Um Algoritmo Distribuído para Detecção e Localização de Falhas em Redes tipo Hipercubo

Saulo Rodrigues do Nascimento, Marco Aurélio Amaral Henriques

DCA/FEEC/UNICAMP
Cx. Postal 6101
Campinas, SP, 13083-970
Fax: +55 19 289 1395
[saulo,marco@dca.fee.unicamp.br]

Resumo—
Este trabalho[1] propõe um algoritmo distribuído para detecção e localização de enlaces e nós falhos em redes tipo hipercubo quando estes, em conjunto, não excedem (n-1), sendo n a dimensão do hipercubo. Tal algoritmo é executado de forma paralela em todos os nós não falhos, utilizando trocas de mensagens entre vizinhos. Cada nó do hipercubo mantém em memória informação das condições de falha de sua vizinhança. São necessários três estágios de trocas de mensagens entre vizinhos para que todos os nós se informem da presença ou não de algum enlace ou nó vizinho defeituoso. O custo total da operação é da ordem de $O(n^2)$. A eficiência e a correção do algoritmo foram verificados através de diversos testes realizados em hipercubos de 8 a 64 nós nos quais falhas foram geradas aleatoriamente.

Palavras-chave— **Tolerância a falhas, hipercubo, detecção de falhas, redes de interconexão, (diagnóstico,) (multiprocessadores,) multicomputadores, algoritmos distribuídos**

I. Introdução

Programação paralela, sistemas distribuídos e multicomputadores são termos cada vez mais freqüentes na linguagem da tecnologia de computação atual. Isto deve-se ao fato do potencial inerente a tais termos ser bastante amplo e diversificado. A utilização de tais conceitos e recursos vem se espalhando e dinamizando velozmente e, com isso, a necessidade de um número maior de pesquisas e experimentos para melhor entendê-los e utilizá-los de forma mais eficiente torna-se evidente.

No que diz respeito à topologia de rede para sistemas multicomputadores, o hipercubo binário (ou simplesmente, hipercubo) vem se destacando por suas diversas características; entre elas podemos citar alta conectividade entre os nós, baixo diâmetro, fácil expansibilidade e tolerância a falhas.

Os hipercubos são em geral encontrados em forma de sistema multicomputador local, ou seja, vários computadores (processadores + memória) interconectados e contidos em um mesmo gabinete. Porém, seu conceito tem sido ampliado e hoje já podemos encontrar os chamados "hipercubos virtuais", que são máquinas interconectadas em rede local ou até mesmo através da Internet (como no caso do sistema JOIN [HUE 98]), que implementam os conceitos básicos e a topologia do hipercubo de forma lógica, aproveitando suas vantagens. Descrições de alguns sistemas que foram implementados com hipercubos podem ser encontradas em [NCU 90] e [HIL 85].

Em qualquer sistema composto de elementos interconectados, expansão é sinônimo de aumento de probabilidade de ocorrência de falhas. Portanto, tolerância a falhas é um aspecto de alta relevância em tais sistemas e tem atraído a atenção de muitos pesquisadores. A eficiência e a exatidão de aplicações executadas em tais sistemas podem ser completamente comprometidas no caso de ocorrência de falhas.

As falhas em sistemas baseados em hipercubo podem ser de várias naturezas; no entanto, está fora do escopo desse trabalho entrar em detalhes aprofundados sobre cada uma delas. Nos concentraremos em dois tipos principais:

Falha em Nó: Quando um nó permanece incomunicável com qualquer um de seus vizinhos;

Falha em Enlace (link): Quando um nó permanece incomunicável com um vizinho específico através do enlace que os conecta.

