Arranjos Sistólicos Redundantes
para
Grades de Computação

MARCOS TATSUO YAMAMOTO

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
PARA OBTENÇÃO DO TÍTULO DE MESTRE
EM
CIÊNCIA DA COMPUTAÇÃO

Área de Concentração: Ciência da Computação
Orientador: Prof. Dr. Kunio Okuda

— São Paulo, novembro de 2005 —

Arranjos Sistólicos Redundantes
para
Grades de Computação

Este exemplar corresponde à redação final
da dissertação devidamente corrigida e
defendida por Marcos Tatsuo Yamamoto
e aprovada pela comissão julgadora.

São Paulo, 30 de novembro de 2005

Banca examinadora:

- Prof. Dr. Kunio Okuda - IME-USP
- Prof. Dr. Siang Wun Song - IME-USP
- Profa. Dra. Liria Matsumoto Sato - EP-USP

Resumo

O número elevado de recursos computacionais sub-aproveitados em aglomerados heterogêneos de computadores pessoais, bem como de estações de trabalho conectadas por uma rede de alta velocidade, viabiliza a construção de um ambiente de sistemas distribuídos que oferece grande capacidade computacional. Neste trabalho, vamos mostrar que podemos obter um algoritmo paralelo não trivial a partir do método de transformação de dependências. Este algoritmo paralelo passará por uma série de transformações com o intuito de adequar-se ao modelo realístico CGM, que introduziu o conceito de granularidade grossa em rodadas de superpassos de comunicação. O resultado desta operação é um algoritmo sistólico paralelo adequado a ambientes distribuídos com elementos homogêneos devido à grande dependência de dados. O objetivo deste estudo é demonstrar como a aplicação de redundância em algoritmos sistólicos pode trazer benefícios em ambientes heterogêneos. A demonstração da aplicação de redundância será baseada no algoritmo sistólico de multiplicação de matrizes que recebe duas matrizes $n \times n$ e executa a multiplicação em p^2 processadores. Com uma redundância de grau r , o problema passa a ser processado por $p \times p \times r$ processadores, agrupados em p^2 grupos de computação redundante. Em nossos experimentos executados no cluster Biowulf de 16 nós, obtivemos um pequeno acréscimo de overhead para o controle de redundância. Contudo, a execução do algoritmo sistólico redundante em ambientes heterogêneos simulados alcançou com grande êxito um desempenho estável e até mesmo superior ao algoritmo original.

Abstract

The huge number of computing resources is not use adequately in heterogeneous groups of personal computers and workstations connected by a high speed lan. It makes it possible to build up a computer's system environment distributed and this will allow an efficient computing capacity. In this study we are going to prove the feasibility of getting a paralel algorithm not trivial using the transformation method of dependence. This parallel algorithm will be submitted to a number of transformation in order to adjust it to a CGM realistic model, which introduced the consistent granularity concept in many steps processing communication. The operation's result is a parallel systolic algorithm adjusted to an environment with homogeneous elements due to the strong data dependence. The goal of this study is to prove how a redundancy application in systolic algorithm could come up with benefits in a heterogeneous environment. The demonstration will be based on a systolic algorithm that receives as an input two $n \times n$ matrices and executes its product with p^2 processors. Using a redundancy of r degrees, the problem will be processed by $p \times p \times r$ processors grouped in p^2 groups of redundancy computing. In our experiments on the Biowulf cluster of 16 nodes, we obtained a tiny overhead increase in the redundancy control, however, in simulated heterogeneous environment we obtained an stable performance and better than the original algorithm.

Sumário

Introdução	2
1 Grades de Computação	4
2 Síntese de Algoritmos Sistólicos	8
2.1 Um Exemplo de Algoritmos Sistólicos	9
2.1.1 Solução Sistólica de Kung-Leiserson	10
2.1.2 Solução Sistólica de Weiser-Davis	11
2.1.3 Solução Sistólica de Okuda-Song	12
2.2 Dependência entre computações	12
2.3 Vetores de Dependência	14
2.4 Transformações de Dependências	14
2.4.1 Equações de Recorrência Uniforme	17
2.4.2 Grafos de Dependências	17
2.4.3 Função de Tempo τ	18
2.4.4 Função de Alocação α	20
3 Algoritmos Sistólicos para Grades de Computação	22
3.1 Sistemas de Computação Paralela e Distribuída	22
3.2 Adaptando para Ambientes Paralelos e Distribuídos	23
4 Tolerância a Falhas	25
4.1 Defeitos em Sistemas de Computação	26
4.2 Tolerância a falhas ou dependabilidade?	26
4.3 Conceitos: Falha, erro e defeito	27
4.4 Dependabilidade	28

4.4.1	Confiabilidade (Reliability)	29
4.4.2	Disponibilidade (Availability)	29
4.4.3	Segurança de Funcionamento (Safety)	30
4.4.4	Outros atributos	30
4.5	Técnicas para alcançar dependabilidade	31
4.5.1	Detecção, localização e reconfiguração	31
4.5.2	Mascaramento de falhas	32
4.6	Redundância	33
4.6.1	Redundância de informação	33
4.6.2	Redundância temporal	34
4.6.3	Redundância de hardware	34
4.6.4	Redundância de software	35
5	Redundância em Algoritmos Sistólicos para Grades de Computação	37
5.1	Adaptando a Estrutura	38
5.2	Um algoritmo redundante simples	39
5.3	Um algoritmo redundante mais sofisticado	41
5.4	Aplicando a Redundância	43
5.5	Implementação do Algoritmo	44
6	Resultados Experimentais	51
6.1	Ferramentas de Trabalho	51
6.2	Ambiente de Testes - Biowulf/IME	52
6.3	Multiplicação de Matrizes	52
6.3.1	Tempos em Ambientes Homogêneos	54
6.3.2	Tempos em Ambientes Heterogêneos	54
6.4	Comparativo Sistólico Com e Sem Redundância	55
7	Conclusão	58
	Bibliografia	61

Introdução

Uma grande distribuição de excelentes computadores pessoais pôde ser observada nas últimas décadas. Certamente, os computadores auxiliam usuários e empresas a melhorarem a eficiência de seus trabalhos, além de servirem como fontes de pesquisas, diversão, trocas de mensagens, entre outros motivos. Conseqüência ou não, a grande quantidade de computadores distribuídos se deve a melhor qualidade dos componentes de hardware (processador, memória, etc) e software mais intuitivos para os usuários em geral. Hoje, a Internet apresenta uma estrutura de milhões de computadores interconectados e distribuídos por todos os cinco continentes.

A extensa quantidade de computadores interconectados através da internet possui uma capacidade de cálculo e de armazenamento de alta escala, comparáveis aos supercomputadores paralelos. Os grandes avanços em redes de computadores viabilizaram a execução de aplicações paralelas em processadores geograficamente distribuídos, pertencentes a diversos domínios administrativos, resultando em uma grande estrutura conhecida como grades de computação. O capítulo 1 descreve uma breve introdução deste conceito promissor.

Existem problemas dos mais diversos gêneros que um computador poderia resolver. Porém, muitos destes problemas necessitam de uma performance diferenciada, exigindo uma grande quantidade de computação ou mesmo um rápido tempo de resposta. Uma solução para atender este tipo de problema foi a implementação de chips capazes de realizar computações paralelamente. No capítulo 2, vamos conhecer a estrutura de arranjos sistólicos que processa computações paralelas com o auxílio do método de transformação de dependências.

Somando o conceito de grades de computação com o conhecimento da técnica de arranjos sistólicos, o capítulo 3 descreve uma série de adequações usadas para

adaptar o algoritmo em um ambiente distribuído, como as grades de computação.

A técnica de arranjos sistólicos não prevê a falha física de componentes de computação durante a execução de suas aplicações. Mas diversas técnicas podem ser empregadas em sistemas computacionais com o objetivo de oferecer sistemas mais confiáveis e mais disponíveis. O capítulo 4 introduz o conceito de redundância e tolerância a falhas.

Devido à sua forte dependência de computações, os algoritmos sistólicos não apresentam um desempenho satisfatório em ambientes heterogêneos como aglomerados de computadores e grades de computação. No capítulo 5, mostraremos como é possível adaptar o algoritmo sistólico para se obter uma estabilidade maior, podendo até mesmo alcançar desempenhos superiores usando a técnica de redundância.

Os algoritmos sistólicos redundantes foram testados em um cluster de 8 elementos. Os experimentos com redundância tiveram resultados satisfatórios em ambientes homogêneos e um ótimo comportamento em ambientes heterogêneos. Maiores detalhes sobre os experimentos podem ser vistos no capítulo 6.

Por último, temos o capítulo 7, onde apresentamos algumas observações, dificuldades e conclusões obtidas durante o desenvolvimento desta dissertação.

Capítulo 1

Grades de Computação

O objetivo desta seção é introduzir o conceito de Grades de Computação, a demanda por este tipo de arquitetura, além de citar alguns projetos existentes na área.

Todos os dias, inúmeros profissionais e pesquisadores de áreas diversas depararam com problemas que necessitam de uma grande computação para se chegar a uma solução ou a um resultado desejado. A resolução de problemas computacionalmente difíceis é hoje de extrema importância na área de ciências como a física, química e biologia, e não menos importante, para empreendimentos comerciais como simulações mercadológicas, sistema financeiro, meteorologia e diversos outros temas variados. Infelizmente, apenas parte desses problemas são resolvidos devido, entre outros motivos, a falta de recursos financeiros.

Para viabilizar o tempo de resposta desses problemas, foram desenvolvidas máquinas paralelas com alto poder de processamento constituídas por múltiplos processadores. Porém, o alto custo para a aquisição de máquinas desse porte não se enquadra no orçamento disponível da maioria das empresas ou centros de pesquisa.

Nas décadas de 80 e 90, o preço acessível e a qualidade de serviço oferecido pelos computadores pessoais gerou uma enorme demanda por este tipo de equipamento. Para atender a essa demanda, e a expectativa de um investimento rentável, novas empresas surgiram para abocanhar uma fatia desse mercado. Como resultado dessa concorrência, houve uma melhoria rápida e constante dos diversos componentes de um computador pessoal, tais como; processador, memória, vídeo, entre outros.

Paralelamente a este fenômeno, houve um crescimento do número de usuários, tanto profissionais como domésticos. Com a popularização, surgiu a necessidade de organizar os computadores de forma estruturada para aproveitar ao máximo os recursos disponíveis como, por exemplo, compartilhamento de informações e dispositivos de impressão. Para melhorar a qualidade dos serviços oferecidos por essa estrutura, conhecida por rede local de computadores, novos dispositivos de comunicação foram desenvolvidos com a finalidade de obter um aumento na performance, confiabilidade e distância física entre os dispositivos da rede, além do incremento de dispositivos participantes.

A combinação da melhoria de desempenho dos processadores e a velocidade oferecida pelos dispositivos de comunicação entre computadores, despertou interesse de empresas e centros de pesquisa da área de computação paralela. A grande quantidade de computadores interconectados por meio de uma rede rápida e confiável, possibilitou o uso dos computadores como nós de processamento e a rede como o dispositivo responsável pela comunicação entre os nós.

Utilizar uma rede de computadores onde diversos usuários compartilham recursos como plataforma para computação de aplicações exigentes, pode tornar o tempo total de execução inviável devido à problemas como, por exemplo, uma sobrecarga de processamento em um processador, ou a concorrência de tarefas pertencentes à outras aplicações, entre outros motivos.

Pensando em problemas como estes e levando em conta fatores financeiros, surgiu um novo tipo de super-computação com custo ainda acessível, o cluster. O cluster é composto por dois ou mais nós (computadores) interconectados por meio de uma rede trabalhando cooperativamente para atingir um objetivo comum, capaz de oferecer um grande potencial de computação gerenciado por uma entidade global centralizada. Dessa maneira, um cluster permite ao usuário controlar as tarefas que serão executadas, de acordo com a necessidade de performance. Com esse conceito, muitos clusters já se encontram na lista dos mais velozes super-computadores do mundo. Apesar de ser uma opção com custo mais acessível do que as máquinas paralelas, o custo de aquisição e manutenção de clusters dedicados ainda é alto, o que inviabiliza seu uso pela grande maioria das empresas de pequeno e médio porte, instituições governamentais e de pesquisa dos países em desenvolvimento.

Ainda assim, muitas instituições de ensino e pesquisa, e também uma infinidade de computadores dispersos e heterogêneos continuam com seus recursos ociosos. O conceito de Grades de Computação visa oferecer uma diversidade de recursos computacionais e repositório de informações capaz de atender as necessidades de diversas aplicações exigentes de tal modo que permita coordenar o processamento das tarefas de forma transparente ao usuário.

