Diagnóstico Distribuído com Testes Imperfeitos Aplicado à Detecção de Estabilidade em Sistemas Baseados em MPI

Edson Tavares de Camargo[1,2], **Elias P. Duarte Jr.**[1], **Weyne Cassou Pietniczka**[1]

[1]Universidade Federal do Paraná (UFPR) – Programa de Pós-Graduação em Informática
Caixa Postal 19081 – 81531-980 – Curitiba – PR – Brasil

[2]Universidade Tecnológica Federal do Parará - Câmpus Toledo (UTFPR)
CEP: 85902-490 – Toledo – PR – Brasil

edson@utfpr.edu.br, elias@inf.ufpr.br, wcp10@inf.ufpr.br

Abstract. *In this work we introduce a novel system-level diagnosis model that is based on tests that may give inaccurate results. If a tested process fails to pass a test it is considered to be unstable, otherwise it is considered to be fault-free. A distributed diagnosis algorithm is proposed that allows fault-free processes to form a "stable core". Futhermore, if an unstable process passes a sequence of tests, the tester may start the execution of a consensus algorithm so that the state of the process may change and it is reintegrated to the stable core. The proposed model was implemented on a MPI-based system. Experimental results show the efficiency of the stable core of processes for parallel sorting based on the HyperQuickSort algorithm, implemented to tolerate up to* $N - 1$ *unstable processes while sorting up to 1 billion integers.*

Resumo. *Este trabalho apresenta um modelo para diagnóstico distribuído que assume testes imperfeitos, permitindo que um teste executado sobre um processo sem-falha indique instabilidade. Um algoritmo de diagnóstico é proposto sobre o modelo e permite que processos testados sem-falha formem um núcleo de processos estáveis. Além disso, um processo considerado instável, mas que responde corretamente a uma sequência de testes, pode ter a sua classificação modificada após a execução do consenso no núcleo de processos estáveis. O modelo proposto foi implementado em um sistema baseado em MPI. Resultados mostram a eficiência do núcleo de processos estáveis para a ordenação paralela baseada no algoritmo HyperQuickSort, implementado de forma a tolerar até* $N - 1$ *processos instáveis na ordenação de até 1 bilhão de inteiros.*

1. Introdução

O diagnóstico em nível de sistema é tradicionalmente usado para identificar quais unidades de um sistema estão funcionando corretamente, de acordo da sua especificação, e quais se encontram falhas. O diagnóstico é baseado no resultado de testes executados entre as unidades do sistema (neste trabalho usamos o termo "processo" como sinônimo de unidade). O primeiro modelo de diagnóstico é o PMC [Preparata et al. 1967], o qual assume que uma unidade sem-falha determina com precisão o estado de outra unidade testada. Essa premissa permanece intacta nos modelos de diagnóstico posteriormente propostos [Ye and Hsieh 2013, Weber et al. 2012].

A implementação de um teste perfeito, no qual uma unidade sempre determina com precisão o estado de outra unidade, apresenta múltiplos desafios em sistemas reais. Um dos problemas é garantir a completude do procedimento de teste empregado, que realmente deve ser capaz de detectar todas as situações de exceção possíveis. Outro problema é determinar o tempo máximo de espera de uma mensagem, tendo em vista que, em geral, os limites de tempo máximos para execução de um processo e tempo de transmissão de uma mensagem não são conhecidos. Desta forma, pode não ser possível um testador determinar sempre com precisão o estado de um processo testado. Este trabalho propõe um novo modelo de diagnóstico em nível de sistema no qual os testes são, por definição, imperfeitos. Em particular, um teste executado sobre um processo que não sofreu falha pode indicar uma suspeita de instabilidade.

Apesar dos testes serem "imperfeitos" é importante destacar que devem ser, na verdade, projetados para realizar sua tarefa perfeitamente, tanto quanto possível. Isto é, o modelo de diagnóstico proposto neste trabalho assume os *mesmos* testes definidos no modelo PMC: a unidade testadora executa uma bateria de testes na unidade testada que permite identificar a falha. Por outro lado, levamos em conta que, mesmo projetando testes perfeitos, sua execução pode gerar imprecisões no resultado. Observe que se o resultado de um teste indica que a unidade testada está sem-falha, então ambas as unidades, testada e testadora, estão sem-falhas (corretas), bem como a infraestrutura responsável pela entrega das mensagens. Na situação que pode levar a uma imprecisão, considera-se que, apesar do processo testado estar correto, alguma condição fora da normalidade (por exemplo: lentidão devida a uma carga elevada) permite a classificação do resultado do teste como indicador de instabilidade.

Em particular, sempre que ocorre um *timeout* no resultado do teste, o testador considera o processo testado como *instável*. É impossível diferenciar um processo instável de um processo falho, ambos recebem a mesma classificação no modelo. Observe que se um mesmo processo é testado por dois testadores distintos, um deles pode classificar o processo testado como instável e o outro testador classificá-lo como correto.