Este trabalho não trata os casos de falhas transientes e intermitentes que, apesar de serem mais freqüentes que as falhas permanentes [IYE 83] no que diz respeito aos nós, não se aplicam no caso dos enlaces. Em geral, as falhas que não são permanentes se apresentam como uma manifestação de uma falha a nível lógico (erro) e podem ser detectadas através de computadores que realizam auto-teste [REN 84] ou por uma série de testes funcionais que um computador realiza em outro [ARM 81] [KUH 81] [DIL 84]. As falhas permanentes, por outro lado, são

[1] Trabalho realizado com o apoio da CAPES/MEC

normalmente defeitos físicos. Desde que se tenha conhecimento de que um específico nó vem apresentando falhas transientes, os processos sendo executados nele podem ser eliminados e o algoritmo de detecção de falhas pode ser executado considerando-o como um nó com defeito físico. Desta forma, é possível combinar as duas abordagens de forma complementar.

Entre as formas de se contornar o problema de falhas em hipercubos, existem os sistemas que lançam mão de componentes reservas [REN 96], os que dividem o hipercubo em subcubos menores e aproveitam aqueles livres de falhas (*fault-free*) [CHE 97], e os que simplesmente deixam tal tarefa para as aplicações [WU1 95] [WU2 95] [WU 96] [BRU 94] [LEE 92] [LEU 96]. Cada uma delas apresenta vantagens e desvantagens [BAN 90]. Entretanto, acreditamos que as aplicações podem ser implementadas de maneira econômica, eficiente e tolerante a falhas, tirando proveito das peculiaridades do hipercubo.

Muitos algoritmos já foram propostos para fazer com que operações comuns em hipercubos, como *unicasting*, *multicasting*, *broadcasting e merging*, se tornem tolerantes a um certo número de falhas. Entre eles podemos destacar os algotimos propostos por J. Wu [WU1 95], [WU2 95] e [WU 96], J. Bruck [BRU 94], T. C. Lee e J. P. Hayes [LEE 92] e Y. Leu e S. Kuo [LEU 96].

Os autores desses algoritmos, normalmente, partem do princípio de que os pontos de falha do hipercubo já são previamente conhecidos pelos nós operantes. Muitos deles assumem a existência de um nó central capaz de detectar a existência de falhas e comunicá-las aos demais nós. Com o conhecimento então dos elementos que são falhos no sistema, e partindo do pressuposto que o número de falhas não excede um determinado limite, eles então propõem seus algoritmos.

A idéia da utilização de um nó central capaz de se comunicar diretamente com todos os nós do cubo é fácil de entender e aplicar. Entretanto, ela não é eficiente e não aproveita as vantagens do caráter distribuído do hipercubo.

Entre os métodos de detecção de falhas disponíveis na literatura, não há nenhum que especifique como um nó consegue diferenciar uma falha de nó de uma falha de enlace usando um método distribuído. Neste trabalho propomos uma forma distribuída, simples e eficiente de, em um hipercubo binário, identificar (e difundir a informação sobre) elementos falhos (sejam nós, enlaces ou uma combinação destes) desde que seu número não exceda n-1, onde n é a dimensão do hipercubo.

O artigo está organizado como segue. Os conceitos básicos e notação encontram-se na seção 2. A seção 3 detalha o método utilizado na detecção e localização das falhas, bem como lemas, teoremas e o algoritmo utilizado. A seção 4 apresenta os custos de tempo envolvidos na operação e os resultados dos teste realizados, e a seção 5 finaliza com as conclusões.

II. Conceitos e Notação

O hipercubo é uma topologia de rede de interconexão muito popular, e uma das mais versáteis e eficientes para processamento paralelo.

Cada nó em um hipercubo de n dimensões (ou n-cubo) está conectado a n outros nós, um em cada dimensão, por meio de enlaces de comunicação. Existem 2^n nós em um hipercubo e a cada um desses nós é conferido um rótulo de n bits do alfabeto {0,1}. Dizemos que dois nós são vizinhos se eles apresentam rótulos que diferem em apenas um bit. A posição do bit diferente em dois nós vizinhos indica a dimensão pela qual eles estão conectados. Assim, por exemplo, os nós 001 e 101 do 3-cubo da Fig. 1 são vizinhos e estão conectados pela dimensão 2.

Podemos representar também cada enlace com um rótulo de n bits e alfabeto {0,1, * }, onde " * " ocupa a posição do rótulo que indica a dimensão pela qual os nós estão conectados. Assim, o rótulo * 00 representa o enlace que interliga os nós 000 e 100, como mostrado na Fig.1.