Ian Foster [14] define o conceito da seguinte maneira: “‘Grades de Computação são sistemas de suporte à execução de aplicações paralelas que acoplam recursos heterogêneos distribuídos, oferecendo acesso consistente e barato aos recursos, independente de sua posição física. A tecnologia de Grades de Computação possibilita agregar recursos computacionais variados e dispersos em um único supercomputador virtual, acelerando a execução de várias aplicações paralelas”’.

O nome grade de computação vem de uma metáfora com a rede elétrica. A rede elétrica possui malhas de interligação dos sistemas de energia elétrica, onde os usuários participantes apenas conectam seus equipamentos elétricos através de uma interface padrão, abstraindo totalmente o processo de geração e transmissão da energia elétrica. De maneira análoga, a grade de computação seria a infra-estrutura de hardware e software necessária para que os usuários precisem apenas conectar o seu computador à rede para obter o poder computacional, sem se preocupar com a forma que o serviço está sendo executado.

Para disponibilizar os serviços (energia elétrica, na analogia) da Grade de Computação, é necessário desenvolver uma série de normas, padrões, protocolos para comunicação e processamento, e também ferramentas de software. A padronização da tecnologia de Grade de Computação já vem ocorrendo há bastante tempo, e devido à extrema importância e urgência de uso, os esforços passaram a ter uma prioridade muito maior nesses últimos anos. Como exemplo de todo esse esforço, podemos citar projetos como o Globus, Legion e OpenGrid.

Um grande problema conhecido na resolução de algoritmos não trivialmente paralelizáveis se deve ao alto custo de comunicação entre os diversos nós de processamento. Nos moldes do conceito de Grade de Computação, o custo de comunicação passa a ser um fator agravante que pode prejudicar o tempo total gasto na execução de determinadas aplicações paralelas. Já conhecida na literatura de algoritmos pa-

paralelos e distribuídos, uma linha a ser seguida para evitar esse tipo de problema é elaborar uma estratégia no algoritmo de modo a minimizar o custo de comunicação, diminuindo o número de rodadas de comunicação. Maiores detalhes sobre algoritmos paralelos podem ser encontrados em [10].

Capítulo 2

Síntese de Algoritmos Sistólicos

Neste capítulo, vamos mostrar a estrutura e a lógica que envolve os algoritmos sistólicos. Também mostraremos o desenvolvimento de um método formal para sintetizar e gerar algoritmos sistólicos. Algumas boas referências sobre algoritmos sistólicos podem ser encontradas em [29, 9, 22, 23, 24, 13, 34].

O termo sistólico vem da palavra *sístole* que é o período de contração do coração que faz o sangue fluir em pulsos através das artérias e veias. Fazendo uma analogia com o estudo em questão, *arrays sistólicos* são estruturas computacionais onde células de processamento computam e bombeiam dados periodicamente por meio dos canais de comunicação, formando ondas de computação.

Inicialmente, a técnica de Algoritmos Sistólicos foi utilizada em circuitos de alta densidade com a tecnologia VLSI (Very Large Scale Integration). Nestes circuitos, as computações são processadas pelas células e os dados são transmitidos entre células vizinhas através de canais de comunicação. Visando melhorar a performance da computação, projeta-se algoritmos com granularidade fina para que o tempo de comunicação seja o menor possível, diminuindo a ociosidade nas células de processamento.

Uma propriedade importante em Algoritmos Sistólicos é a disposição regular das células de processamento e o baixo número de vizinhos com comunicação. Dessa maneira, o tempo gasto na rodada de comunicação é menor pois seria necessário enviar pacotes de dados a poucas células e, de preferência, enviar para aquelas que estejam localizadas próximas a célula remetente (onde o tempo de comunicação fosse

menor).

Como podemos verificar, a técnica admite o uso do modelo SPMD (Single-Program Multiple-Data) onde as instruções do algoritmo são executadas da mesma maneira em todas as células de processamento. Ou seja, em cada instante, todas as células estão processando os seus respectivos dados realizando sempre as mesmas instruções de execução. Essa propriedade facilita a implementação dos algoritmos em computadores paralelos e distribuídos, pois pode-se utilizar bibliotecas existentes e conhecidas como o MPI (Message Passing Interface) e o PVM (Parallel Virtual Machine). Desse modo, torna-se possível a migração da técnica de Arrays Sistólicos com a tecnologia VLSI para uma arquitetura de computadores paralelos, mantendo a essência da idéia original.

Dado que o objetivo dessa seção é apresentar o conceito de Algoritmos Sistólicos para se aplicar em Grades de Computação, vejamos algumas dificuldades que poderão ser encontradas no desenvolvimento da técnica. Um primeiro ponto a ser observado é o tamanho da granularidade utilizada. No circuito, a granularidade fina não chega a prejudicar o desempenho do algoritmo pois o tempo de comunicação entre as células é bem pequeno. Ao se aplicar a granularidade fina em uma Grade de Computação, temos um grande acréscimo no tempo de comunicação devido à operações realizadas na grade (que são desnecessárias no circuito) além da distância física muito superior entre as células de processamento. Uma solução para esse problema seria adaptar o algoritmo de forma que a granularidade passe a ficar mais grossa. Um outro ponto a ser observado é a possibilidade da existência de células de processamento heterogêneas, onde células rápidas ficam ociosas enquanto células lentas ficam sobrecarregadas, mesmo realizando a mesma tarefa.

2.1 Um Exemplo de Algoritmos Sistólicos

Kung e Leiserson [8] realizaram os primeiros trabalhos sobre o conceito de algoritmos sistólicos. Os autores propuseram uma solução para a computação da multiplicação de duas matrizes usando simples células de computação e uma granularidade muito fina de paralelismo. A seguir, a apresentação do problema.

Sejam duas matrizes $n \times n$ $A = (a_{ij})$ e $B = (b_{ij})$. Para simplificar a visualização,

vamos fixar $n = 3$. O produto das matrizes $C = (c_{ij})$

$$C = \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix}$$

pode ser expresso pelas seguintes equações de recorrência:

$$c_{ij}^1 = 0$$

$$c_{ij}^{k+1} = c_{ij}^k + a_{ik}b_{kj}$$

$$c_{ij} = c_{ij}^{n+1}$$

2.1.1 Solução Sistólica de Kung-Leiserson

Kung-Leiserson [8] apresentaram uma solução sistólica para a multiplicação de matrizes, onde cada célula possui três entradas a_{in} , b_{in} , c_{in} e três saídas a_{out} , b_{out} , c_{out} . A figura 2.1 representa a célula.

Figura 2.1: Célula do circuito de Kung-Leiserson

Em uma multiplicação de matrizes 3×3 , a disposição das células ficaria como ilustrado na figura 2.2.

Figura 2.2: Solução Sistólica de Kung-Leiserson

O algoritmo sistólico de Kung-Leiserson computa a multiplicação de duas matrizes $n \times n$ em tempo linear ($5n$ passos) usando $3n^2$ células.

2.1.2 Solução Sistólica de Weiser-Davis

Weiser-Davis apresentaram uma solução sistólica que inverte o fluxo dos valores de c , citado na solução de Kung-Leiserson.

A disposição das células é idêntica, mas a direção do fluxo foi alterada, como se pode ver na figura 2.3.

Figura 2.3: Solução Sistólica de Weiser-Davis

Desse modo, o algoritmo sistólico de Weiser-Davis realiza a multiplicação em $3n$

passos com o mesmo número de células, ou seja, $3n^2$ células.

2.1.3 Solução Sistólica de Okuda-Song

Com o objetivo de reduzir o número de células utilizadas pela solução anterior, Okuda-Song [12] propuseram uma solução sistólica, sem afetar o tempo de processamento, utilizando apenas n^2 células, como ilustrado na figura 2.4.

Figura 2.4: Solução Sistólica de Okuda-Song

Essa solução, diferente das anteriores, armazena os valores de c na própria célula.

O algoritmo sistólico de Okuda-Song realiza a multiplicação de matrizes em $3n$ passos e usa n^2 células.

2.2 Dependência entre computações

Determinadas computações só podem iniciar após o término de outras computações de acordo com a lógica do algoritmo. Sendo assim, é possível determinar uma ordem temporal para que a solução seja corretamente computada.

O método da síntese de algoritmos sistólicos é baseada na transformação dos chamados *vetores de dependência*. O método transforma um algoritmo seqüencial, expresso na forma de três loops aninhados ou na forma de *equações de recorrência uniforme*, em um algoritmo sistólico. A seguir, um algoritmo seqüencial para multiplicação de matrizes com três loops aninhados.

```

for k := 1 to n do
  for i := 1 to n do
    for j := 1 to n do
      c[i,j] := c[i,j] + a[i,k]b[k,j]

```

Podemos notar que a computação no ponto (k, i, j) usa o resultado de computações de outros pontos. Definindo algumas restrições extras, vamos reescrever o algoritmo seqüencial.

1. Toda variável é indexada por todos os índices k, i, j .
2. Toda variável é explicitamente computada e aparece somente uma vez no lado esquerdo (atribuição).
3. Nenhuma variável é utilizada em ambos os lados do operador de atribuição.

Assim,

```

for k := 1 to n do
  for i := 1 to n do
    for j := 1 to n do
      begin
        a[i,j,k] := a[i,j-1,k]
        b[i,j,k] := b[i-1,j,k]
        c[i,j,k] := c[i,j,k-1] + a[i,j,k]b[i,j,k]
      end

```

O algoritmo assume a inicialização, para $1 \leq i, j, k \leq n$:

$$a_{i-1k} = a_{ik}$$

$$b_{-1jk} = b_{kj}$$

2.3 Vetores de Dependência

Na seção anterior, vimos que determinados algoritmos exigem que algumas computações devam ser computadas antes de outras. Por exemplo, uma computação em um ponto (i, j) utiliza o resultado de uma outra computação do ponto $(i - 1, j)$ ou do ponto $(i, j - 1)$, e por isso deve ser computada posteriormente.

Formalizando, dizemos que o ponto (i, j) *depende* dos pontos $(i - 1, j)$ e $(i, j - 1)$. Matematicamente, chamamos de *vetores de dependência* os vetores:

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i - 1 \\ j \end{pmatrix}$$

$$\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} - \begin{pmatrix} i \\ j - 1 \end{pmatrix}$$

Em nosso exemplo de multiplicação de matrizes, os vetores de dependência na base (k, i, j) associados, respectivamente, às variáveis c , a e b estão expressas abaixo:

$$\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

De maneira análoga, os vetores de dependência de um determinado algoritmo podem ser expressos na forma de matriz, a *matriz de dependência* D . Para o algoritmo de multiplicação de matrizes, temos a matriz de dependência:

$$D = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

2.4 Transformações de Dependências

Para simplificar o entendimento desta seção, vamos continuar usando o exemplo da multiplicação de matrizes com os três loops aninhados.

Vamos definir um *grafo de computação* como um grafo onde os nós são os pontos de computação (k, i, j) do algoritmo e as arestas representam a dependência entre

as computações. A figura 2.5 representa o grafo de computação para multiplicação de matrizes.

Figura 2.5: Grafo de computação para multiplicação de matrizes 3 x 3

A partir de um grafo de computação, podemos aplicar uma transformação linear T a fim de obter um *grafo transformado*, onde cada nó (t, x, y) representa o instante de tempo t e a localização (x, y) da célula de processamento em um plano.

Ao aplicar a transformação T , se um nó $v = (k, i, j)$ do grafo de computação é mapeado para o nó $v' = (t, x, y)$ do grafo transformado, então o nó de computação v é computado no instante de tempo t pela célula de computação posicionada nas coordenadas (x, y) do plano.

Formalmente, seja a representação matricial da transformação linear:

$$T = \begin{pmatrix} t_{11} & t_{12} & t_{13} \\ t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{pmatrix}$$

onde t_{ij} são inteiros.

Dessa forma, podemos obter o instante para computar (k, i, j)

$$\begin{pmatrix} t_{11} & t_{12} & t_{13} \end{pmatrix} \begin{pmatrix} k \\ i \\ j \end{pmatrix}$$

e a posição da célula (x, y) para computar (k, i, j)

$$\begin{pmatrix} t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{pmatrix} \begin{pmatrix} k \\ i \\ j \end{pmatrix}$$

Vamos considerar um vetor de dependência $d = p' - p$, onde o ponto $p' = (k', i', j')$ depende de um outro ponto $p = (k, i, j)$ em um grafo de computação. Portanto, temos que:

$$\begin{pmatrix} t_{11} & t_{12} & t_{13} \end{pmatrix} \begin{pmatrix} k' \\ i' \\ j' \end{pmatrix} - \begin{pmatrix} t_{11} & t_{12} & t_{13} \end{pmatrix} \begin{pmatrix} k \\ i \\ j \end{pmatrix} > 0$$

ou seja,

$$\begin{pmatrix} t_{11} & t_{12} & t_{13} \end{pmatrix} d > 0$$

para todos os vetores de dependências d do algoritmo.

