Uma Implementação MPI Tolerante a Falhas do Algoritmo Hyperquicksort

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

[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 - Campus Toledo (UTFPR)
CEP: 85902-490 – Toledo – PR – Brasil

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

Abstract. *Hyperquicksort is a parallel sorting algorithm based on a virtual hypercube. Current implementations of this algorithm do not support fault tolerance. This paper presents an implementation of Hyperquicksort that is able to tolerate up to* $n - 1$ *faulty processes, where n is the total number of processes employed for sorting. Our fault-tolerant Hyperquicksort version was implemented in accordance to the latest MPI fault tolerance specification, the User Level Failure Mitigation (ULFM). A implementation is described in which Hyperquicksort manages to reconfigure itself at runtime proceeding with its execution despite faulty processes. Experimental results show the efficiency of the implementation in ordering up to 1 billion integers.*

Resumo. *O Hyperquicksort é um algoritmo de ordenação paralela baseado em um hipercubo virtual. As implementações disponíveis desse algoritmo não consideraram a tolerância a falhas. Este trabalho apresenta uma implementação do Hyperquicksort que é capaz de tolerar até n-1 falhas de processos, onde n é o número total de processos empregados na ordenação. O Hyperquicksort tolerante a falhas foi implementado de acordo com a mais recente especificação de tolerância a falhas do MPI, a User Level Failure Mitigation (ULFM). Dessa forma, o Hyperquicksort é programado para se reconfigurar e continuar a sua execução, apesar de falhas de processos. Resultados mostram a eficiência da implementação na ordenação de até 1 bilhão de números inteiros.*

1. Introdução

A ordenação é uma das operações fundamentais em computação [Chen and Chung 2001]. O *Hyperquicksort* [Wagar 1987] é a versão paralela do algoritmo sequencial *Quicksort* [Hoare 1962] baseado na topologia de um hipercubo virtual. O hipercubo é uma estrutura largamente utilizada como topologia de interligação e comunicação e para a execução de algoritmos paralelos e distribuídos [Parhami 1999, Duarte, Jr. and Nanya 1998]. As propriedades do hipercubo como, por exemplo, diâmetro logarítmico, além do fato de ser uma estrutura recursiva altamente simétrica, o fazem suportar uma variedade de algoritmos paralelos elegantes e eficientes [Leighton 1991].

Basicamente, o *Hyperquicksort* realiza a ordenação da seguinte forma. Cada processo recebe uma lista de números de igual tamanho. A ordenação ocorre em rodadas de ordenação. A cada rodada pares de processos são formados e trocam partes da

sua lista de números entre si de acordo com um número pivô. Ao final de $log_2 n$ rodadas de ordenação (onde n é uma potência de 2 e representa o número total de processos) cada processo tem sua lista ordenada localmente. O maior número contido na lista do processo com menor identificador é menor ou igual ao menor número contido na lista do processo com maior identificador. Além do *Hyperquicksort*, existem outros algoritmos de ordenação baseados no hipercubo [Johnsson 1984, Won and Sahni 1988, Seidel and George 1988, Plaxton 1989]. No entanto, nenhum desses fornece a capacidade de tolerar falhas [Chen and Chung 2001]. Embora os algoritmos definidos nos trabalhos [Chen and Chung 2001, Sheu et al. 1992], tolerem algumas falhas de nodos os mesmos são propostos no âmbito do *BitonicSort* [Johnsson 1984].

O MPI (*Message-Passing Interface*) é o padrão de *facto* para o desenvolvimento de aplicações paralelas e distribuídas. O MPI baseia-se no paradigma de troca de mensagens, onde os processos, ou nodos, acessam uma memória local e estão conectados através de uma rede. O padrão assume que a infraestrutura subjacente é confiável [MPI Forum 2015]. Dessa forma, o MPI não define o comportamento que as implementações devem adotar perante falhas [Bland et al. 2013, Gropp and Lusk 2004]. Consequentemente, as implementações MPI amplamente usadas, como a OpenMPI [Gabriel et al. 2004] e a MPICH [Gropp et al. 1996], abortam toda a aplicação perante a falha de um único processo.

