solver for tridiagonal linear systems for distributed memory parallel computers. **Parallel computing**, Amsterdam, v.17, n.9, p.1017-1023, Nov, 1991.

CHEN, C.; KUCK, D. J.; SAMEH, A. H. Pratical parallel band triangular systems solvers. **ACM Transactions on the Mathematic Software**, New York, v.4, n.3, p.270-277, Sep, 1978.

COSNARD,M.; ROBERT,Y. Complexity of parallel QR factorization. **Journal of the ACM** New York,v.33,n.4, p.712-723,Oct, 1986.

EVANS, D. J.; HATZOPOULOS, M. The parallel calculation of the eigenvalues of a real matrix A. **Computers & Mathematics**, Oxford, v.4, n.3, p.211-218, 1978.

HELLER,D.E; STEVENSON,D.K; TRAUB,J.F. Accelerated iterative methods for the solution of tridiagonal systems on parallel computers. **Journal of the ACM** New York,v.23,n.4, p.636-654,Oct, 1976.

KEHAT, E.; SCHACHAM, M. A direct method for the solution of large sparse systems of linear equations. **Computer Journal**, London, v.19, n.4, p.353-359, Nov, 1976.

LAMBIOTTE, J.J.; VOIGT, R. S. The solution of tridiagonal linear systems on the COC Star-100 computer. **ACM Transactions on the Mathematical Software**, New York, v.1, n.4, p.308-329, Dec, 1975.

LAWRIE, D. H. ; SAMEH A. H. The computation and communication complexity of a parallel banded system solve. **ACM Transactions on the Mathematical Software**, New York, v.10, n.2, p.185-195, June, 1984. (v.11, n.2, p.188, June, 1985)

MULLER,S.M; CHEERER,D. A method to parallelize tridiagonal solvers. **Parallel Computing**, Amsterdam, v.17, n.2&3, p.181-188, June, 1991.

MUKAI, H. Parallel algorithm for solving systems of nonlinear equations. **Computers & Mathematics**, Oxford, v.7, n.3, p.235-235-250, 1981.

PAPRZYCK,M; GLADWELL,I. Solving almost block diagonal systems on parallel computers. **Parallel computing**, Amsterdam, v.17, n.2&3, p.133-153, June, 1991.

PETERS,A. Sparse matrix vector multiplication techniques on the IBM 3090VF. **Parallel computing**, Amsterdam, v.17,n.12,p.1409-1424, Dec, 1991.

RODRIGUE,G.H; MADSEN,N.K.;KARUSH,J.I Odd-even reduction for banded linear equations. **Journal of the ACM** New York,v.26,n.3, p.72-81, July, 1979.

SKEEL,R.D. Scaling for numerical stability in Gaussian elimination. **Journal of the ACM** New York,v.26,n.3, p.494-526, July, 1979.

SLOTNICK,D.L; SAMEH,A. Numerical calculation and computer design. **Computer & Mathematics with applications** London, v.4, n.3, p.201-210, 1978.

STPICZYNSKI,P. Parallel Cholesky factorization on orthogonal multiprocessors. **Parallel Computing**, Amsterdam, v.18, n.2, p.213-219, Feb, 1992.

STONE, H. S. Parallel tridiagonal equations solves. **ACM Transactions on Mathematical Software**, New York, v.1, n.4, p.289-307, Dec, 1975.

TERVOLA,P; YEUNG,W. Parallel Jacobi algorithm for matrix diagonalisation on transputer networks. **Parallel computing** Amsterdam, v.17, n.2&3, p.155-163, June, 1991.

TRAUB, J.F; WOZNIAKOWSKI,H. On the optimal solution of large systems. **Journal of the ACM** New York,v.31,n.3, p.545-559, July, 1984.

WANG, H. H. A parallel method for tridiagonal equations. **ACM Transactions on the Mathematical Software**, New York, v.7, n.2, p. 170-183, June, 1981.

C3.6 EQUAÇÕES ALGEBRICAS (ZEROS DE FUNÇÕES)

BARLOW, R. H.; EVANS, D. J. A parallel organization of the bisection algorithm. **Computer Journal**, London, v.22, n.3, p.267-269, Aug, 1979.

BOJANCZYK,A. Optimal asynchronous Newton method for the solution of nonlinear equations. **Journal of the ACM** New York,v.31,n.4, p.792-803, Oct, 1984.

BREVER, S.; ZWAS, G. Polynomial iterations for root extraction. **Computer & Educations**, v.12, n.2, p.289-300, 1988.