Além disso, podemos obter a direção da comunicação entre células de processamento vizinhas da seguinte maneira:

$$\begin{pmatrix} t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{pmatrix} \begin{pmatrix} k' \\ i' \\ j' \end{pmatrix} - \begin{pmatrix} t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{pmatrix} \begin{pmatrix} k \\ i \\ j \end{pmatrix}$$

ou

$$\begin{pmatrix} t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{pmatrix} d = \begin{cases} \pm (1, 0)^T \\ \pm (0, 1)^T \\ \pm (1, 1)^T \end{cases}$$

para todos os vetores de dependência d .

2.4.1 Equações de Recorrência Uniforme

Vejamos duas definições de *equações de recorrência uniforme (ERU)*.

Definição 2.1 *Seja D um domínio poliédrico convexo. Um sistema de equações de recorrência uniformes ERU é um sistema de m equações do tipo*

$$V_i(z) = f_i(V_{i_1}(z - \theta_{i_1}), \dots, V_{i_k}(z - \theta_{i_k})), 0 \leq i < m.$$

onde $z \in D$, $\theta_{i_1}, \dots, \theta_{i_k}$ são vetores de Z^n .

Dizemos que um sistema desses computa as m funções

$$V_0, V_1, \dots, V_{m-1}.$$

Vejamos um caso simplificado desse sistema onde apenas o V_0 interessa.

Definição 2.2 *Um sistema de equações de recorrência uniformes ERU (simplificado) é o seguinte sistema de m equações.*

$$\begin{aligned} V_0(z) &= f(V_0(z - \theta_0), V_1(z - \theta_1), \dots, V_{m-1}(z - \theta_{m-1})), \\ V_1(z) &= V_1(z - \theta_1), \\ &\vdots \\ V_{m-1}(z) &= V_{m-1}(z - \theta_{m-1}), \end{aligned}$$

onde $z \in D$ e $\theta_i, 0 \leq i < m$, são vetores de Z^n .

2.4.2 Grafos de Dependências

Um sistema de equações de recorrência uniforme ERU pode ser usado para especificar um algoritmo. Agora, vejamos a definição de um *grafo de dependência*.

Definição 2.3 *Seja $\Theta = \{\theta_0, \theta_1, \dots, \theta_{m-1}\}$ o conjunto dos vetores θ_i de um sistema ERU. Os θ_i são ditos vetores de dependências do sistema. Sejam os pontos $z, y \in D$,*

dizemos que z é dependente de y , por θ_i , se existe $\theta_i \in \Theta$, tal que

$$z = y + \theta_i$$

As dependências podem ser representadas por um grafo orientado chamado grafo de dependência do sistema ERU, denotado por (D, Θ) . Os seus vértices são os pontos de D . Se o ponto z é dependente de y por θ_i , então existe uma aresta do vértice y ao vértice z .

Se $z \in D$ e $z - \theta_i \notin D$, chamamos de *entrada* do algoritmo os valores $V_i(z - \theta_i)$, para $0 \leq ilm$. Por outro lado, chamamos de *saída* do algoritmo as instâncias $V_i(z)$ que aparecem apenas do lado direito das equações.

2.4.3 Função de Tempo τ

Como vimos anteriormente, um grafo de dependência representa as computações que precisam ser calculadas por um algoritmo respeitando sua relação de dependências. A *função de tempo* τ é uma função de transformação que, aplicado ao grafo de dependências, mapeia cada nó de computação em um inteiro positivo representando o seu instante de execução. Karp, Miller e Winograd [33] caracterizam computações de um sistema ERU.

Para um dado $z \in D$, vamos assumir que as computações $V_i(z)$ são executadas em paralelo, e consomem uma unidade de tempo.

Definição 2.4 Seja $x = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix} \in D$, $\delta \in Z$ e $\lambda = \begin{pmatrix} \lambda_0 \\ \lambda_1 \\ \vdots \\ \lambda_{n-1} \end{pmatrix}$, $\lambda_i \in Z$.

A função de tempo $\tau : Z^n \rightarrow Z$ é a seguinte função linear

$$\begin{aligned} \tau(x) &= \lambda_0 x_0 + \lambda_1 x_1 + \dots + \lambda_{n-1} x_{n-1} + \delta \\ &= \lambda^T x + \delta \end{aligned} \tag{2.1}$$

que satisfaz as seguintes duas condições:

1. τ é não negativo sobre D .
2. Se z depende de y , então $\tau(z) > \tau(y)$.

A primeira condição é dada apenas por uma questão de conveniência, desde que a função τ seja interpretada como tempo. A segunda condição torna possível um escalonamento de computações dependentes.

Teorema 2.5 (Quinton e Robert [13])

Para um dado sistema ERU, com grafo de dependências (D, Θ) e com conjunto V_D de vértices de D , os parâmetros $\lambda_0, \lambda_1, \dots, \lambda_{n-1}$ e δ definem uma função de tempo τ se e somente se

1. $\forall v_i \in V_D, \quad \lambda^T v_i + \delta \geq 0$ e
2. $\forall \theta_i \in \Theta, \quad \lambda^T \theta_i > 0$, ou seja, $\lambda^T \theta_i \geq 1$.

Seja um ponto z dependente de um outro ponto y , dado por θ_i , ou seja, $z = y + \theta_i$. Pelo teorema 2.5, podemos dizer que:

$$\lambda^T \theta_i = \lambda^T (z - y) = \tau(z) - \tau(y).$$

Portanto, $\lambda^T \theta_i$ pode ser interpretado como a defasagem de tempo da computação de z com o tempo de y . O teorema 2.5 requer que essa defasagem seja maior ou igual a 1.

Para obter uma *função de tempo ótima* em um sistema ERU com o grafo de dependência (D, Θ) , podemos minimizar o tempo de defasagem entre as dependências. Então,

$$\text{minimizar } \lambda^T \theta_k,$$

sujeito a

$$\lambda^T \theta_i \geq 1, \forall \theta_i \in \Theta.$$

2.4.4 Função de Alocação α

Definida a função de tempo τ , vamos definir a *função de alocação* $\alpha : Z^n \rightarrow Z^{n-1}$, que para cada ponto $z \in D$ mapeia uma posição $\alpha(z)$ onde realiza a computação associada. O ponto $\alpha(z)$ pode ser interpretado como a posição da célula de processamento em um sistema de computação.

Definição 2.6 A função de alocação $\alpha : Z^n \rightarrow Z^{n-1}$ é uma função na forma

$$\alpha(x) = (\alpha_0(x), \alpha_1(x), \dots, \alpha_{n-2}(x))$$

onde cada α_i , é uma função linear de Z^n em Z , tal que

$$\forall x, y \in D, \alpha(x) = \alpha(y) \Rightarrow \tau(x) \neq \tau(y)$$

Um *hiperplano de tempo* é um hiperplano de pontos $z \in D$ que possuem o mesmo valor de $\tau(z)$, ou seja, os pontos cujas computações são executadas ao mesmo tempo. Hiperplanos de tempo são ortogonais a λ . Na figura 2.6, os hiperplanos de tempo são representados pela linha

Figura 2.6: Hiperplano de tempo

Vejamos a seguir, como obter uma função de alocação α projetando o domínio D segundo uma direção u , que não seja ortogonal a λ .

Teorema 2.7 (Quinton e Robert [13])

Seja u um vetor não nulo de Z^n tal que $\lambda^T u \neq 0$.

Seja $u_j \neq 0$ um componente de u . A seguinte aplicação define uma função de alocação $\alpha(x) = (\alpha_0(x), \alpha_1(x), \dots, \alpha_{n-2}(x))$, onde

$$\begin{aligned}\alpha_k(x) &= u_j x_k - u_k x_j & \text{se } 0 \leq k < j \\ \alpha_k(x) &= u_j x_{k+1} - u_{k+1} x_j & \text{se } j \leq k < n-1.\end{aligned}$$

Por meio da função de alocação α , os pontos de D mapeiam as células de processamento.

Para a escolha de *funções de alocação ótimas*, um critério usualmente considerado é a redução no número de células de processamento necessárias. Para o caso da implementação de sistemas sistólicos em VLSI, esse critério é justificado pela necessidade de uma menor área de silício.

Capítulo 3

Algoritmos Sistólicos para Grades de Computação

Descritos os conceitos de Grade de Computação e de Algoritmos Sistólicos, este capítulo tem como objetivo relatar como é possível viabilizar o uso da técnica de arranjos sistólicos em ambientes distribuídos como os clusters e grades de computação. Este assunto pode ser encontrado em [11, 37], de Hayashida-Song, .

3.1 Sistemas de Computação Paralela e Distribuída

A computação paralela e distribuída pode ser usada para diversas finalidades. Em geral, os algoritmos paralelos são projetados para obter um desempenho superior à versão existente do algoritmo sequencial para o mesmo problema. Em busca deste objetivo, os algoritmos paralelos costumam quebrar o problema em problemas menores para serem processados em máquinas distintas.

A eficiência de algoritmos paralelos depende de uma série de fatores que estão ligados diretamente a arquitetura das máquinas e de sua rede de interconexão. Para que um algoritmo paralelo tenha sucesso em um ambiente como este, é importante estar sempre atento ao balanceamento de carga, procurando minimizar a comunicação e sobrepor computação.

Os arranjos sistólicos implementados em circuitos VLSI pertencem a uma classe de algoritmos paralelos que combinam com quase todas estas características. Sua

estrutura possui um bom balanceamento de carga, e praticamente todas as comunicações e computações são sobrepostas. Porém, este tipo de estrutura possui muitas rodadas de comunicação, um ponto não desejado em algoritmos paralelos.

Deficiente de um modelo aceitável, a área de computação paralela e distribuída teve uma significativa mudança nos anos 90 com a chegada dos modelos realísticos. Estes modelos determinaram um novo conceito de troca de mensagens. As antigas trocas individuais de mensagens foram substituídas por uma série de superpassos de comunicação envolvendo mensagens de granularidade grossa.

Ao adotar o uso de superpassos de comunicação com granularidade grossa (conhecido como modelo realístico) mais algumas adaptações nos arranjos sistólicos, foi possível projetar um algoritmo sistólico paralelo e distribuído com alto desempenho.

3.2 Adaptando para Ambientes Paralelos e Distribuídos

Ao aplicar a técnica em grades de computação, surgem alguns problemas que podem diminuir a performance das aplicações. Os problemas mais importantes podem ser tratados da seguinte forma:

- *granularidade*: alteração da granularidade fina do método de paralelização de laços para granularidade grossa, diminuindo o número de rodadas de comunicação que tem um alto custo na grade.
- *sincronização*: o método prevê uma sincronização explícita entre as rodadas de comunicação e computação, mas com uma comunicação baseada em troca de mensagens podemos obter uma sincronização implícita entre as rodadas.
- *heterogeneidade*: diferente dos chips VLSI, a grade pode possuir células de processamento com potencial diferenciado, além da possibilidade da célula estar executando múltiplas tarefas. Se o problema não for tratado, uma célula de processamento lenta em conjunto com a sincronização pode prejudicar a performance da aplicação.
- *tolerância a falhas*: em uma grade, existe a possibilidade de uma célula de processamento (um computador, por exemplo) falhar. Nesse caso, toda a com-

putação realizada em outras máquinas seriam desperdiçadas. A redundância de computações seria uma forma de tratamento para tentar evitar esse tipo de problema, distribuindo a mesma tarefa em mais de uma célula de processamento.

O trabalho de Hayashida-Song [11, 37] objetivou a criação de aplicações que possam ser utilizadas em uma grade de computação, desenvolvidas a partir do método de paralelização de laços aninhados para problemas não trivialmente paralelizáveis. Algumas aplicações foram desenvolvidas e testadas no Beowolf do DCC/IME, utilizando a biblioteca MPI para a implementação dos algoritmos. Bons resultados foram obtidos para o problema da multiplicação de matrizes, que também está sendo testado no projeto do InteGrade-IME/USP.