O objetivo do modelo de diagnóstico é encontrar em um sistema um *núcleo de processos estáveis* que se classificam como estáveis entre si. A estabilidade de um processo é definida pelo fato de que todos os testadores do processo o consideram sem-falhas. Os resultados de testes são propagadas entre os processos, permitindo que o o núcleo de processos estáveis se forme, removendo todo processo considerado instável.

Uma estratégia é definida para permitir que processos que são classificados como instáveis retornem ao sistema. Um processo instável que é testado como sem-falha por ζ (zeta) testes consecutivos por outros processos estáveis pode ter a sua classificação modificada após uma rodada de consenso entre os processos do núcleo de processos estáveis. Dessa forma, o núcleo estável decide se reintegra o processo classificado como instável. A identificação dos processos instáveis pode ser vista como uma implementação de um detector de falhas [Chandra and Toueg 1996]. No entanto, a formação e a manutenção do núcleo estável é particularidade do modelo proposto.

O modelo proposto foi implementado em um sistema MPI (*Message Passing Interface*). O diagnóstico foi implementado subjacente ao algoritmo de ordenação paralela *HyperQuickSort* [Wagar 1987]. O *HyperQuickSort* ordena os números em processos or-

ganizados na forma de um hipercubo virtual. No nosso trabalho o *HyperQuickSort* foi adaptado para se reconfigurar em tempo de execução a fim de manter a ordenação sem perdas com até $N - 1$ processos instáveis, onde N é o número total de processos.

Na implementação, foi seguida a mais recente especificação de tolerância a falhas em MPI, a *User Level Failure Mitigation* (ULFM) [Bland et al. 2012a]. Na especificação ULFM, por padrão, a falha é reconhecida localmente, ou seja, é detectada somente por um processo que se comunica com um processo falho. A biblioteca oferece mecanismos para que uma falha seja conhecida pelos demais processos. Desta forma, uma estratégia de comunicação global das falhas é implementada por meio da biblioteca ULFM e o seu desempenho comparado com o desempenho do modelo de diagnóstico. Resultados são apresentados para a ordenação de até 1 bilhão de inteiros e confirmam o bom desempenho do núcleo de processos estáveis proposto. Resultados de simulação são também apresentados, nos quais a reintegração de processos instáveis ao núcleo estável é feita através da execução do algoritmo Paxos [Lamport 1998].

Este trabalho segue organizado da seguinte forma. A Seção 2 apresenta os modelos de diagnóstico em nível de sistema e a tolerância a falhas em MPI. A Seção 3 detalha o modelo proposto. A Seção 4 aborda a implementação do modelo. A Seção 5 apresenta os resultados experimentais. Por fim, a Seção 6 apresenta a conclusão.

2. Trabalhos Relacionados

Este trabalho se fundamenta em duas grandes áreas: diagnóstico em nível de sistema e tolerância a falhas para MPI. Uma breve revisão de ambos é apresentada a seguir.

2.1. Diagnóstico em Nível de Sistema

O diagnóstico em nível de sistema é uma abordagem para identificar quais unidades de um sistema estão funcionando corretamente, de acordo com a sua especificação, e quais se encontram falhas. O diagnóstico se baseia no resultado de um conjunto de testes onde uma unidade é capaz de testar o estado de outro unidade.

O primeiro modelo de diagnóstico em nível de sistema é o modelo PMC [Preparata et al. 1967], cujo nome provém das iniciais de seus autores: Preparata, Metze e Chien. O modelo PMC assume um sistema S composto por um conjunto de N unidades independentes e não necessariamente idênticas que formam o conjunto $U = \{u_0, u_1, ..., u_{N-1}\}$. Nesse modelo, cada unidade u_i, também chamada nodo i ou processo i, pode estar em um entre dois possíveis estados: falho ou sem-falha. No modelo PMC, uma unidade é testada como um todo e o estado da unidade não muda durante o diagnóstico. Um teste envolve a aplicação controlada de estímulos e a observação da resposta correspondente retornada pela unidade testada. Um teste é definido na verdade como uma "bateria de testes" sendo adaptado para a tecnologia do sistema. Uma unidade que realiza um teste também é chamada de unidade testadora.

Uma importante definição no modelo PMC assume que uma unidade sem-falha determina corretamente o estado de outra unidade testada. E, além disso, é capaz de reportar os resultados dos testes com precisão. Mais precisamente, com base no resultado dos estímulos aplicados, o resultado do teste é classificado como *pass* (0) ou *fail* (1). Nenhuma definição é feita sobre uma unidade falha, isto é, os testes executados por uma unidade falha podem produzir resultados incorretos. A definição de qual unidade é a

testadora e qual é a testada é chamada de assinalamento de testes (*connection assigment*) e é representado por um grafo direcionado. O conjunto de resultados de todos os testes realizados é chamado de síndrome do sistema. A síndrome é processada por uma entidade externa, chamada de observador central, que realiza o diagnóstico do sistema, ou seja, determina o estado de todas as unidades do sistema.