Apesar de não ser uma condição necessária, assumimos que cada nó consegue comunicar-se simultaneamente com todos os seus vizinhos.

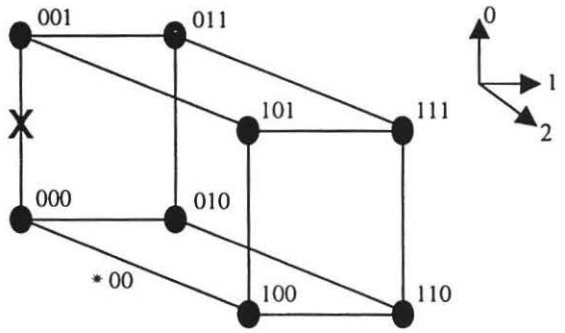

Fig. 1 Exemplo de Hipercubo de 3 dimensões e com um enlace falho

III. Algoritmo de Detecção e Localização de Elementos Falhos

O método de detecção de falhas em hipercubos que será apresentado baseia-se na troca de informações entre nós vizinhos. As informações recolhidas pelos nós são armazenadas em um vetor de n posições e alfabeto {0,1,2} denominado *fault* [LEU 96]. Essa troca de informações entre nós é na realidade uma pequena mensagem que cada nó envia aos seus vizinhos para efeito de teste. Ela pode ser realizada logo que a máquina é ligada ou sempre que se deseja testar o estado atual de falhas do sistema.

A operação é iniciada com todos os nós enviando mensagens de teste aos seus vizinhos, configurando a primeira etapa de troca de mensagens. Em caso de

assincronismo total entre os nós com relação ao início do processo, o algoritmo pode ser escrito de forma que um único nó inicie o processo e os demais sejam "despertados" assim que recebam a mensagem. A partir daí, a operação se desenvolverá sob um assincronismo parcial, controlado por *timeout*.

Inicialmente, todos os nós criam o vetor *fault* com todas as posições iguais a 1. Assim sendo, logo que um nó recebe uma mensagem, ele atualiza o vetor *fault* com um "0" na posição que indica a dimensão pela qual ele recebeu a mensagem. Após um certo período de tempo, caso não tenha recebido mensagem de um vizinho, ele assume a presença de falha, a qual pode ser de nó ou de enlace. Daí, o nó apresentará "1" em toda posição do vetor *fault* que corresponde a uma dimensão que apresenta falha.

Por exemplo, o nó 000 na Fig. 1 envia mensagem teste para os seus vizinhos 001, 010 e 100, porém só recebe resposta dos nós 010 e 100. Portanto sua palavra *fault* será [001], indicando que existe uma falha na dimensão 0. A princípio o nó 000 não tem como saber onde está a falha, se no nó 001, se no enlace 00 * ou em ambos. A forma de resolver tal problema será apresentada mais adiante.

Uma vez expirado o tempo de espera, os nós iniciam a segunda etapa de troca de mensagens enviando seu próprio vetor *fault* a seus vizinhos. Cada nó recebe então até n vetores *fault* e os organiza como mostrado a seguir.

Cada vetor *fault* será inserido em uma linha de uma matriz F, linha esta que representa a dimensão pela qual ele recebeu o vetor *fault*. Todas as posições da matriz F são iniciadas com 1 e, à medida que os vetores *fault* vão chegando, cada linha da matriz é atualizada. Vejamos o caso do nó 100. Ele recebe do nó 000 o vetor [001], do nó 101 o vetor [000] e do nó 110 o vetor [000]. Ao final ele terá então a seguinte matriz.

$$\begin{array}{ll} \text{Dimensão } 0 \Rightarrow & \\ \text{Dimensão } 1 \Rightarrow & \\ \text{Dimensão } 2 \Rightarrow & \end{array} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{array}{l} \\ \Rightarrow \text{recebida do nó 101} \\ \Rightarrow \text{recebida do nó 110} \\ \Rightarrow \text{recebida do nó 000} \end{array}$$

Após receberem os vetores *fault* dos vizinhos, obviamente daqueles por dimensões não falhas, os nós realizam a operação descrita a seguir.