Este trabalho apresenta uma implementação MPI tolerante a falhas do algoritmo paralelo de ordenação *Hyperquicksort*. A implementação é capaz de tolerar até $n - 1$ falhas de processos em tempo de execução. Na implementação, foi utilizada a mais recente especificação de tolerância a falhas em MPI, a *User Level Failure Mitigation* (ULFM) [Bland et al. 2012a] proposta pelo MPI-Fórum. A ULFM oferece um conjunto mínimo de interfaces para recuperar a capacidade do MPI de continuar transportando suas mensagens após uma falha. Uma estratégia de comunicação global das falhas é implementada através da ULFM. A partir de então, uma função de mapeamento é aplicada para permitir que os processos que não falharam assumam as funções dos processos falhos e continuem a computação. Ao final do processo de ordenação as listas inicialmente distribuídas a cada processo estarão ordenadas conforme prevê o *Hyperquicksort*. Resultados experimentais são apresentados para a ordenação de até 1 bilhão de inteiros e confirmam a eficiência da implementação tolerante a falhas.

Este trabalho segue organizado da seguinte forma. A Seção 2 apresenta o modelo de sistema e as definições básicas. A Seção 3 apresenta o padrão MPI e a proposta de tolerância a falhas do MPI-Fórum. O algoritmo *Hyperquicksort* e a sua versão tolerante a falhas é apresentada na Seção 4. A implementação é apresentada na Seção 5. Os resultados experimentais na Seção 6. Por fim, a conclusão é apresentada na Seção 7.

2. Modelo de Sistema e Definições

Os processos se comunicam por operações de envio e recebimento de mensagens através de uma rede. A rede é representada por um grafo completo. Os processos estão organizados em uma topologia de hipercubo virtual. Um hipercubo de d dimensões possui 2^d processos. A Figura 1 apresenta um hipercubo de 3 dimensões. Cada processo i é identificado pelo código binário $(n_0, ..., n_{2^s-1})$ do seu identificador. Dois processos estão conectados se seus endereços diferem em apenas um bit, conforme ilustrado na Figura 1. A comunicação é confiável, garantindo que mensagens trocadas entre dois processos não

são perdidas, corrompidas ou duplicadas. Falhas de particionamento não são tratadas.

O modelo de falhas é o *fail-stop*: um processo falha permanentemente e os demais processos podem detectar a falha [Freiling et al. 2011]. Um processo possui um entre dois estados possíveis: falho ou sem-falha. Um processo que nunca falha é considerado correto ou sem-falha. Falhas são detectadas por um serviço de detecção de falhas perfeito, isto é, nenhum processo é detectado como falho sem estar falho e todo processo falho é detectado por todos os processos corretos em um tempo finito [Chandra and Toueg 1996]. Os processos têm acesso a um diretório compartilhado que é confiável, ou seja, não falha.

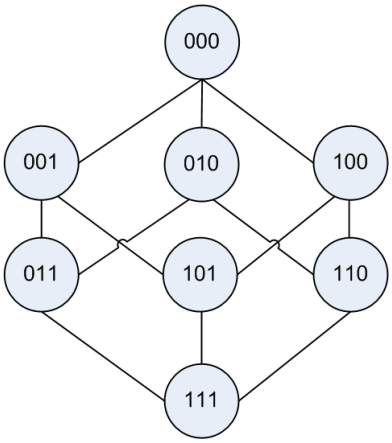

Figura 1. Hipercubo de 3 dimensões.

3. O Padrão MPI

O padrão MPI (*Message Passing Interface*) oferece um dos principais modelos para o desenvolvimento de aplicações paralelas e distribuídas baseado no paradigma de troca de mensagens. O paradigma de troca de mensagens se destina a ambientes computacionais em que os nodos acessam uma memória local e estão conectados através de uma rede - que pode ser tanto um barramento de alta velocidade quanto uma rede local de computadores. Embora o MPI seja baseado no paradigma de troca de mensagens, o padrão também pode ser utilizado em computadores que fazem uso de memória compartilhada.