De forma similar à proposta do presente trabalho, um modelo de diagnóstico que modificou a forma como que os resultados de testes são interpretados é o modelo BGM [Barsi et al. 1976]. As suas suposições básicas são as seguintes: cada teste é executado por uma única unidade; nenhuma unidade testa a si mesma; e, para qualquer par de unidades u_i, u_j, a unidade u_i executa pelo menos um teste na unidade u_j. O modelo de diagnóstico é definido da seguinte forma:

- Se u_i está sem-falha, o resultado do teste é 0 se u_j está sem-falha; o resultado do teste é 1 se u_j está falho;
- Se u_i está falho e u_j está sem-falha, ambos resultados são possíveis;
- Se u_i e u_j estão falhos, o resultado dos testes é necessariamente 1.

No modelo BGM se o resultado do teste $a_{i,j} = 0$, isto é, a unidade i testa a unidade j como sem-falha, então é possível concluir que u_j não é falho. Porém, se $a_{i,j} = 1$ então não é possível que tanto u_i quanto u_j estejam sem-falha. Nenhuma outra possibilidade pode ser excluída dados os resultados realizados por u_i sobre u_j.

No diagnóstico adaptativo proposto por [Nakajima 1981, Hakimi and Nakajima 1984], ao invés de determinar o conjunto fixo de testes sempre executado por cada nodo, como nos modelos anteriores os nodos determinam os testes que devem executar unicamente a partir dos resultados dos testes realizados na rodada anterior. No diagnóstico distribuído, proposto por [Hosseini et al. 1984], os próprios nodos sem-falhas do sistema diagnosticam o estado de todas as unidades. Os nodos sem-falha executam os testes e trocam os resultados dos testes entre si.

O diagnóstico em nível de sistema adaptativo e distribuído, através do algoritmo *Adaptive-DSD*, foi proposto por [Bianchini and Buskens 1991]. O algoritmo *Adaptive-DSD* é executado em cada nodo do sistema em intervalos de testes pré-definidos. Uma rodada de testes é definida como o período de tempo no qual todos os nodos do sistema executam os seus testes pelo menos uma vez. Todos os nodos sem-falha completam o diagnóstico em no máximo N rodadas de testes.

O diagnóstico hierárquico foi proposto com o objetivo de reduzir a latência do diagnóstico adaptativo e distribuído [Duarte and Nanya 1998]. Nesse modelo, os nodos são organizados em *clusters* virtuais de tamanho progressivo. O algoritmo *Hierarchical Adaptive Distributed System-level Diagnosis* (*Hi-ADSD*) tem uma latência de diagnóstico de no máximo ${log_2}^2 N$ rodadas de testes para um sistema com N nodos.

2.2. Tolerância a Falhas em MPI

O padrão MPI (*Message Passing Interface*) define um dos mais importantes modelos para o desenvolvimento de aplicações paralelas e distribuídas baseado no paradigma de troca de mensagens. O MPI consiste de um conjunto de bibliotecas de funções padronizadas pelo MPI-Fórum [MPI Forum 2013]. O modelo de troca de mensagens se destina a ambientes computacionais em que os nodos estão conectados através de uma rede de

computadores, cada um dos quais mantendo memória local. As principais rotinas são de comunicação e sincronização de tarefas. Nesse ambiente, um requisito fundamental, porém ausente na norma MPI, é a tolerância a falhas [MPI Forum 2013]. Além da ausência de uma especificação formal de tolerância a falhas, o suporte a esse requisito pelas implementações de MPI disponíveis é considerado inadequado [Bland et al. 2012b].

As abordagens tradicionais de tolerância a falhas para MPI geralmente fazem uso da técnicas de *checkpoint-restart* [Egwutuoha et al. 2013]. Nessa técnica, que pode ser empregada com ou sem o suporte do MPI, o estado de um processo sem-falha em execução é armazenado periodicamente para permitir que o mesmo seja reiniciado após uma falha e assim reduzir a quantidade de trabalho perdido. O estado salvo é chamado de *checkpoint* e a ação de reiniciar o processo a partir de um *checkpoint* anterior é chamado de *rollback-recovery*. Embora eficiente, o custo para armazenar os *checkpoints* é alto e conforme cresce o número de processos envolvidos a confiabilidade do sistema diminui [Elnozahy and Plank 2004, Bland et al. 2012b] .

Diferente da abordagem de *checkpoint-restart*, na qual a execução da aplicação é interrompida e depois reiniciada, outras técnicas de tolerância a falhas, como Replicação, Migração de Processos, Transações e ABFT (*Algorithm-Based Fault Tolerant*) possuem em comum o objetivo de continuar a execução da aplicação apesar de falhas [Bland et al. 2013]. Dessa forma, em comparação com *checkpoint-restart*, essas técnicas diminuem o volume de entrada e saída de dados, o tempo de inatividade e a sobrecarga causada pela perda de computação. Em particular, a técnica de ABFT faz uso das propriedades do algoritmo da aplicação para recuperá-la de falhas durante a sua execução, como se ignorasse a existência de falhas [Du et al. 2012].