FREEMAN,T.L; BANE,M.K Asynchronous polynomial zero-finding algorithms. **Parallel computing** Amsterdam,v.17,n.6&7,p.673-681, Sep, 1991.

GARGANTINI, I. Comparing parallel Newton's method with parallel Laguerre's method. **Computers & Mathematics**, Oxford, v.2, n.3/4, p.201-206, 1976.

GATI,G. The complexity of Solving polynomial equations by quadrature. **Journal of the ACM**, New York, v.30,n.3, p.637-640, July, 1983.

JENKINS, M. A.; TRAUB, J. F. Principles for testing polynomial zero finding programms. **ACM Transactions on Mathematical Software**, New York, v.1, n.1, p.26-34, Mar, 1975.

LORD,R.E; KOWALIK,J.S.; KUMAR,S.P. Solving linear algebric equations on a MIMD computer. **Journal of the ACM** New York,v.30,n.1, p.103-117, Jan, 1983.

MAEDER,A.J; WYNTON,S.A Some parallel methods for polynomial root-finding. **Journal of computational and applied Mathematics** Amsterdam, v.18, p.71-81, 1987.

MUKAI, H. Parallel algorithm for solving systems of nonlinear equations. **Computers & Mathematics**, Oxford, v.7, n.3, p.235-235-250, 1981.

WASILKOWSKI,G.W n-Evaliation conjecture for multipoint iterations for the solution of scalar nonlinear equations. **Journal of the ACM** New York,v.28,n.1, p.71-80, Jan, 1981.

WILF,H.S. A global bisection algorithm for computing the zeros of polynomial in the complex plane. **Journal of the ACM** New York,v.25,n.3, p.415-420, July, 1978.

C3.7 EQUAÇÕES DIFERENCIAIS

BACCIOTTI, A.; BOIERI, P.; MORONI, P. A computer approach to nonlinear planar systems of ODE'S. **Computer & Educations**, 11, n.4, p.253-265, 1987.

BUTCHER,J.C. A transformed implicit runge-kutta method. **Journal of the ACM** New York,v.26,n.4, p.731-738, Oct, 1979.

FRANKLIN, M. A. Parallel solution of ordinary differential equations. **IEEE Transactions on Computers**, New York, v.C-27, n.5, p.413-420, May, 1978.

GEAR, C. W. Runge-Kutta starters for multistep methods. **ACM Transactions on the Mathematical Software**, New York, v.10, n.2, p.161-184, June, 1984.

MACHURA, M. M; SWEET, R. A. A survey of software for partial differential equations. **ACM Transactions on Mathematical Software**, New York, v.6, n.4, p.461-488, Dec, 1980.

C3.8 PROBLEMAS DE COMPARAÇÃO (SORTING)

AGGARWAL, A.; VITTER, J. S. The input/output complexity of sorting and related problems. **Communication of the ACM**, New York, v.31, n.9, p.1116-1127, Sep, 1988.

BAUDET, G.; STEVENSON, D. Optimal sorting algorithms for parallel computers. **IEEE Transactions on Computers**, New York, v.C-27, n.1, p.84-87, Jan, 1978.

BENTLEY, J. L. How to sort. **Communication of the ACM**, New York, v.27, n.4, p.287-291, Apr, 1984.

BITTON, D.; DeWITT, D. J.; HSIAD, D. K.; MENON, J. A Taxonomy of parallel sorting. **ACM Computing Surveys**, New York, v.16, n.3, p.287-318, Sep, 1984.

CHUNG, K.; LUCCIO, F.; WONG, C. K. On the complexity of sorting in magnetic bubble memory systems. **IEEE Transactions on Computers**, New York, v.C-29, n.7, p.553-563, July, 1980.

COLE, R. Slowing down sorting networks to obtain faster sorting algorithms. **Journal of the ACM** New York, v.34, n.1, p.200-208, Jan, 1987.

COLE, R.; SIEGEL, A. Optimal VLSI circuits for sorting. **Journal of the ACM** New York, v.35, n.4, p.777-809, Oct, 1988.

DITTERT, E.; O'DONNELL, M.J. Lower bounds for sorting with realistic instructions sets. **IEEE Transactions on computers**. New York, v.C-34, n.4, p.311-317, Apr, 1985.

ER, M. C. A parallel computation approach to topological sorting. **Computer Journal**, Londol, v.26, n.4, p.294-295, Nov, 1983.