Para cada posição p_j igual a 1 no vetor *fault* vindo de cada dimensão d_i execute os seguintes passos:

- Obtenha o valor da posição p_i do vetor *fault* vindo da dimensão d_j.
- Se $p_i = 0$, então envie ao nó vizinho pela dimensão d_i uma mensagem informando que ele possui um enlace falho na dimensão p_j.
- Se $p_i = 1$, não há informação suficiente para determinar a localização da falha, e o algoritmo se repete para outros valores de i e j.

Para o caso do exemplo acima, teremos apenas um vetor *fault* que apresenta valor 1, no caso aquele vindo de d_2. A posição que apresenta valor 1 será então p_0. O valor da posição p_2 para o vetor da dimensão d_0 é zero. Assim sendo o nó 100 enviará mensagem ao nó 000 informando que ele apresenta um enlace falho na dimensão 0. O envio da notificação aos nós que apresentam enlaces falhos constitui a terceira etapa da troca de mensagens do método. Note que um nó que apresenta enlace falho pode vir a receber esta notificação de mais de um nó vizinho.

O algoritmo detalhado, chamado *DETECT*, é apresentado na Fig. 2.

```
Begin     (node é a seqüência de n bits do nó e n é a
          dimensão do hipercubo)

   fault:= 1^n ;        (1^n representa uma seqüência de n
                         bits todos 1's)
   for i = 0 to n-1 do begin
     matriz[i]:= 1^n;  (inicia matriz com 1 em todos os
                         campos)
   endfor;

   for i = 0 to n-1 do begin
     viz := node ⊕ I_i;    (I_i é a seqüência de n bits
onde
                              apenas o i-ésimo bit é 1)
     envia mensagem teste ao vizinho viz pelo
     enlace i;
   endfor;

   While wait do begin
   (wait simboliza qualquer recurso de programação
    utilizado para criar uma espera necessária por
    mensagens)
     recebe mensagem pela dimensão j;
     fault[j] := 0;
   endwhile;

   for i = 0 to n-1 do begin
     viz := node ⊕ I_i;
     envia fault ao vizinho viz pelo enlace i;
   endfor;

   While wait do begin
     recebe palavra fault pela dimensão j
     matriz[j] := fault;
   endwhile;

   for i = 0 to n-1 do begin
      for j = 0 to n-1 do begin
          if (matriz[i])[j] = 1 and
             (matriz[j])[i] = 0 then
             informa vizinho viz pelo enlace i que
             seu enlace na dimensão j é falho;
          elseif
             não faz nada;
          endif
      endfor;
   endfor;

   While wait do begin
     recebe dimensão j que apresenta enlace
     falho;
     fault[j] := 2;
   endwhile;

end.
```

Fig. 2 Algoritmo *DETECT*

O algoritmo foi escrito objetivando clareza. É possível escrevê-lo de forma a priorizar o desempenho, mas em detrimento da clareza. Deixaremos isso a cargo da implementação. Dependendo da aplicação, pode ser necessário uma quarta troca de mensagens para que os nós recebam os vetores *fault* de seus vizinhos a fim de atualizarem a matriz *F*.

Será demonstrado a seguir como o Algoritmo *DETECT* é capaz de detectar e localizar até n-1 falhas em um n-cubo.

LEMA 1: Cada enlace em um n-cubo pertence a n-1 subcubos de ordem 2 (2-cubos) distintos.

DEMONSTRAÇÃO:

Os 2-cubos contidos em um hipercubo de maior dimensão podem ser representados por rótulos de n bits e alfabeto {0,1, * }. Eles são da forma $[0,1]^i * [0,1]^j * [0,1]^k$, onde $[0,1]^m$ é uma seqüência de m bits ($m \geq 0$). Por exemplo, na Fig. 1 temos o 2-cubo * * 1 formado pelos nós 001, 011, 101 e 111, e pelos enlaces * 01, * 11, 0 * 1 e 1 * 1. (Nota: apesar de seqüências de bits com um asterisco poderem ser interpretadas como sendo um 1-cubo, neste trabalho elas significam o enlace entre os dois nós deste 1-cubo.)