O padrão MPI não define o comportamento preciso da aplicação perante falhas. Tal fato se torna um empecilho para a ampla adoção de técnicas de tolerância a falhas pelos desenvolvedores de aplicações MPI. De forma geral, a interrupção de um único processo em uma aplicação MPI devido a uma falha implica em interromper toda a aplicação [MPI Forum 2013]. Diversos trabalhos desde a origem do padrão MPI tratam de forma particular a tolerância a falhas em suas implementações. Nesse sentido, destacam-se os seguintes trabalhos: o FT-MPI [Fagg and Dongarra 2000] e FT/MPI [Batchu et al. 2004] e [Gropp and Lusk 2004]. Desses trabalhos, o mais promissor, denominado FT-MPI (*Fault-Tolerant MPI*), foi descontinuado.

O mais recente esforço do MPI-Fórum para padronizar a semântica de tolerância a falhas em MPI resultou em duas propostas: a *Run-through Stabilization Proposal* [Hursey et al. 2011] e a *User Level Failure Mitigation* (ULFM). Devido à inclusão de muitos novos construtores e a complexidade envolvida na sua implementação a primeira acabou não sendo adotada pelo padrão MPI. A ULFM é mais recente tentativa de padronização e consiste tanto em uma especificação da semântica de tolerância a falhas em MPI quanto em uma biblioteca para programação atualmente codificada como subprojeto do projeto Open MPI.

A detecção de falhas na ULFM é local, isto é, somente os processos que se comunicam com o processo falho a detectam. A falha considerada é por parada *crash* permanente. A norma ULFM permite que o desenvolver da aplicação escolha a técnica de tolerância a falhas que melhor se adequa a sua aplicação. A responsabilidade da ULFM

é informar quais condições específicas impedem que a entrega da mensagem ocorra com sucesso, sem que isso promova, como outrora, a interrupção automática da aplicação. Entre os requisitos observados na especificação ULFM se destaca a compatibilidade de código com as versões anteriores do MPI e o baixo impacto no desempenho da aplicação em estado livre de falhas [Bland et al. 2013]. Existe a expectativa de que a adotação da ULFM pelo padrão MPI se dê a partir das próximas versões da norma MPI.

O modelo de estabilidade proposto neste trabalho faz uso da biblioteca ULFM para detectar as falhas e assim oferecer aos desenvolvedores de aplicações MPI um núcleo de processos estáveis para execução de suas aplicações. Ao contrário do modelo proposto, A ULFM não permite que processos instáveis retornem a execução.

3. Modelo Proposto

O modelo de diagnóstico assume um sistema *S* representado por um grafo completo $G = (V, E)$, onde V é o conjunto de N vértices, $\{n_0, n_1, ..., n_{N-1}\}$ e E o conjunto de arestas, sendo que existe uma ligação entre quaisquer dois vértices (n_i, n_j). Neste trabalho, nos referimos a um vértice n_i também como nodo i ou processo i. A ligação entre dois nodos se dá por um canal confiável. Cada nodo i assume um de dois estados, *estável*, ou *instável*. Um *evento* é definido como a troca de estado de um nodo, de estável para instável ou vice-versa. O modelo é assíncrono, isto é, não há premissas temporais. O modelo de falhas de processos é o modelo *crash*.

A estabilidade de um nodo é definida pelo fato de que nenhum outro nodo que o testa o considera instável. Para ser considerado estável, ou sem-falha, o nodo precisa responder a requisições de acordo com a sua especificação e dentro de um limite de tempo. As informações sobre os resultados de testes são propagadas entre os processos, permitindo que um núcleo de *processos estáveis* se forme.

Um nodo executa testes periodicamente em um intervalo de testes fixo, de acordo com o seu relógio local, que pode ir de alguns nanosegundos a minutos dependendo da tecnologia adotada no modelo. Um teste é ação do nodo testador de verificar o estado do nodo testado. A especificação do procedimento de testes depende frequentemente da tecnologia adotada. Um nodo i ao testar um nodo j como estável recebe de j as informações sobre os estados dos nodos testados por j. Sendo assim, se o nodo j testou um nodo k como instável, essa informação chegará a i. Mesmo que o nodo i considere o nodo k estável atualizará o estado de k devido à nova informação recebida de j. Dessa forma, o nodo k é excluído do núcleo de processos estáveis. A Figura 1 apresenta um grupo de oito processos onde os nodos 0, 2, 5 e 7 formam um núcleo estável.

Uma *rodada de testes* é caracterizada como o período de tempo no qual cada nodo realiza os seus testes. A latência do algoritmo é definida como o tempo necessário para que nodos estáveis recebam informações sobre um determinado evento. Diferentemente do modelo PMC, que assume que todo nodo sem-falha identifica corretamente o estado do nodo testado, no nosso modelo um teste não é perfeito, ou seja, é possível que o testador se engane em relação ao estado do nodo testado. Concretamente, isso se deve à impossibilidade de aguardar indefinidamente pela resposta a um teste executado, pois um nodo i que deixa de responder a requisições de um nodo j não necessariamente se encontra falho. O limite de tempo em que um nodo deve aguardar pela resposta de teste é calculado de forma adaptativa segundo o algoritmo do TCP [Jacobson and Karels 1988].