GUAN, K.; LANGSTON, M. A. Time-space optimal parallel marging and sorting. **IEEE Transactions on Computers**, New York, v.C-40, n.5, p.596-602, May, 1991.

HIRSCHBERG, D. S. Fast parallel sorting algorithms. **Communication of the ACM**, New York, v.21, n.8, p.657-661, 1986.

KUNG, H. T.; THOMPSOM, C. D. Sorting on a mesh-connect of parallel computer. **Communication of the ACM**, New York, v.20, n.4, p.260-279, Apr, 1977.

LAKSHMIVARAHAN, S.; SUDARSHAN, K.; MILLER, L. L. Parallel sorting algorithms. **Advances in Computers**, New York, v.23, p.295-254, 1984.

LEIGHTON, T. Tight bounds on the complexity of parallel sorting. **IEEE Transactions on computers**, New York, v.C-34, n.4, p.344-354, Apr, 1985.

MANACHER, G.K. Significant improvements to the Hwang-Lin merging algorithms. **Journal of the ACM** New York, v.26, n.3, p.434-440, July, 1979.

MANACHER, G.K The ford-Johnson sorting algorithm is not optimal. **Journal of the ACM** New York, v.26, n.3, p.441-456, July, 1979.

MANACHER, G.K; BUI, T.D; MAI, T. Optimum combinations of sorting and merging. **Journal of the ACM** New York, v.36, n.2, p.290-334, Apr, 1989.

MARSLAND, T. A.; CAMPBELL, M. Parallel search of strongly ordered game trees. **ACM Computing Surveys**, New York, v.14, n.4, p.533-552, Dec, 1982.

McGLINN, R. J. A parallel version of Cook and Kim's algorithm for pre-sorted lists. **Software: Practice & Experience**, v.19, n.10, p.917-930, Oct, 1989.

NASSIMI, D.; SAHNI, S. Biotonic sort on a mesh-connect parallel computer. **IEEE Transactions on Computers**, v.C-28, n.1, p.2-7, Jan, 1979.

NASSIMI,D; SAHNI,S. Parallel permutation and sorting algorithms and a new generalized connection network.**Journal of the ACM** New York,v.29,n.3, p.642-667, July, 1982.

PREPARATA, F. P. New parallel-sorting schemes. **IEEE Transactions on Computers**, New York, v.C-27, n.7, p.669-673, July, 1978.

WHEAT,M. EVANS,D.J An efficient parallel sorting algorithm for shared memory multiprocessors. **Parallel computing**, Amsterdam, v.18, n.1, p.91-102, Jan, 1992.

C3.9 PONTO FLUTUANTE E ARITMETICA INTERVALAR

BUSTOZ, J.; FELDSTEIN, A; GOODMAN, R; LINNAINMAA, S Improved trailing digits estimatives applied to optimal computer arithmetic. **Journal of the ACM** New York, v.26, n.4, p.716-732, Oct, 1979.

CLENSHAW, C.W; OLVER, F.W.J Beyond floating point **Journal of the ACM** New York, v.31, n.2, p.319-328, Apr, 1984.

COLE, A. J.; MORRISON, R. Triplex: a system for interval arithmetic. **Software: Practice & Experience**, v.12, n.4, p.341-350, Apr, 1982.

FARNUN, C. Compiler support for floating-point computation. **Software: Practice & Experience**, v.18, n.7, p.701-709, July, 1988. (v.18, n.12, p.1113-1194, 1988)

JANSEN, P. ; WELDNER, P. High-accurancy arithmetic software - some tests of the ACRITH problems - solving routines. **ACM Transactions on the Mathematical Software**, New York, v.12, n.1, p.62-70, Mar, 1986.

McCLELLAN, M. T. The exact solution of linear operations with rational function coefficients. **ACM Transactions on the Mathematical Software**, New York, v.3, n.1, p.1-25, Mar, 1977.

NEDELCHEVA, T. K.; ELENKOVA, N. G.; KOSTHDINOVA, L. S. Program for investigating the effect of measurement errors on the accurancy of the calculated results in classical chemical analysis. **Computer & Educations**, v.13, n.4, p.363-366, 1989.

NELSON, J. M.; COHN, C. E. Parallel processing in FORTRAN with floating-point hardware. **Software: Practice & Experience**, v.5, n.1, p.65-68, Jan/Mar, 1975.

RALL, L. B. Differentiation in Pascal-SC type gradiente. **ACM Transactions on the Mathematical Software**, New York, v.10, n.2, p.161-184, June, 1984.