Suponhamos então que se queira formar um 2-cubo a partir de um enlace qualquer $[0,1]^i * [0,1]^j$. Para tal, devemos substituir um dos bits do rótulo deste enlace por um outro " * " a fim de se configurar a representação de um 2-cubo. Como o rótulo do enlace possui n bits e um deles já é " * ", existem apenas n-1 outras dimensões que podem ser ocupadas pelo segundo " * ". Logo, um enlace qualquer pode pertencer unicamente a n-1 2-cubos distintos. —

LEMA 2: Dois enlaces, ou um enlace e um nó que não estejam conectados, só tomam parte juntos em até 1 subcubo de ordem 2.

DEMONSTRAÇÃO:

Na representação de um 2-cubo os * 's ocupam dimensões específicas no rótulo. Um enlace e_1 qualquer cujo * está na dimensão d estará presente em um 2-cubo que apresentar um de seus * 's na dimensão d e for equivalente em 0's e 1's em todos os outros bits do rótulo, exceto onde se encontra o outro * . Isto é fácil de perceber pois o rótulo do 2-cubo deve abranger o rótulo do enlace. Assim, por exemplo, o enlace 001 * 0 faz parte do 2-cubo * 01 * 0, já que este abrange os enlaces 001 * 0 e 101 * 0.

Suponha agora que os enlaces e_1 e e_2 que apresentam o * nas dimensões d_1 e d_2, respectivamente, estão presentes em um mesmo 2-cubo. Caso d_1 e d_2 não sejam iguais, ao alterarmos o 2-cubo trocando pelo menos um dos * 's de posição, e_1 e/ou e_2 não poderão estar presente neste novo ou em qualquer outro 2-cubo. Caso d_1 e d_2 sejam iguais, ao alterarmos o 2-cubo trocando de posição o * que não corresponde à d_1 (e d_2), esta posição que ele ocupava tornar-se-á obrigatoriamente 1 ou 0. Como apenas um dos enlaces possui 1 ou 0 em tal posição, então a presença dos dois em outro 2-cubo qualquer torna-se impossível, já que os bits do rótulo do enlace não mais coincidirão com os do novo 2-cubo. Observe que, se a troca fosse do asterisco correspondente a d_1, os 2 enlaces já estariam fora do novo 2-cubo.

Vejamos o caso da presença de um enlace e e um nó N não conectado a e em um mesmo 2-cubo cujos * 's estão nas dimensões d_1 e d_2. Suponhamos também que e possui o * na dimensão d_1 e um "1" na dimensão d_2. Como um 2-cubo abrange 4 nós, dois deles apresentam "1" na dimensão d_2 (os conectados por e) e os outros dois, entre eles N, "0" na dimensão d_2. Ao alterarmos o 2-cubo mudando a posição do * de d_1, o 2-cubo não conterá mais e. Se alterarmos o 2-cubo mudando a posição do * de d_2, ela se tornará 1 ou 0. Se for 0, o novo 2-cubo não conterá mais e, já que e possui um bit 1 na posição d_2. Se for 1, o novo 2-cubo não conterá mais N. Assim sendo, a presença de e e N em outro 2-cubo qualquer torna-se impossível.

Observe que o caso de um enlace falho e um nó falho conectados está contido no caso mais geral de um nó falho, e por isso não é considerado no lema 2. —

TEOREMA 1: Na presença de um número de falhas inferior a n em um n-cubo, o algoritmo *DETECT* consegue detectar, localizar e distinguir todas elas.

DEMONSTRAÇÃO:

Para a demonstração, analisaremos primeiramente o caso do 2-cubo e em seguida generalizaremos para o caso de um n-cubo.

A Fig. 3(a) apresenta um 2-cubo com um enlace falho entre os nós 00 e 10. Vale ressaltar que o 2-cubo pode apresentar no máximo 1 falha sob nossas considerações.