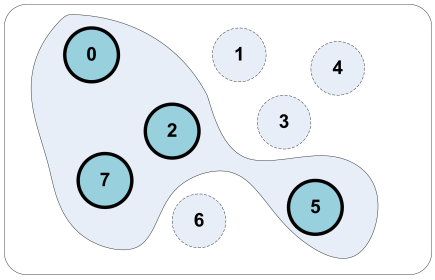


Figura 1. Núcleo estável formado pelos processos 0, 2, 5 e 7.

A estratégia de testes determina o *grafo de testes*, também chamado $T(S)$. $T(S)$ é um grafo direcionado onde os seus nodos são nodos do sistema *S*. Há uma ligação do nodo i ao nodo j se o nodo i testou o nodo j como estável e não recebeu a informação de que o nodo j se encontra instável. Dessa forma, $T(S)$ representa o núcleo de processos estáveis do sistema. No algoritmo proposto, inicialmente o estado de cada nodo é desconhecido. Cada nodo testa todos os demais nodos a cada intervalo. Um nodo j propaga a sua visão sobre os estados dos demais nodos do sistema quando o nodo i o testa como estável.

Um nodo que se mantém instável na avaliação de pelo menos um processo pode ter o seu estado modificado para estável após uma rodada de consenso envolvendo todos os nodos estáveis. Um nodo i ao testar o nodo k como estável por um número ininterrupto ζ de rodadas de testes pode acionar o algoritmo de consenso a fim de reingressar o nodo k ao núcleo de processos estáveis. O algoritmo de consenso usado neste trabalho foi o Paxos [Lamport 1998].

4. Implementação de um Núcleo de Processos Estáveis em MPI

O modelo de diagnóstico proposto foi implementado em um sistema MPI através de *threads*, que constantemente monitoram o estado dos nodos do sistema. Uma vez que o último nível de *threads* atualmente oferecido pela biblioteca ULFM não permite que múltiplas chamadas MPI sejam feitas simultaneamente por cada *thread*, o modelo também foi implementado sem o uso de *threads*. Nessa abordagem, o monitoramento é invocado pela aplicação. Também foi implementado, para fins de comparação com o modelo, uma abordagem de detecção global de falhas via ULFM. Tal abordagem exige revogar o comunicador MPI e restabelecer um novo comunicador, excluindo os processos falhos.

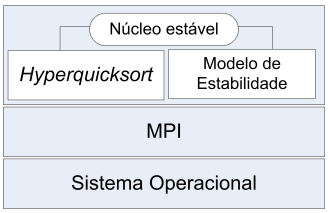


Figura 2. Arquitetura do sistema implementado.

O diagnóstico foi implementado subjacente ao algoritmo de ordenação paralela *HyperQuickSort* (Figura 2), o qual foi implementado também em MPI e adaptado para se autoconfigurar durante a sua execução. A implementação considera que processos se

tornam instáveis e deixam de fazer parte do núcleo de processos estáveis fornecidos pelo modelo. O retorno de um nodo instável ao núcleo estável foi implementado através de simulação.

4.1. Ordenação no Hipercubo Tolerante a Falhas

O algoritmo *HyperQuickSort* consiste na versão paralela do algoritmo *Quicksort* [Wagar 1987]. Os processos são organizados em *clusters* virtuais de tamanho regressivo e o problema consiste em ordenar um conjunto de $K = \{a_0, a_1, ..., a_{k-1}\}$ números, por meio de um conjunto de 2^P processos $P = \{b_0, b_1, ..., b_{2^p-1}\}$. Inicialmente os K números são divididos igualmente entre os P processos. A ordenação ocorre em rodadas, onde pares de processos trocam conjuntos de números entre si de acordo com um número fornecido pelo processo pivô de cada *cluster*. Ao final de $\log_2 P$ rodadas de ordenação, um subconjunto de números está ordenado em ordem crescente em cada processo e o maior número da porção de números do processo b_i é menor ou igual ao menor número da porção de números do processo $b_i + 1$, onde $0 \leq i \leq P - 2$. Considera-se que cada tem acesso exclusivo a sua memória local e comunica-se pela infraestrutura de rede.

A função $C_{i,s}$ definida no algoritmo *Hi-ADSD* [Duarte and Nanya 1998] - que também segue a estrutura de um hipercubo virtual - foi utilizada para encontrar os pares de processos (i, j) que se comunicam durante as rodadas de ordenação. A Figura 3 apresenta a execução do algoritmo *HyperQuickSort* para 8 processos. Na primeira rodada de ordenação há 1 *cluster* com os 8 processos, sendo o processo 0 o pivô. Os seguintes pares de processos são formados de acordo com a função $C_{i,s}$: (0, 4), (1, 5), (2, 6) e (3, 7). Na segunda rodada existem dois *clusters* cada um com 4 processos. Os processos 0 e 4 são os pivôs de seus respectivos *clusters*. Por fim, na rodada 3, o processo se repete considerando os pares e os pivôs da Rodada 3. A cada rodada ocorre a troca de valores entre os processos e cada processo reordena a sua porção de números.