Capítulo 4

Tolerância a Falhas

Nesta seção, vamos introduzir o conceito de tolerância a falhas. Boas referências podem ser encontradas em [17, 1, 2, 3, 4, 5, 6, 7, 15, 16, 18, 19, 21, 26, 25, 27, 28, 30, 31, 32, 35, 36, 38, 39, 42, 41, 40].

Em busca de uma solução econômica para sistemas computacionais que suportem grande volume de processamento, muitas instituições adotaram a infra-estrutura de redes de computadores como plataforma para execução de suas aplicações. Em um ambiente distribuído como este, supõem-se que o sistema computacional opere apropriadamente sem interrupção de serviço e sem perda de dados. No mundo ideal, sistemas de computação são cem por cento confiáveis e cem por cento disponíveis. Já no mundo real, confiabilidade e disponibilidade absolutas estão muito longe de serem alcançadas.

O grande número de componentes e a complexidade de especificação de um sistema computacional tornam inevitáveis a presença de falhas. Mas as consequências das falhas, uma interrupção no fornecimento do serviço ou perda de dados, podem ser evitadas pelo uso adequado de técnicas viáveis e de fácil compreensão.

No entanto, o emprego de técnicas que toleram falhas envolve um alto custo. A simples necessidade de backup de dados consome espaço de armazenamento mais o tempo da cópia, sem contribuir para o aumento de desempenho do sistema.

Devido ao custo envolvido, os sistemas tolerantes a falhas eram encontrados com maior frequência apenas em sistemas críticos como controle aéreo, monitoração de

pacientes, orientação e controle de mísseis, entre outros. Com a expansão dos diversos tipos de serviços oferecidos na rede mundial internet, aumentou também a dependência tecnológica da população em relação a qualidade dos serviços. A presença de falhas nesses serviços podem ser catastróficas para a segurança da população ou para a imagem das empresas.

4.1 Defeitos em Sistemas de Computação

A grande presença de sistemas automatizados e informatizados no cotidiano da população, criou uma forte dependência por sistemas de computação mais confiáveis e mais disponíveis. Seja no controle de tráfego aéreo ou usinas de geração de energia, na segurança e manutenção de dados sigilosos sobre pessoas e empresas, entre outros exemplos, os computadores atuam continuamente. Um simples defeito nesses sistemas pode levar a grandes catástrofes.

É notável como os componentes de hardware evoluíram em qualidade e confiabilidade nos últimos tempos. Entretanto, as aplicações e os procedimentos de projetos estão cada vez mais complexos e apresentando cada vez mais problemas. Apenas a confiabilidade dos componentes de hardware não garante a qualidade e segurança desejada aos sistemas de computação.

Alguns casos de defeitos ficaram bem conhecidos, como a falha de projeto na unidade de ponto flutuante do Pentium, falhas em mísseis usados na guerra do Golfo em 1991, colapso no sistema de cartão de crédito da França em 1993, entre outros. Todos estes defeitos foram analisados e suas causas foram determinadas, mas não há garantias de que algo semelhante não volte a se repetir.

4.2 Tolerância a falhas ou dependabilidade?

O termo tolerância a falhas foi cunhado por Avizienis em 1967, sendo interpretado por acadêmicos como comportamento de sistemas computacionais sujeitos a ocorrência de falhas. Na indústria o termo nunca teve boa aceitação, sendo que desenvolvedores de sistemas de controle preferem o termo sistemas redundantes para seus equipamentos. Na comercialização de sistemas computacionais como mainfra-

mes e servidores de rede, o termo usual é alta disponibilidade.

Sistemas redundantes e sistemas de alta disponibilidade usufruem de técnicas comuns para se alcançar resultados diferentes, onde os primeiros buscam alta confiabilidade e os outros continuidade de serviços.

O uso do termo tolerância a falhas para designar a área de estudo sofre várias críticas no mundo inteiro. A maior delas é entender o termo como uma propriedade absoluta. Nesta visão distorcida, um sistema tolerante a falhas teria que tolerar qualquer falha em qualquer situação, o que é uma promessa irrealizável e pode conduzir a falsas expectativas.

Aos poucos o termo dependabilidade vem substituindo a tolerância a falhas no meio acadêmico. Por exemplo, em 2000 o Fault Tolerant Computing Symposium (FTCS) foi rebatizado para Dependable Systems and Networks.

4.3 Conceitos: Falha, erro e defeito

Um *sistema* consiste de vários componentes, onde cada componente pode ser visto como um sub-sistema. No nível mais baixo temos os componentes atômicos que não podem ser divididos ou possuem uma estrutura que podemos abstrair.

É importante que um sistema de computação atenda sua especificação com sucesso. O sistema apresenta um *defeito* quando há um desvio da especificação. *Defeitos* não podem ser tolerados mas deve-se evitar o que o sistema apresente defeito ao usuário. Um defeito pode ocorrer quando o sistema processa a partir de um estado de *erro*. Finalmente define-se falha, ou falta, como a causa física ou algorítmica do erro.

Barry W. Johnson sugeriu o *modelo 3 universos* como simplificação para os conceitos de falha, erro e defeito. Falhas estão associadas ao universo físico, erros ao universo da informação e defeitos ao universo do usuário.

Para deixar claro estes conceitos, vejamos um exemplo. Um chip de memória apresenta uma falha do tipo grudado-em-zero em um de seus bits (falha no universo físico), pode provocar uma interpretação errada da informação armazenada (erro

no universo da informação) e o sistema pode apresentar saldo insuficiente na conta para a compra de um automóvel (defeito no universo do sistema).

Falhas são inevitáveis mas defeitos são evitáveis usando técnicas de tolerância a falhas.

O período de tempo desde a ocorrência da falha até a manifestação do erro é chamado de *latência de falha*. Já a *latência do erro* tem seu início da ocorrência do erro até a manifestação do defeito para o usuário. O tempo total da ocorrência da falha até o surgimento do defeito é a soma da latência de falha com a latência de erro.

As falhas são classificadas geralmente como *falhas físicas* e *falhas humanas*. As principais causas de falhas físicas são imperfeições de manufatura, fadiga dos componentes físicos, além de distúrbios externos como radiação, interferência eletromagnética e variações ambientais (temperatura, pressão, umidade). As falhas humanas podem ser divididas em *falhas de projeto*, como problemas de especificação ou implementação, e *falhas de interação* como problemas na operação do sistema.

Além das causas das falhas, é importante considerar a *natureza* (hardware, software, projeto), a *duração ou persistência* (permanente ou temporária), *extensão* (local no módulo, global) e o *valor* (determinado ou indeterminado no tempo).

Em sistemas críticos, as falhas de software e de projeto são classificadas como os problemas mais graves. Tradicionalmente, sistemas críticos são construídos a suportar falhas físicas e, portanto, as falhas humanas não previstas tem grande potencial de comprometer a confiabilidade e disponibilidade do sistema.

4.4 Dependabilidade

O termo dependabilidade é uma tradução literal do termo inglês *dependability*, que indica a qualidade do serviço de um sistema e a confiança depositada no serviço fornecido.

Dessa forma, pode-se dizer que o objetivo de tolerância a falhas é alcançar dependabilidade. A dependabilidade de um sistema pode ser mensurada com o auxílio

de uma série de medidas numéricas, tais como confiabilidade, disponibilidade, segurança de funcionamento, segurança, manutenibilidade, testabilidade e comprometimento de desempenho (performability).

4.4.1 Confiabilidade (Reliability)

A confiabilidade é a capacidade de um sistema atender a especificação dentro das condições definidas, durante um certo período de funcionamento e condicionado a estar operacional no início do período.

Para determinar se o sistema está operando conforme esperado ou não, a especificação deve ser formal e completa. Por exemplo, a especificação deve conter as condições de funcionamento bem definidas (condições ambientais) e a extensão do período de funcionamento do sistema (tempo de missão). Além disso, não é possível mensurar a confiabilidade se o sistema iniciar sua operação com defeito.

Confiabilidade é a probabilidade do sistema funcionar, dentro das condições definidas, sem ocorrências de defeitos (conforme especificação) durante o período estipulado.

Não podemos confundir confiabilidade com disponibilidade. Um sistema pode ter alta confiabilidade e baixa disponibilidade. Por exemplo, um avião que funcione perfeitamente durante o voo e precisa de manutenção nos intervalos.

4.4.2 Disponibilidade (Availability)

A disponibilidade é probabilidade do sistema estar operacional em um determinado instante de tempo. Esta medida mostra a relação entre o tempo de funcionamento e o tempo de reparo do sistema.

Um sistema pode ser altamente disponível mesmo apresentando períodos de inoperabilidade, quando está sendo reparado, desde que estes períodos sejam curtos e não comprometam a qualidade do serviço.

Disponibilidade é desejável em diversos tipos de sistemas. Sistemas de consultas a base de dados, servidores de rede, servidores de páginas web, são alguns exemplos

onde alta disponibilidade é requerida.

4.4.3 Segurança de Funcionamento (Safety)

Segurança de funcionamento é a medida da capacidade do sistema se comportar livre de falhas. Um sistema com alta segurança de funcionamento deve exercer sua função corretamente ou, caso contrário, deve levar o sistema a um estado seguro quando não conseguir garantir seu funcionamento.

A segurança de funcionamento é desejável para evitar que falhas do sistema provoquem danos a outros sistemas ou pessoas que dele dependam. Por exemplo, um sistema de transporte ferroviário onde os controles do trem providenciam uma desaceleração e parada automática quando não conseguir garantir o funcionamento correto.

4.4.4 Outros atributos

Um sistema é *seguro* (Security) quando possui proteção contra falhas maliciosas. Esta medida visa fornecer privacidade, autenticidade e integridade dos dados de um sistema.

O *comprometimento do desempenho* (Performability) está relacionado à queda de desempenho provocada por uma falha. O sistema continua operando mas o seu desempenho é abaixo do esperado.

A medida de *manutenabilidade* fornece informações sobre o tempo que o sistema leva para ser restaurado após a ocorrência do defeito.

Já a *testabilidade* é a capacidade de realizar certos testes internos ao sistema. Quanto melhor a testabilidade, menor o tempo de manutenção e, conseqüentemente, menor o tempo de sistema indisponível.

4.5 Técnicas para alcançar dependabilidade

Para o desenvolvimento de um sistema com atributos de dependabilidade desejáveis, um conjunto de técnicas pode ser empregada. Todas as técnicas tem em comum o fato de se preocuparem com a ocorrência inevitável de falhas.

A técnica de *prevenção de falhas* impede a ocorrência ou a introdução de falhas no sistema usando metodologias de projeto e tecnologias adequadas para seus componentes.

A *validação* é uma técnica que testa o sistema em busca de falhas para, em seguida, iniciar o processo de remoção das falhas.

A aplicação de técnicas de *tolerância a falhas* são necessárias em sistemas quando a prevenção e remoção de falhas são insuficientes para a confiabilidade e disponibilidade desejada. Baseadas em redundância, estas técnicas exigem componentes adicionais ou algoritmos especiais para garantir o funcionamento correto do sistema mesmo na ocorrência de falhas.

Podemos dividir as técnicas de tolerância a falhas em duas classes disjuntas: *mascaramento* ou *detecção, localização e reconfiguração*.

4.5.1 Detecção, localização e reconfiguração

Diversos autores apresentam suas próprias classificações para técnicas de tolerância a falhas. A mais comum é a de Anderson e Lee composta de quatro fases: detecção, confinamento, recuperação e tratamento.

A primeira fase é a detecção de um erro. Uma falha precisa primeiro se manifestar como um erro, para então ser encontrada por um mecanismo de detecção. Um mecanismo de detecção conhecido é o esquema de duplicação e comparação. Dois componentes idênticos realizam a mesma computação e os resultados são comparados. Havendo diferença, o erro se manifesta.

A fase de confinamento tenta evitar a propagação de dados inválidos entre a ocorrência da falha e a detecção do erro. O confinamento estabelece limites para a propagação do dano conforme previsão do projeto.

A recuperação de erros é a fase que ocorre após a detecção do erro. Nesta fase, é preciso trocar o estado atual incorreto para um estado livre de falhas. A recuperação pode ser feita de duas formas: recuperação por retorno (backward error recovery) ou recuperação por avanço (forward error recovery).

- Recuperação de erro por retorno: o estado do sistema é registrado (checkpoint) periodicamente durante a execução da computação normal para que possa ser restaurado na ocorrência de erros.
- Recuperação de erro por avanço: não registra estado algum. O objetivo é corrigir o estado do sistema aplicando ações corretivas adequadas mas é necessário um bom diagnóstico sobre a natureza do erro.