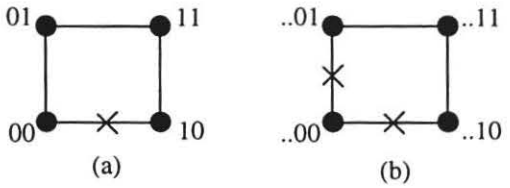

Fig. 3 Exemplos de falhas em um 2-cubo

Neste exemplo é fácil perceber que o nó 11 tendo conhecimento das falhas nos vizinhos 01 e 10 consegue avaliar a condição do enlace * 0. O mesmo pode-se dizer do nó 01 quando analisa as falhas de seus vizinhos 00 e 11. Após conferir os vetores *fault* de seus vizinhos, o nó 11 analisará a situação da seguinte forma: se meu vizinho na dimensão 0 (nó 10) não se comunica com seu vizinho na

dimensão 1 (nó 00) e o meu vizinho na dimensão 1 (nó 01) consegue se comunicar com seu vizinho na dimensão 0 (nó 00), então 00 não é falho e sim o enlace que o conecta ao meu vizinho na dimensão 0 é que é falho.

A idéia do algoritmo *DETECT* é analisar todos os 2-cubos do hipercubo que apresentam algum enlace falho. Pode acontecer de encontrarmos a situação mostrada na Fig. 3(b) para o caso de um hipercubo de dimensão maior que 2. Para este 2-cubo em particular percebe-se facilmente que o algoritmo *DETECT* não é capaz de fazer com que $[0,1]^i$ 11 e $[0,1]^i$ 01 percebam que $[0,1]^i$ * 0 é falho nem que $[0,1]^i$ 10 e $[0,1]^i$ 11 percebam que $[0,1]^i$ 0 * é falho. Porém, como o nosso objetivo é fazer com que todos os nós conectados por um enlace falho sejam notificados disso, basta garantir que pelo menos um de seus vizinhos faça tal notificação.

Se para cada enlace falho existir pelo menos um 2-cubo do qual ele faz parte e que não apresente nenhuma outra falha, assim como exemplificado no caso da Fig. 3(a), os nós que ele conecta sempre serão notificados da falha.

Mostraremos que tal 2-cubo sempre existirá para o caso de um n-cubo com menos de n falhas.

Suponha a existência de um enlace e falho. Ele pertence a n-1 2-cubos distintos (LEMA 1). Como o número máximo de elementos falhos no sistema não pode exceder n-1, além de e somente mais n-2 elementos falhos são permitidos. No pior caso cada um destes (n-2) elementos falhos formará um, e somente um (LEMA 2), 2-cubo com o enlace e, definindo um número máximo de (n-2) 2-cubos distintos. Portanto, o enlace e estará presente em pelo menos (n-1) - (n-2) = 1 subcubo como o da Fig. 3(a), o que viabiliza a detecção e localização deste enlace falho pelo Algoritmo *DETECT*. —

IV. Simulação e Análise do Algoritmo

Abaixo é detalhado o esforço computacional local (em cada nó) para cada etapa do algoritmo *DETECT*. A ordem do custo de cada etapa pode ser avaliada em cada passo executado.

Iniciação de variáveis	$O(n)$
Envio de mensagem teste a cada vizinho	$O(n)$
Recebimento e armazenamento das respostas	$O(n)$
Envio do vetor *fault* a cada vizinho	$O(n)$
Conferência, armazenamento e atualização de variáveis para cada resposta recebida	$O(n)$
Análise de cada palavra fault e informação aos nós vizinhos em caso de detecção de enlace falho	$O(n^2)$
Conferência, armazenamento e atualização de variáveis para cada notificação recebida	$O(n)$

Total = $O(n^2)$ passos.

Uma análise geral do código apresentado na Fig. 2 revela um custo total de operação em torno de $\mathbf{2n^2 + 11n}$ **passos.** Essa aproximação supõe que os passos do algoritmo apresentam tempo de processamento não muito discrepantes uns dos outros.