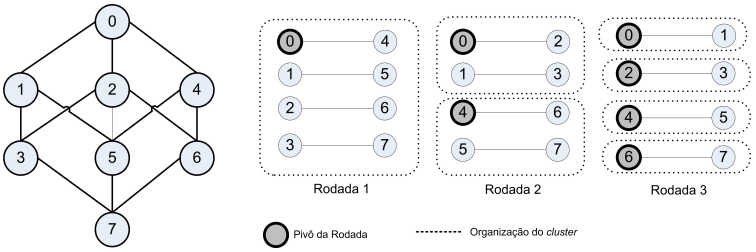


Figura 3. Algoritmo *HyperQuickSort* com 8 processos.

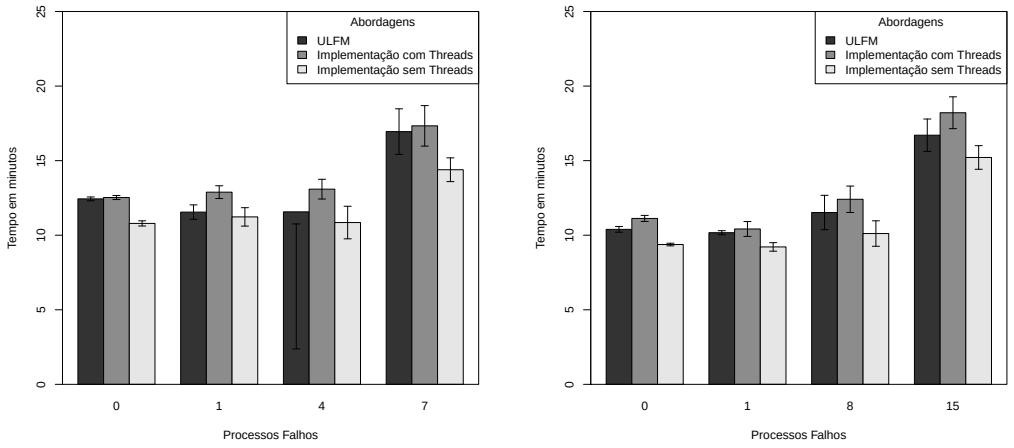
Para suportar até $N - 1$ processos instáveis, assume-se que as falhas ocorrem antes de uma rodada de ordenação. Durante a rodada de ordenação o processo se mantém estável. Cada processo mantém em um disco compartilhado por todos os processos a sua porção de números. Basicamente, para escolher o processo estável que substitui um processo instável, solicita-se à função $C_{i,s}$ o próximo processo estável disponível no *cluster*.

5. Resultados Experimentais

A seguir são apresentados dois conjuntos de resultados. O primeiro se refere aos resultados de implementação em MPI e o segundo se refere aos resultados de simulação sobre a manutenção do núcleo estável.

5.1. Desempenho do HyperQuickSort Executando Sobre o Modelo

Os experimentos foram executados no sistema operacional *Linux Kernel* 3.2.0 com 8 e 16 processadores AMD *Opteron* com 2.400 MHz. A rede do laboratório é *Ethernet* de 100 Mbps. Para todos os resultados apresentados cada experimento foi repetido 30 vezes; são apresentadas a média e intervalo de confiança de 95%.



(a) 8 nodos ordenando 1.024 x 10^6 números. (b) 16 nodos ordenando 1.024 x 10^6 números.

Figura 4. Comparativo entre o modelo proposto e a estratégia ULFM .

O desempenho do *HyperQuickSort* é avaliado de acordo com as três abordagens implementadas: executando sobre modelo de diagnóstico com e sem o uso de *threads* e executando sobre a abordagem de detecção global de falhas via ULFM. Cada abordagem é avaliada em 4 cenários: a) não há processo instável; b) 1 processo instável; e) 50% de processos instáveis e; d) $N - 1$ processos instáveis. As falhas ocorrem aleatoriamente antes do início de cada rodada de ordenação.

Os gráficos da Figura 4(a) e 4(b) apresentam os resultados para o cenário com 8 e 16 processos, respectivamente, ordenando 1.024 x 10^6 números. Observa-se que o tempo de ordenação com 16 nodos é inferior ao tempo com 8 nodos. O cenário com 16 nodos realiza a ordenação com uma rodada a mais de ordenação, o que exige mais troca de dados entre os processos. Por outro lado, porções menores de dados diminuem o tempo de ordenação de dados em cada processo. Veja que o objetivo não é apresentar o *speedup*, mas a capacidade do algoritmo de se manter em execução mesmo com alterações no núcleo estável

A substituição dos processos instáveis por processos estáveis na ocorrência de falhas não prejudicou o desempenho da ordenação. Os dados dos processos instáveis são incorporados pelos processos estáveis, o que diminui o custo da troca de dados entre os processos, ou seja, o custo dos *MPI_Sends* e *MPI_Receives*. A exceção é no cenário onde há $N - 1$ falhas, onde o custo da substituição de processos instáveis por estáveis se mostrou superior.