A recuperação de erros pode ser simples para sistemas com processo único mas é complexa para processamento distribuído. A recuperação nestes sistemas podem provocar um efeito dominó devido a existência de mensagens órfãs deixadas por processos ao se desfazer da computação.

A última fase é o tratamento de falhas que consiste em localizar a origem do erro, localizar a falha de forma precisa, reparar a falha e, em seguida, recuperar o resto do sistema. A localização da falha é determinada através de testes de comparação de resultados gerados com resultados previstos. O reparo é feito através da remoção do componente danificado de forma manual ou automática. O reparo automático pode simplesmente reconfigurar o sistema para trabalhar com um número reduzido de componentes ou substituir por um outro componente disponível.

4.5.2 Mascaramento de falhas

O mascaramento de falhas garante resposta correta mesmo na presença de erros. Ou seja, a falha não se manifesta como erro e o sistema não entra em um estado errôneo. Em caso de falhas permanentes, a localização e o reparo são necessários.

Para se obter tolerância a falhas com o mascaramento, podemos replicar componentes de hardware, criar diversas versões do mesmo programa, usar codificação para detecção e correção de erros, entre outros.

Preferida para sistemas de tempo real críticos por não envolver tempos gastos com detecção, localização e reconfiguração.

4.6 Redundância

A redundância é a principal ferramenta utilizada em tolerância a falhas. A aplicação de redundância para implementar técnicas de tolerância a falhas pode aparecer de várias formas:

- redundância de informação.
- redundância temporal.
- redundância de hardware.
- redundância de software.

Todas as formas de redundância causam algum impacto no sistema, seja no custo, no desempenho, ou na potência consumida. Assim, é preciso ponderar o uso desta ferramenta de tolerância a falhas.

A redundância serve tanto para detecção de falhas como para mascaramento de falhas. Mas o grau de redundância é diferente para cada um dos casos. Para detecção de falhas são usados dois componentes idênticos mais um comparador. A falha é detectada quando o comparador acusa resultados distintos. Entretanto, o uso de redundância desta forma não indica qual dos dois componentes é o defeituoso.

4.6.1 Redundância de informação

Na redundância de informações, informações extras são armazenadas ou transmitidas junto ao dado original. Estas informações servem para detecção de erros ou mascaramento de falhas.

Para detecção de erros, as informações extras são usadas para verificar a correção do dados original. Códigos de paridade e checksums são alguns exemplos de detecção de erros.

O mascaramento de falhas utiliza a informação redundante para armazenar códigos de correção de erro como o ECC (error correction code) que servem tanto para detecção de erros como para correção.

4.6.2 Redundância temporal

A redundância temporal repete a computação no tempo em um mesmo componente. Esta técnica evita custos adicionais com hardware mas aumenta o tempo necessário para realizar a computação.

A obtenção de resultados diferentes indicam uma forte indicação de uma falha transitória. Mas resultados iguais podem estar errados quando a falha é permanente.

4.6.3 Redundância de hardware

A redundância de hardware está baseada em replicação de componentes físicos. Podemos dividir em 3 tipos de redundância de hardware:

- redundância passiva ou estática.
- redundância ativa ou dinâmica.
- redundância híbrida.

Redundância de hardware passiva

Na redundância de hardware passiva temos uma replicação de componentes onde todos executam a mesma computação e o resultado é determinado por uma votação. Este método é eficiente para mascarar falhas.

Uma das técnicas mais conhecidas em tolerância a falhas é o TMR (triple modular redundancy). A técnica TMR consegue mascarar falhas triplicando o componente e determinando o resultado por votação da maioria, ou seja, pelo menos dois votos iguais.

Apesar de simples, o votador é ponto crítico de falhas no esquema TMR. Como o votador está em série com os outros componentes, uma baixa confiabilidade neste componente torna o esquema todo de baixa confiabilidade.

Podemos contornar este problema usando um votador de alta confiabilidade, ou triplicar o votador ou ainda realizar a votação por software.

Redundância de hardware ativa

A redundância de hardware ativa utiliza técnicas de detecção, localização e recuperação para obter tolerância a falhas.

Esta técnica é usada em aplicações que suportam ficar em estado errôneo durante um curto período de tempo. As vezes, é preferível defeitos temporários do que suportar o custo de grande quantidade de redundância.

4.6.4 Redundância de software

Em redundância de software, a replicação de componentes idênticos não é uma estratégia válida. Executar os componentes idênticos de software em série ou em paralelo devem apresentar erros de forma idêntica para os mesmos dados de entrada.

Existem outras formas para aplicar redundância em software para mascaramento e detecção de falhas, sem envolver cópias idênticas de software. Entre elas podemos citar: diversidade (ou programação n-versões) e blocos de recuperação.

É importante ressaltar que se o software for projetado corretamente desde o início, então não existe a necessidade de usar técnicas de tolerância a falhas para software.

A técnica de diversidade é uma técnica de redundância usada para obter tolerância a falhas de um software. A partir de uma especificação, são implementadas diversas soluções alternativas e a resposta do sistema determinada por votação.

Uma outra forma de redundância de software é a técnica de blocos de recuperação. Semelhante a programação de n-versões, esta técnica utiliza programas secundários apenas quando o programa primário não passa no teste de aceitação.

Desta forma, os próximos programas executam a computação de forma sequencial até que um deles passe no teste de aceitação.

Capítulo 5

Redundância em Algoritmos Sistólicos para Grades de Computação

Os resultados experimentais de [11, 37] mostram que a adaptação dos Algoritmos Sistólicos para Grades de Computação obteve grande sucesso devido ao bom rendimento de desempenho no ambiente testado.

Entretanto, apesar de ser um ambiente distribuído, o conjunto de máquinas do ambiente testado era homogêneo e os testes foram executados em um momento onde as máquinas estavam praticamente dedicadas ao processamento do problema. A existência de um elemento de processamento com uma capacidade menor de computação dedicada influi significativamente no desempenho global da solução.

A existência de concorrência de outros processos que estão fora do escopo da solução podem tornar um ambiente homogêneo sem controle de balanceamento de carga em um ambiente heterogêneo onde uns elementos possuem maior capacidade de processamento que outros.

Basicamente, o desempenho de um conjunto de máquinas heterogêneas é similar ao desempenho de um conjunto de máquinas homogêneas semelhantes à máquina de menor capacidade de computação do conjunto anterior. Isso ocorre devido às dependências de computação existentes no modelo, criando situações onde o elemento de menor capacidade atrasa o envio de dados para seu vizinho posterior cujo estado é ocioso, à espera de dados para computação.

Em geral, os aglomerados de computadores possuem dezenas, centenas ou até

milhares de máquinas interconectadas por uma rede de alta velocidade. A existência dessa abundância de recursos torna possível a aplicação de uma técnica muito conhecida na área de Tolerância a Falhas: a redundância.

A redundância é bastante usada em Tolerância a Falhas para aumentar a confiabilidade dos sistemas, porém, esse ganho impacta de alguma forma no sistema. Geralmente, afeta o desempenho global do sistema devido à necessidade de tarefas adicionais para o gerenciamento da redundância.

O objetivo desse capítulo é mostrar como a técnica de redundância pode ser aplicada em Algoritmos Sistólicos para grades de computação ou aglomerados com elementos de processamento heterogêneos a fim de evitar que um elemento ruim prejudique o desempenho global. É importante ressaltar que este ganho será mais visível em grades de computação ou aglomerados com poucos elementos de baixa capacidade/disponibilidade de processamento.

5.1 Adaptando a Estrutura

A técnica de redundância de computação se torna interessante quando os recursos computacionais são encontrados em grande quantidade, como nos casos de aglomerados de computadores ou grades de computação. Em geral, estas arquiteturas permitem que os elementos de processamento comuniquem entre si através de canais de comunicação de alta velocidade. Antes de criar a estrutura redundante, vamos definir alguns termos que serão usados na leitura do texto deste ponto em diante.

Um *grupo de redundância* é o conjunto de elementos de processamento redundantes que executam a mesma computação com os mesmos dados de entrada e, consequentemente, que produziram o mesmo resultado. O número de elementos de cada *grupo de redundância* não precisam ter necessariamente a mesma quantidade, porém, nesse texto, a redundância aplicada será sempre uniforme. Desse modo, todos os grupos de redundância terão sempre a mesma quantidade de elementos.

Entende-se por *grau de redundância* a quantidade de elementos de processamento dentro de cada *grupo de redundância*. Por exemplo, se a estrutura fosse duplicada para se obter redundância então o *grau de redundância* seria igual a dois.

Para cada *grupo de redundância* de *grau de redundância* r , vamos identificar cada elemento de processamento do grupo com o valor h , onde $0 \leq h < r$, sem permitir repetições. Dessa modo, podemos definir como *camada de redundância* h o conjunto de elementos de processamento cuja identificação seja igual a h .

A nova estrutura com suporte a redundância será composta por cópias da estrutura usada pelo Algoritmo Sistólico sem suporte a redundância visto anteriormente adicionando, se necessário, canais de comunicação entre as *camadas de redundância*, como visto na Figura 5.1.

Figura 5.1: Uma estrutura para Algoritmos Sistólicos com 9 elementos e suporte de redundância com grau de redundância igual a 2.

5.2 Um algoritmo redundante simples

A redundância pode ser aplicada de diversas maneiras em algoritmos sistólicos. A existência de redundância de componentes físicos (processadores, memória, canal de comunicação) torna possível o uso de computação redundante onde os mesmos dados de entrada são computados pelo mesmo algoritmo em máquinas distintas e, possivelmente, ao mesmo tempo.

Uma primeira solução seria duplicar a estrutura da solução original sem redundância e iniciar o processamento em cada conjunto de forma independente. Desse modo, o primeiro conjunto de máquinas, camada de redundância 0, trabalha para retornar a resposta da solução e o segundo conjunto de máquinas, camada de redundância 1, trabalha paralelamente em busca do mesmo resultado. O tempo de resposta desse algoritmo redundante é dado pelo conjunto de máquinas que tiver melhor desempenho conjunto e obter a resposta primeiro.

Figura 5.2: Estruturas independentes para redundância de computação.

Nessa solução, praticamente não há acréscimo de *overhead* quando comparada à solução original pois os dois conjuntos de máquinas trabalham de forma totalmente independente, sem qualquer tipo de comunicação entre si. Esse exemplo tem *grau de redundância 2* e, devido a independência entre as camadas de redundância, é possível incrementar esse grau conforme a necessidade e a disponibilidade, sem prejudicar o desempenho global.

Em uma análise rápida desse algoritmo redundante, o tempo de resposta de cada uma das camadas de redundância serão similares se o potencial disponível para processamento de todos os elementos envolvidos for homogêneo. Em casos como esse, quando todos os elementos disponíveis são homogêneos, o aumento do *grau de redundância* não implica em um aumento de desempenho.

Por outro lado, se cada uma das camadas de redundância tiver ao menos um elemento ruim, o tempo de resposta do algoritmo global será igualmente ruim. Isso ocorre devido à existência de uma forte dependência de computação entre os elementos, uma característica de algoritmos sistólicos.

Para facilitar a leitura desse ponto em diante, o termo *elemento ruim* irá representar o elemento de processamento que se destaca negativamente em termos de potencial de processamento, seja por falta de capacidade ou disponibilidade. Conseqüentemente, o termo *elemento bom* irá representar aqueles que se destacam positivamente. E para simplificar o estudo do problema, todo elemento de processamento é *elemento ruim* ou *elemento bom*.

Para mensurar o sucesso ou fracasso de um algoritmos sistólicos com redundância, vamos definir a medida *eficiência de redundância*. Dada uma estrutura para algoritmos sistólicos reduntantes com *grau de redundância* r e p elementos de processamento por camada de redundância, a medida *eficiência de redundância* é a probabilidade do algoritmo sistólico ser capaz de executar a computação completa sem que um elemento ruim prejudique no desempenho, onde x elementos dos $r \times n$ são elementos ruins.

5.3 Um algoritmo redundante mais sofisticado

Na solução anterior, a técnica de redundância teve como alvo a computação completa do algoritmo. Tendo o conhecimento da existência de *elementos ruins* na estrutura, essa solução possui um risco consideravelmente alto de apresentar um baixo desempenho, pois, bastaria apenas um elemento ruim em cada camada de redundância para degradar o desempenho global. Nesta seção apresentaremos uma solução que visa diminuir esse risco.

A idéia da nova solução consiste em dividir o escopo da computação redundante em partes menores, resultando em diversas pequenas computações redundantes. Desta forma, podemos ter mais de um elemento ruim na mesma camada de redundância sem apresentar um baixo desempenho. Em nossa estrutura sistólica, o escopo da computação redundante será definido pelo escopo da computação de cada elemento, ou seja, a computação de cada elemento da estrutura original

será replicada em um ou mais elementos redundantes.