O algoritmo *DETECT* foi implementado em C e testado extensivamente em um hipercubo n-Cube2, da empresa NCUBE [NCU 90], de 64 processadores com 4 Mbytes de memória local cada um. As falhas de enlace e nó, num total de n–1 falhas, foram geradas aleatoriamente em cada uma das simulações e os testes realizados para valores de n variando de 3 (8 nós) a 6 (64 nós). Em todos os testes o algoritmo *DETECT* conseguiu detectar, distinguir e localizar todas as falhas. Os valores de tempo registrados são apresentados na TABELA 1. A coluna de tempo médio é referente à média dos valores registrados durante as simulações, em função de n, enquanto a coluna dos valores máximos apresenta valores que foram registrados para as configurações de falhas mais trabalhosas de detectar, como por exemplo as que provocam congestionamento de mensagens ou fazem com que um mesmo nó detecte vários enlaces falhos e por isso tenha que enviar várias mensagens durante o processo. A TABELA 2 apresenta a performance do algoritmo quando não existem falhas no sistema. É interessante notar que o algoritmo *DETECT* tem desempenho sensível ao número de falhas no sistema.

TABELA I
TEMPO PARA DETECTAR N-1 FALHAS

N° de dimensões n	Médio (µs)	Máximo (µs)
3	5824	6912
4	8298	10368
5	11409	14464
6	15332	20224

TABELA II
TEMPO DE EXECUÇÃO EM SISTEMA SEM FALHAS

N° de dimensões n	Tempo (µs)
3	4895
4	6551
5	8225
6	9924

Após alguns testes de performance realizados no n-Cube2, descobrimos que há uma diferença de tempo de processamento significativa (até 100 vezes maior) entre as chamadas de envio e recebimento de mensagens e as demais operações realizadas. Este fato deve-se à forma como estão implementadas as chamadas de tais funções e a interação entre o sistema de roteamento e o sistema operacional do n-Cube2. Em um sistema que possibilitasse

uma otimização dos processos de envio e recebimento de mensagens para que fossem feitos por hardware ao invés de totalmente monitorados por software, tal discrepância não existiria. Por isso, diferentemente da equação apresentada anteriormente, a equação que rege o desempenho do algoritmo utilizado nas simulações no n-Cube é:

$$195n^2 + 1453n + 8 \ (\mu s)$$

É evidente que o tempo gasto em cada passo varia de acordo com a implementação, mas o mais importante é confirmar que a complexidade do algoritmo é $O(n^2)$. Vale observar que todos os nós executam o Algoritmo *DETECT* simultaneamente e que a equação acima foi obtida do código implementado em C. A Fig. 4 apresenta um gráfico comparativo entre as curvas com os valores de tempo medidos durante as simulações (médio e máximo) e a curva da equação acima. Note que a curva dos valores médios é bem semelhante à da equação.

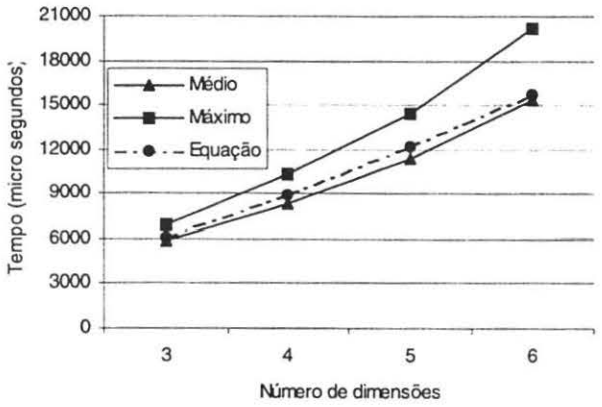

Fig. 4 Desempenho de *DETECT*

V. Conclusões

Algoritmos que provêem tolerância a falhas em hipercubos geralmente assumem que a informação sobre os elementos falhos já estão disponíveis aos nós, de alguma forma, e não detalham este aspecto do problema. Neste trabalho foi proposto um algoritmo distribuído para detecção e localização de falhas em hipercubos baseado em trocas de mensagens. O mesmo consegue detectar e localizar todas as falhas de enlace ou nó desde que, em conjunto, elas não ultrapassem o valor n-1, sendo n o número de dimensões do hipercubo. A complexidade do algoritmo é $O(n^2)$. Experimentos foram feitos e comprovaram que o algoritmo é não somente correto mas também apresenta desempenho satisfatório.