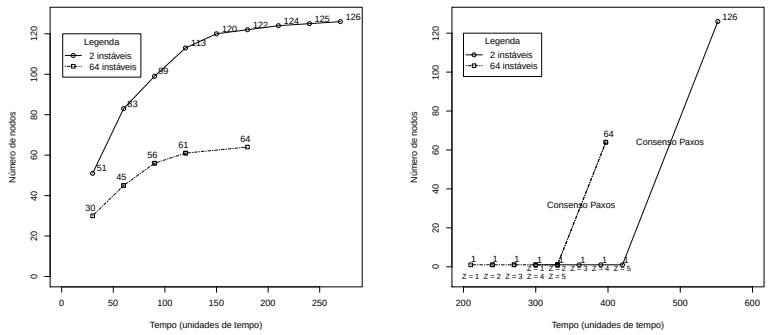
Dentre as três abordagens comparadas, o melhor desempenho foi observado quando o *HyperQuickSort* fez uso do modelo de diagnóstico na sua versão sem o uso de

threads. O segundo melhor desempenho foi apresentado pela abordagem implementada na ULFM. Porém, essa abordagem ficou próxima da abordagem que fez uso do modelo de diagnóstico na sua versão com as *threads*.

Discussão O custo de sincronização entre os processos é observado na abordagem ULFM, a qual exige a sincronização dos processos. Em contrapartida, a abordagem de diagnóstico com *threads* tem o custo da troca de contexto entre as execuções das *threads*, sem falar que o último nível de *threads* disponível é o *MPI_THREAD_SERIALIZED* que impede chamadas MPI simultâneas. A abordagem sem o uso de *threads* apresentou melhor desempenho pois não exige a mesma sincronização de processos da estratégia implementada na ULFM e não apresenta o custo das *threads*.

5.2. Resultados de Simulação

A seguir são apresentados resultados de simulação da detecção e da manutenção do núcleo estável na Figura 5. A simulação é realizada através do SMPL [MacDougall 1987]. São apresentados dois cenários de execução, ambos considerando 128 nodos. O primeiro resultado é de um cenário com dois nodos instáveis (linha contínua). O segundo resultado é de um cenário com 64 nodos instáveis (linha pontilhada). Os nodos se tornam instáveis no tempo 20. A cada 30 unidades de tempo os nodos se testam mutuamente. O retorno de um nodo ao núcleo estável ocorre com $\zeta = 5$.



(a) Detecção da instabilidade (b) Retorno do nodo à estabilidade

Figura 5. Simulação com 128 nodos.

No cenário com 2 nodos instáveis, Figura 5 (a), no tempo 30, 51 nodos detectam a instabilidade. Após isso, 83 nodos identificam os processos instáveis. Por fim, no tempo 270, todos os 126 nodos estáveis detectam a instabilidade nos dois nodos e o núcleo estável está formado. No tempo 300, Figura 5(b), um nodo detecta que um dos nodos instáveis se comporta como estável. Então, após 5 rodadas de testes sucessivas recebendo retorno aos testes executados, o nodo estável chama o consenso envolvendo o núcleo estável no tempo 420 para decidir se o nodo antes instável deve ter seu estado modificado para estável. Na sequência, o nodo instável tem o seu estado modificado para estável por todos os processos estáveis. O tempo de execução do consenso e a atualização do estado do nodo instável é indicado na linha diagonal da Figura 5(b). No cenário com 64 nodos instáveis, os nodos instáveis são identificados em menos tempo.

6. Conclusão

Neste trabalho apresentamos um novo modelo de diagnóstico que leva em conta a impossibilidade de implementar testes perfeitos em sistemas reais. Desta forma, um processo sem-falha pode não passar em todos os testes a que é submetido. Neste caso o nodo é considerado instável. As informações sobre os resultados de testes são propagadas entre os nodos permitindo que um núcleo de processos estáveis se forme. Um processo instável que passa a responder de forma correta por uma sequência ininterrupta de testes pode retornar ao núcleo de processos estáveis após a execução do consenso no núcleo de processos estáveis.

Um algoritmo baseado no modelo proposto foi implementado em um sistema baseado em MPI. Foram codificadas duas versões do algoritmo, com e sem *threads*, e foram comparadas com a estratégia recente de tolerância a falhas do MPI. Os melhores resultados foram sempre obtidos com a estratégia proposta implementada sem *threads*. Os resultados destacam a carência do suporte efetivo a *threads* na biblioteca ULFM.

Um simulador foi implementado para avaliar a reclassificação de processos antes instáveis como estáveis usando o Paxos como algoritmo de consenso. Resultados mostram a efetividade da solução proposta.