Figura 5.3: Passo de comunicação independente nas camadas de redundância e passo de processamento com esforço conjunto do grupo de redundância.

Assim, cada elemento de processamento da estrutura original (sem redundância) torna-se em um *grupo de processamento redundante*, onde todos os elementos do grupo executam paralelamente a mesma computação. Dessa forma, o tempo de resposta do *grupo de processamento redundante* para uma computação qualquer é dada pelo elemento de processamento que fornecer a resposta antes dos demais.

Implementando a redundância dessa forma, um *grupo de processamento redundante* apresentará um desempenho baixo se o grupo for composto integralmente por *elementos ruins*. Devido à forte dependência entre as computações, característica de algoritmos sistólicos, a existência de um *grupo de processamento redundante* com desempenho baixo irá prejudicar o desempenho global da solução.

Diferente da solução anterior, é possível incrementar o *grau de redundância* para diminuir o risco da estrutura possuir um *grupo de processamento redundante* com essa indesejável composição. Entretanto, a prática desse incremento gera um *overhead* proporcional ao *grau de redundância*, o que não acontecia na outra solução.

Esse *overhead* existe, pois é necessário adicionar um gerenciamento dos elementos redundantes para controle e sincronização dos resultados do *grupo de processamento redundante*. Quanto maior for o *grau de redundância*, maior será o número de elementos a serem tratados e, conseqüentemente, maior será o tempo gasto com o tratamento.

5.4 Aplicando a Redundância

Todo elemento de processamento que compõe a estrutura do algoritmo sistólico possui, basicamente, três fases: *recebimento de dados*, *execução da computação* e *envio de dados*.

Na fase de *recebimento de dados*, o elemento de processamento aguarda, ociosamente, os dados necessários para a computação. O número de elementos remetentes varia de acordo com a solução adotada, podendo ser um ou mais. A fase de *recebimento de dados* só termina após o elemento completar o recebimento de dados de todos seus remetentes.

Após receber todos os dados, a fase de *execução da computação* tem seu início. Como esta fase é totalmente independente dos outros elementos, geralmente usa-se o algoritmo seqüencial mais eficiente para a execução da computação. Terminada a computação, o elemento armazena o resultado localmente e passa para a terceira fase.

Na fase de *envio de dados*, a última do ciclo, o elemento não precisa mais dos dados usados e os envia para os elementos vizinhos destinatários. Como na primeira fase, o número de elementos destinatários podem variar de acordo com a solução. Após o envio para todos os remetentes, o elemento termina a terceira fase e inicia um novo ciclo, na fase de *recebimento de dados*.

No algoritmo sistólico reduntante, as fases de *recebimento de dados* e de *envio de dados* permanecem inalteradas em suas respectivas *camadas de redundância*. Ou seja, no início da execução da solução todos os dados de entrada são replicados em cada *camada de redundância* e, a partir desse momento, a camada torna-se responsável por distribuir os dados entre os elementos de sua composição de forma

semelhante ao algoritmo sistólico original.

Já para a fase de *execução da computação*, será preciso montar uma estrutura para o gerenciamento da execução de computações redundantes. Basicamente, a idéia é realizar uma disputa entre os elementos do *grupo de redundância* onde apenas o resultado de um único elemento, de preferência o mais veloz, será considerado. Determinado o elemento, ainda é necessário atualizar e distribuir o resultado para os demais elementos do *grupo de processamento* antes de terminar esta fase.

Este gerenciamento não era necessário na versão do algoritmo sistólico sem redundância. Portanto, será preciso implementar um bom algoritmo para este gerenciamento a fim de minimizar o tempo extra gasto com estas tarefas adicionais.

5.5 Implementação do Algoritmo

Para facilitar a leitura deste capítulo, vamos mostrar como implementar a redundância em algoritmos sistólicos usando como exemplo o problema da multiplicação de duas matrizes. $n \times n$: $C = A \times B$.

Exemplo 5.1 *Dadas duas matrizes $A_{n \times n}$ e $B_{n \times n}$, calcular a matriz $C_{n \times n}$ resultado da multiplicação de matrizes $C_{n \times n} = A_{n \times n} \times B_{n \times n}$.*

As computações existentes na multiplicação de duas matrizes são distribuídas entre os elementos de processamento com o auxílio do método de transformação de dependências, visto no capítulo 2.

A implementação do algoritmo sistólico para esse problema será baseada em uma solução sistólica proposta na literatura por Okuda e Song [12]. A princípio, esta solução foi adotada na implementação de circuitos VLSI, entretanto pode ser perfeitamente estendida para ambientes paralelos e distribuídos.

As vantagens do emprego deste modelo de solução sistólica são: o número reduzido de elementos de processamento necessários, a disposição regular de seus elementos e a comunicação restrita apenas entre vizinhos.

No caso de ambientes distribuídos, como clusters e grades de computação, a estrutura de rede correspondente é conhecida como topologia de grade retangular.

Uma vez definido o uso da topologia em grade retangular de tamanho $p \times p$, as computações necessárias para o problema de duas matrizes serão determinadas com o auxílio do método de transformação de dependências e distribuídas entre os p^2 elementos de processamento compostos pela grade.

Seguindo as propriedades da solução sistólica de Okuda e Song, a matriz resultado $C_{n \times n}$ será dividida em p^2 submatrizes $C_{n/p \times n/p}$. Cada submatriz C_{ij} , $0 \leq i, j < p$, será mantida por um elemento de processamento da grade.

De modo análogo, as matrizes $A_{n \times n}$ e $B_{n \times n}$ também são divididas em p^2 submatrizes A_{ij} e B_{ij} de tamanho $n/p \times n/p$, $0 \leq i, j < p$, que servirão como dados de entrada para as computações dos elementos de processamento. Essa situação pode ser vista na Figura 5.4.

Figura 5.4: Solução sistólica para a multiplicação de duas matrizes A e B em seu segundo passo de computação.

Vale lembrar que os algoritmos sistólicos trabalham de modo síncrono, consequência de passos intercalados de comunicação e computação. Na multiplicação de duas matrizes, o elemento de processamento responsável pela submatriz C_{ij} recebe, durante o passo de comunicação, duas submatrizes A_{ik} e B_{kj} , onde $0 \leq i, j, k < p$.

Já no passo de computação, o elemento de processamento acumula o resultado da multiplicação entre as submatrizes A_{ik} e B_{kj} e armazena localmente na submatriz resultado C_{ij} . Terminada a computação, o elemento de processamento envia as submatrizes A_{ik} e B_{kj} para os elementos responsáveis por $C_{i(k+1)}$ e $C_{(k+1)j}$, respectivamente, iniciando um novo passo de comunicação.

Na solução redundante deste problema, a topologia de grade $p \times p$ passará a ter uma nova dimensão assemelhando-se a um cubo tridimensional. Para uma solução com *grau de redundância* r , a estrutura completa terá $p \times p \times r$ elementos de processamento e cada um será responsável por uma submatriz resultado C_{ij}^h , onde $0 \leq i, j < p$ e $0 \leq h < r$.

O algoritmo sistólico redundante manterá o sincronismo entre os passos de comunicação e computação. O passo de comunicação, que executa a troca de submatrizes A_{ij} e B_{ij} entre os elementos, não terá alterações significativas. Os dados das matrizes A e B serão replicados em cada uma das r camadas de redundância, e serão transportados dentro da camada de redundância de forma independente. Já o passo de computação sofrerá algumas alterações para que os elementos redundantes possam compartilhar informações em prol de um melhor desempenho em conjunto.

Figura 5.5: Replicação das matrizes A e B para serem utilizadas independentemente pelas camadas de redundância.

O novo passo de computação redundante cria uma situação de corrida entre os elementos de processamento do grupo de redundância. Esta corrida determinará apenas um elemento vencedor, preferencialmente aquele que concluir a computação

em menor tempo. Mesmo com esta alteração, a tarefa de acumular o resultado parcial da submatriz C_{ij} é mantida, porém, replicada em cada elemento do grupo de redundância nas submatrizes C_{ij}^h , para $0 \leq h < r$.

A implementação desta corrida consiste em criar um processo local, em cada elemento de processamento, cuja tarefa é executar a computação da multiplicação das submatrizes de A e B . Enquanto isto, o outro processo coopera com os demais elementos do grupo de redundância trocando informações a fim determinar um único elemento vencedor.

Esta divisão em processos é importante para evitar uma possível e indesejável sincronização de computação entre os elementos do grupo de redundância. A comunicação local existente entre os processos é realizada através do modelo de memória compartilhada com o devido tratamento de exclusão mútua, ou seja, apenas um processo por vez poderá ler e/ou atualizar os dados compartilhados.

O processo responsável pela computação possui um mecanismo que permite uma parada forçada antes do término de sua execução. Ao receber a informação de que já existe um elemento de processamento vencedor, o processo de controle envia o comando para o processo de computação cancelar sua execução imediatamente. Caso termine a execução, o processo da computação atualiza o dado compartilhado responsável por fornecer o aviso de computação finalizada ao processo de controle.

Além da concorrência local entre os processos de computação e controle, existe uma concorrência distribuída entre os elementos de processamento dentro de cada grupo de redundância. A tarefa do processo de controle é representar o elemento de processamento, informando aos demais elementos se finalizou a computação ou não. Devido ao processamento paralelo existente, um ou mais elementos de processamento poderão finalizar a execução em um período de tempo relativamente pequeno. Para determinar somente um elemento de processamento vencedor dentro do grupo de redundância, esta solução irá utilizar o algoritmo token ring.

Em sistemas distribuídos, o *algoritmo token ring* pode ser usado para obtenção de exclusão mútua, como o problema em questão. A estrutura desse algoritmo requer a topologia em anel que é composta por participantes candidatos a entrar em regiões crítica. O algoritmo atribui uma posição para cada participante, criando uma ordenação de modo que cada participante saiba exatamente quem é o próximo na

sequência. No início do algoritmo, o participante 0 recebe o *token*. Este *token* circula pelo anel, passando do participante a_i para o participante $a_{[(i+1) \bmod r]}$ em mensagens ponto-a-ponto. Quando o participante recebe o token de seu vizinho, verifica se ele está tentando entrar em uma região crítica. Se tiver, o participante entrará na região crítica, executará todo o trabalho necessário, deixará a região e passará o token para o participante seguinte ao longo do anel. Caso o participante que estiver de posse do token e não precisar entrar em uma região crítica, ele simplesmente passará adiante o token. Quando nenhum participante precisa executar qualquer região crítica, o token circula ao longo do anel em alta-velocidade.

Figura 5.6: Topologia em anel usada pelo algoritmo Token Ring para obtenção de exclusão mútua em sistemas distribuídos.

Em nosso algoritmo sistólico redundante, os participantes candidatos são os elementos de processamento do grupo de redundância e a posição atribuída para cada participante é o índice da camada de redundância do elemento de processamento. Além de ser usado para entrar na região crítica, o token carrega o índice do participante vencedor ou, caso ainda não tenha um vencedor, o valor -1. No início do algoritmo, o participante a_0 recebe o token com o valor -1.

Ao receber o token, o participante poderá tomar 3 atitudes distintas:

1. Se o valor do token for igual a -1. Ainda não existe um vencedor. O participante faz uma consulta para verificar se o processo de computação terminou sua execução. Se o processo terminou a computação então o valor do token passa a ter o índice do participante e o token é passado para o próximo participante. Caso não tenha terminado, o participante apenas repassa o token

- adiante.
- Se o valor do token for diferente de -1 e diferente do índice do participante. Um outro participante venceu a corrida então o participante perdedor cancela o processo de computação. Em seguida, repassa o token e aguarda o envio da submatriz C_{ij} do participante vencedor.
 - Se o valor do token for igual ao índice do participante vencedor. O participante terminou a execução da computação na rodada anterior e, após uma volta completa do token, todos os outros participantes sabem que ele foi o vencedor. Nesse momento, o participante vencedor compartilha o resultado C_{ij} com os demais participantes em uma troca de mensagens do tipo *Broadcast*.

Figura 5.7: Exemplo de uma Corrida.