Trabalhos futuros incluem a expansão do algoritmo para cobrir falhas transientes e intermitentes.

VI. Referências

[LEE 92] Lee, Tze Chiang; Hayes, John P. *A Fault-Tolerant Communication Scheme for Hypercubes Computers.* IEEE Transactions on Computers, Vol. 41, No. 10, Oct. 1992.

[BRU 94] Bruck, Jehoshua. *An Optimal Broadcasting in Faulty Hypercubes.* Discrete Applied Mathematics 53, 3-13, 1994.

[WU1 95] Wu, Jie. *Unicasting in Faulty Hypercubes Using Safety Levels.* IEEE Transactions on Computers, Vol. 46, No. 2, pp. 241-247, Feb. 1995.

[WU2 95] Wu, Jie. *Safety-Levels - An Efficient Mechanism for Achieving Reliable Broadcasting in Hypercubes.* IEEE Transactions on Computers, Vol. 44, No. 5, May 1995.

[LEU 96] Leu, Yuh-Rong; Kuo, Sy-Yen. *A fault Tolerant Tree Communication Scheme for Hypercube Systems.* IEEE Transactions on Computers, Vol. 45, No. 6, Jun. 1996.

[WU 96] Wu, Jie. *Optimal Broadcasting in Hypercubes with Link Faults Using Limited Global Information.* Journal of Systems Architecture 42, 367-380, 1996.

[ARM 81] Armstrong, J. R.; Gray, F. G. *Fault Diagnosis in a Boolean n-Cube Array of Microprocessors.* IEEE Transactions on Computers, Vol. C-30, No. 8, Aug. 1981.

[CHE 97] Chen, Hsing-Lung; Tzeng, Nian-Feng. *Subcube Determination in Faulty Hypercubes.* IEEE Transactions on Computers, Vol. 46, No. 8, Aug. 1997.

[REN 96] Rennels, D. A. *On Implementing Fault-Tolerance in Binary Hypercubes.* Proc. 16th Int'l Symp. Fault-Tolerant Computing, pp. 344-349, Jul. 1996.

[HUE 98] Huerta, Eduardo J.; Henriques, Marco A. A. *Uma Introdução a JOIN: Um Sistema para o Processamento Massivamente Paralelo na Internet.* Relatório Técnico DCA-RT 02/98, FEEC/UNICAMP, Mar. 1998.

[NCU 90] NCUBE Corporation, *n*-Cube-2 Processor Manual. NCUBE Corporation, 1990.

[HIL 85] Hillis, W. D. *The Connection Machine.* Cambridge, Mass.: MIT Press, 1985.

[BAN 90] Banerjee, Prithviraj; Rahmeh, Joe T.; Stunkel, Craig; Nair, V. S., Roy, Kaushik; Balasubramanian, Vilay; Abraham, Jacob A. *Algorithm-Based Fault Tolerance on a Hypercube Multiprocessor.* IEEE Transactions on Computers, Vol. 39, No. 9, Sep. 1990.

[KUH 81] Kuhl, J. G.; Reddy, S. M. *Fault Diagnosis in Fully Distributed Systems.* Proc. 11th Int'l Symp. Fault-Tolerant Computing, pp. 100-105, Jun. 1981.

[DIL 84] Dilger, E.; Ammann, E. *System Level Self-Diagnosis in n-Cube Connected Multiprocessor Networks.* Proc. 14th Int'l Symp. Fault-Tolerant Computing, Kissimmee, FL, pp. 184-189, Jun. 1984.

[IYE 83] Iyer, R. K.; Rossetti, D. J. *Permanent CPU Errors and System Activity: Measurement and Modeling.* Proc. Real-Time Syst. Symp., 1983.

[REN 84] Rennels, D. A. *Fault Tolerant Computing — Concepts and Examples.* IEEE Transactions on Computers, Vol. C-33, pp. 1116-1129, Dec. 1984.