Trabalhos futuros incluem a implementação da reclassificação de processos com consenso no MPI. É fundamental analisar diversos aspectos da execução do consenso neste contexto: em especial o impacto frente a processos que intermitentemente se tornam falhos. A codificação de outros algoritmos paralelos e de outras estratégias de diagnóstico, incorporando o modelo de testes imperfeitos, também são caminhos a explorar.

Referências

Barsi, F., Grandoni, F., and Maestrini, P. (1976). A theory of diagnosability of digital systems. *IEEE Trans. on Computers*, C-25(6):585–593.

Batchu, R., Dandass, Y. S., Skjellum, A., and Beddhu, M. (2004). MPI/FT: A model-based approach to low-overhead fault tolerant message-passing middleware. *Cluster Computing*, 7(4):303–315.

Bianchini, R., J. and Buskens, R. (1991). An adaptive distributed system-level diagnosis algorithm and its implementation. In *Fault-Tolerant Computing, 1991*, pages 222–229.

Bland, W., Bosilca, G., Bouteiller, A., Herault, T., and Dongarra, J. (2012a). A proposal for User-Level Failure Mitigation in the MPI-3 Standard. Technical report, Department of Electrical Engineering and Computer Science, University of Tennessee.

Bland, W., Bouteiller, A., Hérault, T., Bosilca, G., and Dongarra, J. (2013). Post-failure recovery of MPI communication capability: Design and rationale. *International Journal of High Performance Computing Applications*, 27(3):244–254.

Bland, W., Bouteiller, A., Hérault, T., Hursey, J., Bosilca, G., and Dongarra, J. J. (2012b). An evaluation of user-level failure mitigation support in MPI. In *EuroMPI*, volume 7490 of *LNCC*, pages 193–203. Springer.

Chandra, T. D. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267.

Du, P., Bouteiller, A., Bosilca, G., Herault, T., and Dongarra, J. (2012). Algorithm-based fault tolerance for dense matrix factorizations. In *Proceedings of the 17th ACM/SIGPLAN Symposium on PPOPP*, pages 225–234, New Orleans, LA, USA. ACM Press.

Duarte, E. P. and Nanya, T. (1998). A hierarachical adaptive distributed system-level diagnosis algorithm. *IEEE Trans. Computers*, 47(1):34–45.

Egwutuoha, I. P., Levy, D., Selic, B., and Chen, S. (2013). A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326.

Elnozahy, E. N. and Plank, J. S. (2004). Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *IEEE Trans. Dep. Sec. Comp.*, 1(2):97–108.

Fagg, G. E. and Dongarra, J. (2000). FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *PVM/MPI*, volume 1908 of *LNCS*. Springer.

Gropp, W. and Lusk, E. L. (2004). Fault tolerance in message passing interface programs. *International Journal of High Performance Computing Applications*, 18(3):363–372.

Hakimi, S. L. and Nakajima, K. (1984). On adaptive system diagnosis. *IEEE Trans. Comput.*, 33(3):234–240.

Hosseini, S. H., Kuhl, J. G., and Reddy, S. M. (1984). A diagnosis algorithm for distributed computing systems with dynamic failure and repair. *IEEE Trans. Comput.*, 33(3):223–233.

Hursey, J., Graham, R. L., Bronevetsky, G., Buntinas, D., Pritchard, H., and Solt, D. G. (2011). Run-through stabilization: An MPI proposal for process fault tolerance. In *EuroMPI*, volume 6960 of *LNCS*, pages 329–332. Springer.

Jacobson, V. and Karels, M. J. (1988). Congestion avoidance and control. *ACM Computer Communications Review*, 18(4):314–329.

Lamport, L. (1998). The part-time parliament. *ACM Trans. Comput. Syst*, 16(2):133–169.

MacDougall, M. H. (1987). *Simulating Computer Systems. Techniques and Tools*. Computer Systems Series. MIT. Discrete Event Simulation mittels SMPL.

MPI Forum (2013). Document for a standard message-passing interface 3.0. Technical report, University of Tennessee, http://www.mpi-forum.org/docs/mpi-3.0.

Nakajima, K. (1981). A new approach to system diagnosis. *Proc. of the 19th Allerton Conf. on Communication, Control and Computing*, pages 697–706.

Preparata, Metze, and Chen (1967). On the connection assignment problem of diagnosable systems. In *IEEE Transactions on Electronic Computers*, volume 16.

Wagar, B. (1987). Hyperquicksort: A fast sorting algorithm for hypercubes. *Hypercube Multiprocessors*, 1987:292–299.

Weber, A., Kutzke, A. R., and Chessa, S. (2012). Energy-aware test connection assignment for the self-diagnosis of a wireless sensor network. *J. BCS*, 18(1):19–27.

Ye, T.-L. and Hsieh, S.-Y. (2013). A scalable comparison-based diagnosis algorithm for hypercube-like networks. *Reliability, IEEE Transactions on*, 62(4):789–799.