A Figura 5.7 mostra um exemplo, passo a passo, de uma corrida disputada por elementos de um grupo de redundância. Neste exemplo, o grupo de redundância é composto por dois elementos identificados por 0 e 1. Em (a), os dois elementos de processamento estão recebendo as submatrizes A e B durante o passo de comunicação. Neste momento, o processo de computação ainda não existe e a submatriz C aguarda por uma atualização dos resultados da rodada. Após receber os dados de entrada, o algoritmo entra no passo de computação em (b), onde cada elemento cria um processo de computação local e inicia o processamento da multiplicação. Apenas o elemento 0 cria o token, inicializado com valor -1, e começa a circular na estrutura de anel formada pelos elementos do grupo. Nos passos (c) e (d), o elemento recebe o token com valor -1 e realiza uma consulta para verificar se o processamento local foi terminada. Em caso negativo, apenas repassa o token para o próximo elemento do anel. Em um dado momento (e), o elemento 1 termina a computação e atualiza a matriz C , que é compartilhada entre os processos locais. Em (f), o elemento 1 recebe o token ainda sem vencedor e verifica que o processamento terminou. O token é atualizado com a identificação do vencedor em (g) e enviado ao próximo elemento do anel. Ao receber o token com um vencedor em (h), o elemento 0 cancela a execução do processo de computação e repassa o token adiante. Com todos os elementos devidamente informados, e processamento cancelado, o elemento 1 executa a troca de mensagem *Broadcast* em (i), distribuindo a submatriz C atualizada aos demais integrantes do grupo de redundância. Finalmente em (j), todos os elementos do grupo de redundância estão com a submatriz C atualizadas e podem distribuir os dados de entrada A e B para os próximos elementos da grade.

Ao final de inúmeras rodadas do token, todos os elementos de processamento do grupo de redundância estão com a submatriz resultado parcial C_{ij} devidamente calculada. Com os elementos atualizados, o algoritmo sistólico redundante passa para o próximo passo de comunicação das submatrizes de entrada A_{ij} e B_{ij} .

É importante ressaltar que o tempo gasto no passo de computação é a soma de tempo de execução do elemento vencedor, em geral o mais rápido dos elementos, mais um tempo extra de *overhead* gastos pelo algoritmo token ring e pela atualização do resultado parcial da submatriz C_{ij} .

Capítulo 6

Resultados Experimentais

O campo de algoritmos paralelos atravessou uma fase de incertezas e resultados inesperados na década de 80. Muitos problemas haviam sido profundamente estudados, inclusive com demonstrações teóricas de desempenho. Porém, quando estes problemas eram implementados para as máquinas existentes da época, os ganhos desejados pelo uso do paralelismo eram abaixo do esperado em muitas ocasiões. Por este, e muitos outros motivos, é de extrema importância observar o comportamento dos modelos teóricos de computação paralela quando estão sendo implementados na prática. Tomando este cuidado, o atual capítulo mostra o caminho escolhido para implementar o trabalho na prática, fornecendo informações sobre ferramentas utilizadas, ambiente de testes e resultados obtidos.

6.1 Ferramentas de Trabalho

Este trabalho foi desenvolvido no laboratório de computadores da Rede IME-USP. As implementações dos experimentos foram desenvolvidas na linguagem de programação ANSI C, compiladas pela versão **GNU gcc 2.95.2-13**.

Por se tratar de um experimento paralelo e distribuído, usamos o pacote adicional **POSIX Threads** para o uso de threads locais e o pacote **LAM-MPI** para a troca de mensagens distribuídas.

POSIX Threads é uma especificação de uma API padrão para o desenvolvimento de aplicativos usando threads, que está descrita no IEEE POSIX 1003.1c standard,

popularmente conhecido como Pthreads. O POSIX (Portable Operating System Interface) threads permite a criação de sub-tarefas, cuja execução pode ser escalonada ou rodadas em paralelo.

O pacote LAM-MPI é a implementação da interface MPI (Message Passing Interface), uma importante padronização de sistemas de troca de mensagens amplamente conhecida. Este padrão fornece uma biblioteca prática, permitindo um rápido desenvolvimento de baixo custo para aplicações paralelas de grande escala.

6.2 Ambiente de Testes - Biowulf/IME

Os experimentos foram testados em um cluster Beowulf, um sistema de processamento paralelo de 16 nós instalado no IME. Conhecido por Biowulf/IME, o cluster é composto por 16 nós Athlon Thunderbird conectados por um Switch 3COM 3300 Fast Ethernet 100Mbit/s. Cada nó tem basicamente a seguinte configuração:

- Sistema Operacional Debian Linux 2.2.19.
- Processador 1.2GHz Athlon Thunderbird com 256 KB L2 cache;
- Memória de 768 MB PC133 SDRAM;
- Disco Rígido de 30.73 GB ATA100 7200rpm HD Deskstar 756 GXP DLT-307030 IBM;

O Biowulf pode ser acessado pela rede IME através de um ssh via *tiramisu.ime.usp.br*. Os demais nós do cluster podem ser acessados por *node**, onde * pode ser de 1 a 16. Infelizmente, nem todos os nós do cluster estavam disponíveis para serem usados.

6.3 Multiplicação de Matrizes

Nesta seção, apresentamos um estudo dos experimentos realizados com algoritmos sequenciais, algoritmos sistólicos e algoritmos sistólicos com redundância. O problema escolhido para a implementação destes três algoritmos foi a multiplicação de

duas matrizes, por sua simplicidade na implementação, ajudando o leitor a focar na solução da redundância.

Os algoritmos foram levemente adaptados para executar uma sequência de 50 operações de multiplicação de matrizes $n \times n$. Conhecida por *pipeline*, esta operação tem como objetivo minimizar o tempo de ociosidade dos elementos durante a execução dos algoritmos sistólicos. No algoritmo de multiplicação de matrizes, este problema ocorre apenas no início da primeira operação e ao final da última operação. Portanto, um aumento significativo no número de operações tornará o tempo de ociosidade desprezível.

Devido ao baixo número de elementos no cluster, não foi possível fazer um estudo de *speedup* para os algoritmos sistólicos com redundância que requer no mínimo 8 elementos de processamento. Mais adiante, será comentado sobre as expectativas de *speedups* para os algoritmos sistólicos com redundância.

O foco deste experimento é observar o comportamento dos algoritmos sistólicos em ambientes distribuídos heterogêneos. Em testes individuais sobre os nós do cluster, verificamos que os elementos de processamento são semi-homogêneos, variando aleatoriamente durante o dia. A diferença de tempo constatado entre o elemento mais rápido e o elemento mais lento foi de 25%. Portanto, para obter a heterogeneidade desejada no cluster, forçamos uma dupla computação local da mesma operação para simular o que consideramos *elemento lento* ou *elemento ruim*.

O algoritmo sequencial foi implementado para utilizar apenas o elemento de *rank*0. Já o algoritmo sistólico utiliza 4 elementos com *rank* de 0 a 3, formando uma grade 2×2 . Por último, o algoritmo sistólico redundante usará todos os 8 elementos com *rank* de 0 a 7, formando uma estrutura cúbica $2 \times 2 \times 2$.

Os tempos dos experimentos foram medidos, em segundos, com o auxílio do comando *MPI_Wtime()* da biblioteca do MPI. Este método foi usado para medir o tempo dos três tipos de algoritmos (sequencial, sistólico e sistólico com redundância) implementados. Todos os algoritmos executam o processo de inicialização *MPI_Init()* e finalização *MPI_Finalize()* do MPI, porém, estes tempos não serão considerados.

6.3.1 Tempos em Ambientes Homogêneos

Vamos mostrar o resultado da primeira bateria de testes. Aqui, os algoritmos estão sendo testados sem a intervenção de *elementos lentos*. Implementamos uma versão para cada algoritmo e executamos o experimento com 50 operações de multiplicação de matrizes em *pipeline*. Realizamos testes com matrizes quadradas $n \times n$, começando por matrizes de 180×180 até matrizes 420×420 .

O resultado do experimento observado na Figura 6.1 mostra uma curva de um crescimento polinomial para todos os algoritmos, consequência do crescimento polinomial no número de operações por multiplicação de matrizes.

O mau desempenho do algoritmo sequencial é facilmente explicado pela maior quantidade de computação realizada sequencialmente em um único elemento. Enquanto isso, o algoritmo sistólico mostra um bom desempenho ao computar paralelamente com seus 4 elementos. No meio termo, temos o algoritmo sistólico redundante com marcas um pouco acima da curva de tempo de seu predecessor. Mesmo com a tarefa de executar a mesma quantidade de operações, os algoritmos sistólicos redundantes tiveram desempenho ligeiramente inferior devido ao overhead gerado pelo controle de computações redundantes dentro de cada grupo de redundância.

6.3.2 Tempos em Ambientes Heterogêneos

Em nossa segunda bateria de testes, vamos mostrar o comportamento dos algoritmos quando um de seus elementos possuem um desempenho muito inferior que os demais. Foram realizados cinco experimentos diferentes: (a) Sequencial 1 Lento, (b) Sistólico 1 Lento, (c) Redundante 1 Lento, (d) Redundante 2 Lento, e (e) Redundante Grupo Lento.

Os algoritmos (a), (b) e (c) tiveram um de seus elementos comprometidos por uma dupla computação forçada. Ou seja, estes elementos rendem no máximo metade da sua capacidade normal. O algoritmo (d) teve dois elementos comprometidos, porém, os elementos pertencem a grupos de redundância distintos. Em (e), ocorre justamente o caso em que dois *elementos lentos* pertencem ao mesmo grupo de redundância.

Figura 6.1: Tempos de processamento para multiplicação de matrizes em ambientes homogêneos

Descartando o algoritmo sequencial (a) que apresenta disparado o pior desempenho na Figura 6.2, é possível observar em (c) e (d) a redundância garantindo um bom desempenho do algoritmo mesmo com *elementos lentos* em sua composição. Isso já não ocorre quando apenas um elemento do algoritmo sistólico (b) está comprometido, ou quando, o algoritmo redundante (e) possui um grupo de redundância composto integralmente por elementos ruins.

A Figura 6.3 mostra os resultados do mesmo experimento por uma visão diferente. Por este gráfico, podemos observar que o tempo gasto pelo algoritmo sistólico (b) chega a ser 50% maior quando comparado aos algoritmos redundantes (c) ou (d).

6.4 Comparativo Sistólico Com e Sem Redundância

Este último experimento comparativo mostra claramente os benefícios que a redundância pode trazer. Neste caso, apenas um único elemento de cada algoritmo

Figura 6.2: Tempos de processamento para multiplicação de matrizes em ambientes heterogêneos

será selecionado para ter a sua capacidade de processamento testada, reduzindo em $1\times$, $2\times$, $4\times$, $8\times$ e $16\times$. O algoritmo sequencial não é mostrado por ser óbvio o desempenho ruim.

Começando por $1\times$, observamos que o algoritmo sistólico executa a multiplicação de matrizes em um tempo inferior ao do algoritmo redundante. Ao reduzir a capacidade dos elementos selecionados em $2\times$, podemos observar que o tempo do algoritmo sistólico ultrapassa a linha do algoritmo redundante e continua subindo proporcionalmente ao fator de redução da capacidade de processamento.

O fato mais interessante a ser observado na Figura 6.4 é a eficácia do emprego da redundância em algoritmos sistólicos. No experimento, a variação de capacidade do elemento selecionado não interferiu no tempo de execução total do algoritmo sistólico redundante.

Figura 6.3: Comparativo entre algoritmos em ambientes heterogêneos

Figura 6.4: Algoritmos sistólicos e algoritmos sistólicos com redundância ao sensibilizar a capacidade de processamento de um único elemento de cada algoritmo.

Capítulo 7

Conclusão

O crescimento constante da participação de computadores pessoais em nosso cotidiano nas últimas décadas, desencadeou um grande interesse por este tipo de equipamento. Seja em casa ou no trabalho, os computadores pessoais estão presentes e, em muitos deles, a capacidade computacional é similar ou superior ao de muitas super-máquinas que custavam milhões de dólares há pouco tempo atrás.

Em função desta tendência, houve uma melhora significativa na qualidade dos equipamentos, além destes tornarem-se mais acessíveis ao público em geral. Este ganho de qualidade não é exclusivo de equipamentos físicos, mas também em softwares e prestação de serviços.

A comunidade de computação paralela e distribuída acompanhou este avanço tecnológico, e observou que muitas instituições possuem uma boa estrutura de equipamentos que são sub-utilizadas por seus usuários. Ao invés de gastar milhões de dólares em compras de equipamentos de últimas geração, profissionais e pesquisadores estão aproveitando a ociosidade de aglomerados de computadores interconectados para construir super-máquinas de baixo custo.

De nada adianta uma super-máquina distribuída se não existirem softwares capazes de gerenciar e administrar eficientemente os recursos disponíveis neste tipo de arquitetura. No capítulo 1, Grades de Computação, foi mostrado como os profissionais e pesquisadores estão trabalhando no sentido de explorar ao máximo esse tipo de estrutura.