SCHECHTER, M. Computer summation of series: floats vs reals. **Computer & Educations**, v.14, n.6, p.489-493, 1990.

VERMA, S. B.; SHARAN, M. Multiple precision floating-point operation in FORTRAN. **Software: Practice & Experience**, v.10, n.3, p.163-173, 1980.

YOHE, J. M. Software for interval arithmetic: a reasonably portable package. **ACM Transactions on the Mathematical Software**, New York, v.5, n.1, p.50-63, Mar, 1979.

C3.10 APLICAÇÕES

CONERY, J. S.; KIBLER, D. F. AND parallelism and non-determinism in logic programs. **New Generation Computing**, v.3, n.1, p.43-70, 1985.

BAUDET,G.M Asynchronous iterative methods for multiprocessors. **Journal of the ACM** New York,v.25,n.2, p.226-244, Apr, 1978.

BRENT,R. Fast multiple-precision evaluation of elementary functions. **Journal of the ACM** New York,v.23,n.2, p.242-251, Apr, 1976.

BRENT,R; KUNG,H.T. Fast algorithms for manipulation for formal power series. **Journal of the ACM** New York,v.25,n.4, p.581-595, Oct, 1978.

BUI,T.D. Some A-stable and l-stable methods for the numerical integration of stiff ordinary differential equations. **Journal of the ACM** New York,v.26,n.3, p.483-493, July, 1979.

GAL-EZER, J.; ZWAS, G. The computational potencial of rational approximations. **Computer & Educations**, v.11, n.1, p.33-46, 1987.

KANEDA, Y.; MATSUDA, H. Introduction of dos in predicate to Prolog. **New Generation Computing**, v.8, n.2, p.163-189, 1990.

KUNG,H.T. New algorithms and lower bounds for the parallel evaluation of certain rational expression and recurrences. **Journal of the ACM** New York,v.23,n.2, p.252-261, Apr, 1976.

KUNG,H.T;TRAUB,J.F. All algebric functions can be computed fast. **Journal of the ACM** New York,v.25,n.2, p.245-260, Apr, 1978.

LAW, A. G.; MAGUIRE, R. B.; SABO, D. F. G.; SHUPARSKI, B. M. Computer voice support for visually nondicapped students. **Computer & Educations**, v.8, n.1, p.35-39, 1984.

McKENZIE, J. Interactive computer graphics for undergraduate science teaching. **Computer & Educations**, v.2, n.1-2, p.25-48, 1978.

NEDELCHEVA, T. K.; ELENKOVA, N. G.; KOSTHDINOVA, L. S. Program for investigating the effect of measurement errors on the accurancy of the calculated results in classical chemical analysis. **Computer & Educations**, v.13, n.4, p.363-366, 1989.

RICE, J. Parallel algorithmy for adaptive quadrature III program correctness. **ACM Transactions on the Mathematical Software**, New York, v.2, n.1, p.1-30, Mar, 1976.

SAHASRABUDHE,S.C; KULKARNI,A.D On solving fredholm integral equations on the first kind. **Journal of the ACM** New York,v.24,n.4, p.624-629, Oct, 1977.

WALKS, S. Computer vision - not for experts only. **Computer & Educations**, v.14, n.2, p.113-181, 1990.

YASUHARA, H.; NITADORI, K. Orbit: a parallel computing model of Prolog. **New Generation Computing**, v.2, n.3, p.277-288, 1984.

C3.11 SOFTWARE MATEMATICO

COX, M. G. Topic libraries for mathematical computation. **Software: Practice & Experience**, v.15, n.4, p.397-411, Apr, 1985.

FORCHERI, P.; LEMUT, E.; MOLFINO, M. T. The Graf system: an interactive graphic system for teaching mathematics. **Computers & Educations**, v.7, n.3, p.177-182, 1983.

GEAR, C. W. Numerical Software: Science or alchemy? **Advances in Computers**, New York, v.19, p.229-249, 1980.

HALIM, Z. A data-driven machine for OR-parallel evaluation of logic programs. **New Generation Computing**, v.4, n.1, p.5-33, 1986.

JANSEN, P. ; WELDNER, P. High-accurancy arithmetic software - some tests of the ACRITH problems - solving routines. **ACM Transactions on the Mathematical Software**, New York, v.12, n.1, p.62-70, Mar, 1986.

SAMEH, A.; SLOTMOK, D. L. Numerical calculation and computer design. **Computers & Mathematics**, Oxford, v.4, n.3, p.201-210, 1978.