Também foi visto no capítulo 2 um tipo de computação paralela muito usada em circuitos VLSI. Podemos tirar idéias interessantes desse tipo de estrutura, como o bom balanceamento de carga, a alta taxa de sobreposição de comunicação e computação, o baixo número de elementos vizinhos, e a distribuição dos dados de entrada em sístoles. O mesmo método de transformação de dependências usado nos circuitos, pode ser usado na geração de algoritmos paralelos não triviais para aglomerados de computadores ou grades de computação, bastando algumas adaptações.

Estas adaptações foram comentadas no capítulo 3 e aplicadas em ambientes distribuídos, obtendo bons resultados. Podemos destacar algumas adaptações como a sincronização ajustada pela natureza de troca de mensagens não-bloqueante e bloqueante, o uso de granularidade grossa baseada nos superpassos do modelo realístico CGM (Coarse Grained Multicomputers), o emprego de pipeline de operações para minimizar o tempo de elementos ociosos e, por último, implementar a tolerância a falhas para este tipo de solução.

No capítulo 4 introduzimos um estudo sobre os conceitos envolvidos em um sistema tolerante a falhas. Diversas técnicas de tolerância a falhas foram citadas, embora a técnica de redundância de componentes tenha sido a mais explorada. Esta técnica viabilizou a construção de um algoritmo capaz de utilizar recursos ociosos em prol de uma melhor performance.

O próximo passo foi projetar um algoritmo para a inclusão de redundância em algoritmos sistólicos. A implementação deste algoritmo foi desenvolvida e testada no cluster Biowulf da rede IME. Além da interface de troca de mensagens MPI, foi necessário o uso da biblioteca Pthreads para evitar uma indesejável sincronização entre os grupos de redundância, consequência do gerenciamento da redundância.

Após projetar um algoritmo sistólico, podemos duplicar cada um dos elementos formando duplas ou grupos de redundância. Caso um dos elementos do grupo de redundância falhar, ou tardar, o elemento redundante a ele realiza toda a computação e, em seguida, atualiza o elemento atrasado. Para o controle interno do grupo de redundância, usamos o algoritmo Token Ring que se mostrou ágil na passagem de pequenas informações entre os elementos. Enquanto isso, a distribuição de dados em sístoles é praticada de forma independente entre os níveis de redundância.

Para obter resultados experimentais, adotamos o problema para multiplicação de

matrizes e implementamos três versões deste algoritmo: uma versão seqüencial, uma versão sistólica e uma sistólica com redundância. Os resultados obtidos mostraram ser satisfatórios pois mesmo com o overhead gerado pelo controle de redundância, o tempo de execução do algoritmo redundante não ficou muito distante do algoritmo sistólico sem redundância. Em experimentos simulando ambientes heterogêneos, o tempo de execução obtido pela redundância foi inferior ao tradicional sem redundância. Isso se deve a uma situação de corrida encontrada dentro do grupo de redundância. Por exemplo, ao receber as matrizes de entrada, todos os elementos iniciam a execução da computação e aquele que terminar em primeiro lugar envia o resultado obtido para os companheiros derrotados.

O melhor resultado da implementação do algoritmo sistólico redundante foi no experimento de redução da capacidade de um dos elementos que compõem a estrutura. No algoritmo tradicional, o desempenho do algoritmo é proporcional ao desempenho do pior elemento da estrutura. Já para o algoritmo redundante, o tempo de execução permaneceu constante ao reduzir a capacidade de processamento de um elemento em até 16×. Resumindo, a redundância é uma boa ferramenta para se obter, ou manter, um bom desempenho em ambientes distribuídos cujos elementos de processamento são heterogêneos.

Durante este trabalho, também cogitou-se implementar um algoritmo sistólico com alguns elementos de reserva. Estes elementos seriam substituídos no caso de algum elemento titular falhar ou apresentar um desempenho abaixo do esperado. Essa idéia foi descartada devido à necessidade de armazenar dados de diversos elementos da estrutura, que exigiria rodadas de comunicação envolvendo muitos elementos, fugindo das características dos algoritmos sistólicos.

Os resultados experimentais foram obtidos em um ambiente de teste acadêmico, um cluster com 16 nós. A lógica por trás de algoritmos sistólicos exige um grande número de nós, e com o emprego da redundância deste trabalho, esse número é no mínimo duplicado. Logo, os nossos testes ficaram limitados ao número de nós disponíveis pelo cluster. Em trabalhos futuros, seria bastante interessante testar os algoritmos sistólicos em grades de computação reais com centenas de nós.

Empresas e instituições de ensino podem aplicar a técnica de algoritmos sistólicos redundantes em diversos problemas paralelos, utilizando a infra-estrutura já exis-

tente, sem necessitar de grandes investimentos de aquisição e manutenção.

Em conjunto com os professores Siang Wun Song e Kunio Okuda, submetemos o artigo *Heterogeneous and fault-tolerant grid and cluster computing through redundancy* [20] para a *International Conf. on Grid and Pervasive Computing GPC 2006*, em Taiwan.

Referências Bibliográficas

- [1] P. Anderson, T; Lee. *Fault tolerance - principles and practice*. Englewood Cliffs, Prentice Hall, 1981.
- [2] A Avizienis. Infrastructure-based design of fault-tolerant systems. *Proceedings of the IFIP International Workshop on Dependable Computing and its Applications. DCIA 98, Johannesburg, South Africa, January 12-14, 1998. p. 51-69.*
- [3] J. P. Avizienis, A; Kelly. Fault tolerance by design diversity - concepts and experiments. *Computer, New York, 17(8):67-80, Aug. 1984.*
- [4] K. Birman. Building secure and reliable network applications. 1996.
- [5] Rajkumar Buyya. Building secure and reliable network applications. 1996.
- [6] A. Chen, L.; Avizienis. N-version programming: a fault tolerance approach to reliability of software operation. *In: Annual International Symposium on Fault-Tolerant Computing, 8, 1978. Proceedings. New York, IEEE, 1978. p. 3-9.*
- [7] C.J. Creveling. Increasing the reliability of electronic equipment by the use of redundant circuits. *Proceedings of the IRE. New York, 44(4):509-515, abr. 1956.*
- [8] H.T. Kung e C.E. Leiserson. Systolic arrays for VLSI. *in: Introductions to VLSI Systems, C.A. Mead and L.A. Conway, Chapter 8.3, Addison-Wesley, 1980.*
- [9] M. J. Foster e H. T. Kung. The design of special-purpose vlsi chips. *Computer, 13, p. 26-40, 1980.*

-
- [10] H. Mongelli e S. W. Song E. N. Cáceres. Algoritmos paralelos usando cgm/pvm: Uma introdução. *XXI Congresso da Sociedade Brasileira da Computação, Jornada de Atualização em Informática*, 2001.
- [11] U. K. Hayashida e S. W. Song. Desenvolvimento de algoritmos paralelos para aglomerados com a técnica de paralelização de laços. *Dissertação de Mestrado, IME-USP*, 2004.
- [12] Kunio Okuda e Siang W. Song. Um algoritmo de multiplicação de matrizes para implementação em vlsi. *I Congresso da Sociedade Brasileira de Microeletrônica, Campinas*, pp.383-393, 1986.
- [13] P. Quinton e Y. Robert. Algorithmes et architectures systoliques. *Masson, Paris*, 1989.
- [14] Ian Foster. What is the grid? a three point checklist. <http://www.fp.mcs.anl.gov/foster/Articles/WhatIsTheGrid.pdf>.
- [15] T. B.; Lala J. H. Hopkins, A. L; Smith. Ftmp - a highly realible fault-tolerant multiprocessor for aircraft. *Procedings of the IEEE, New York*, 66(10):1221-1239, Oct. 1978.
- [16] M. Hsueh. Ftmp - fault injection techniques and tools. *IEEE Computer*, v. 30, n. 4, Apr. 1997.
- [17] P. Jalote. *Fault Tolerance in Distributed Systems*. Prentice Hall, 1994.
- [18] T. S. Jansch-Porto, I. E. S; Weber. Recuperação em sistemas distribuídos. In: *XVI Jornada de Atualização em Informática, XVII Congresso da SBC, Brasília, 2-8 agosto de 1997. anais*. pp 263-310.
- [19] D. Johnson. The intel 432: a vlsi architecture for fault-tolerant computer systems. *Computer, New York*, 17(8):40-48, Aug. 1984.
- [20] S. W. Song K. Okuda and M. T. Yamamoto. Heterogeneous and fault-tolerant grid and cluster computing through redundancy. *International Conf. on Grid and Pervasive Computing GPC 2006 - Taiwan*, 2006.
- [21] J. A. Katzman. A fault-tolerant computing system. In: *Hawaii International Conference of System Sciences, 1978, Proceedings*. p. 85-102.

-
- [22] H. T. Kung. Let's design algorithms for vlsi systems. *Proceedings of Caltech Conference on VLSI*, p. 65-90, 1979.
 - [23] H. T. Kung. The structure of parallel algorithms. *Advances in computing*, 19, p. 65-111, 1980.
 - [24] H. T. Kung. Why systolic architectures. *Computers*, 15, p. 37-46, 1982.
 - [25] J. C. Laprie. Dependability: von concepts to limits. In: *Proceedings of the IFIP International Workshop on Dependable Computing and its Applications. DCIA 98, Johannesburg, South Africa, January 12-14, 1998*. p. 108-126.
 - [26] J. C. Laprie. Dependable computing and fault-tolerance: concepts and terminology. In: *Annual International Symposium on Fault Tolerant Computing*, 15. Ann Arbor, jun. 19-21, 1985. *Proceedings. New York, IEEE*, 1985. p. 2-11.
 - [27] T. S. Liu. The role of a maintenance processor for a general-purpose computer system. *IEEE Transactions on Computers. New York*, c-33(6):507- 517. June 1984.
 - [28] W. Lyons, R.E.; Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development. New York*, 6(3): 200-209, abr. 1962.
 - [29] Y. Robert e M. Tchuenté M. Cornard, P. Quinton. Parallel algorithms and architectures. *North Holland*, 1986.
 - [30] S. Mullende. *Distributed systems*. Addison-Wesley, ACM Press, New York, 1993.
 - [31] Dhiraj. Pradhan. *DFault-Tolerant Computer Design*. Englewood Cliffs, Prentice-Hall, 1996.
 - [32] Dhiraj. Pradhan. *Fault-Tolerant System Design*. Englewood Cliffs, Prentice-Hall, 1996.
 - [33] R. E. Miller e S. Winograd R. M. Karp. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14. p. 563-590, 1967.

-
- [34] Y. Robert. Systolic algorithms and architectures. *Automata Networks in Computer Science, Manchester University Press*, p. 187-228, 1987.
- [35] R. S. Siewiorek, D. P.; Swarz. *The Theory and Practice of Reliable System Design*. Bedford, Digital, 1982.
- [36] W. N. Toy. Fault-tolerant design of local ess processors. *Proceedings of the IEEE, New York*, 66(10):1126-1145, Oct. 1978.
- [37] K. Okuda e J. Panetta U. K. Hayashida, S. W. Song. Generating parallel algorithms for cluster and grid computing. *The 2005 International Conference on Computational Science - ICCS 2005. Lecture Notes in Computer Science, Vol. 3514, Vaidy S. Sunderam, Geert Dick van Albada, Peter M. A. Sloot et al. (Eds.), Springer-Verlag. Atlanta, U.S.A., 2005.*
- [38] J. Von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *In: Automata Studies, Shannon McCarthy eds. Princeton Univ. Press, 1956. p. 43-98.*
- [39] I.; Weber R. Weber, T.; Jansch-Pôrto. Fundamentos de tolerância a falhas. *Vitória: SBC/UFES, 1990. (apostila preparada para o IX JAI - Jornada de Atualização em Informática, no X Congresso da Sociedade Brasileira de Computação).*
- [40] T. S. Weber. Um roteiro para exploração dos conceitos básicos de tolerância a falhas. *Curso de especialização em Redes e Sistemas Distribuídos.*
- [41] T. S. Weber, R. F.; Weber. Um experimento prático em programação diversitária. *In: III Simpósio em Sistemas de Computadores Tolerantes a Falhas, SCTF, Rio de Janeiro, 20-22 set. Anais. Rio de Janeiro, 1989. p. 271-290.*
- [42] J. H. Wensley. Sift: design and analysis of fault-tolerant computer for aircraft control. *Proceedings of the IEEE, New York*, 66(10):1240-1254, Oct. 1978.