O MPI consiste de um conjunto de bibliotecas de funções padronizadas pelo MPI-Fórum[1]. Há rotinas para comunicação direta entre dois processos, chamada de comunicação ponto-a-ponto, e rotinas para comunicação coletiva. Além disso, há primitivas para o gerenciamento e criação de processos, entrada e saída de dados em paralelo, gerenciamento de grupos e sincronização de processos e o estabelecimento de topologias virtuais. O MPI-Fórum é a entidade composta por pesquisadores, desenvolvedores e organizações responsáveis por desenvolver e manter a norma MPI, atualmente na sua versão 3.1 [MPI Forum 2015]. Entre as principais implementações da norma MPI se destacam a MPICH [Gropp et al. 1996] e a Open MPI [Gabriel et al. 2004].

Um conceito fundamental em MPI é o comunicador (*communicator*), uma importante estrutura de dados que define o contexto da comunicação e o conjunto de processos pertencentes a esse contexto. No comunicador, os processos são identificados

[1]http://www.mpi-forum.org/

unicamente por meio de um número inteiro positivo chamado *rank*. Há um comunicador pré-definido chamado `MPI_COMM_WORLD` que reúne todos os processos disponíveis no início da execução de uma aplicação ou programa MPI. Um programa MPI pode possuir um ou mais comunicadores.

Uma propriedade fundamental, porém ausente na especificação MPI, é a tolerância a falhas [Gropp and Lusk 2004]. O padrão MPI assume que a infraestrutura subjacente é totalmente confiável [MPI Forum 2015]. Dessa forma, o padrão não define o comportamento preciso que as implementações MPI devem adotar perante falhas [Bland et al. 2013, Gropp and Lusk 2004]. Basicamente, uma falha é tratada como um erro interno da aplicação como, por exemplo, a violação de um espaço de memória. Dessa forma, as falhas de processo ou de rede são repassadas à aplicação simplesmente como se fossem erros de chamadas de funções. Consequentemente, desloca-se a responsabilidade de detectar e de tratar as falhas para as implementações MPI. A norma define os manipuladores de erros (*error handlers*), que são associados ao comunicador MPI, para lidar com os erros do programa.

O manipulador de erros `MPI_ERRORS_ARE_FATAL` é associado por padrão ao comunicador `MPI_COMM_WORLD` e especifica que a manifestação de um erro durante a chamada de uma função MPI leva a todos processos no comunicador a abortar sua execução, encerrando assim toda a aplicação. Por outro lado, o manipulador `MPI_ERRORS_RETURN` retorna um código de erro que indica que uma falha ocorreu [MPI Forum 2015]. Mesmo que um código de erro seja retornado ao programa MPI, o padrão MPI não estabelece mecanismos para lidar com as falhas. Por essa razão, o suporte a tolerância a falhas pela norma MPI é considerado inadequado [Bland et al. 2012b, Bland et al. 2012c]. Além disso, as duas principais implementações MPI citadas acima adotam o manipulador de erro `MPI_ERRORS_ARE_FATAL` por padrão e não dão suporte adequado ao manipulador de erros `MPI_ERRORS_RETURN`, impedindo a continuidade da aplicação no caso de falha.

Diversos trabalhos visam adicionar à implementação MPI rotinas específicas para lidar com as falhas. Entre esses estão o FT-MPI [Fagg and Dongarra 2000], FT/MPI [Batchu et al. 2004], Gropp e Lusk [Gropp and Lusk 2004] e o NR-MPI [Suo et al. 2013]. No entanto, os mesmos não foram adotados pela norma MPI e foram descontinuados. De forma a padronizar e incorporar a tolerância a falhas na norma MPI, o MPI-Fórum criou um grupo de trabalho específico. Os esforços do grupo de trabalho resultaram em duas propostas: a RTS[2] (*Run-through Stabilization Proposal*) [Hursey et al. 2011] e a ULFM[3] (*User-Level Failure Mitigation*) [Bland et al. 2012a, Bland et al. 2013]. A RTS foi a primeira proposta do grupo de tolerância a falhas. Devido à complexidade presente na implementação das primitivas, a proposta RTS não prosseguiu o seu desenvolvimento. A seguir a especificação ULFM é apresentada.

3.1. A Especificação ULFM

A especificação ULFM é o mais recente esforço do MPI-Fórum para padronizar a semântica de tolerância a falhas em MPI[4]. A implementação da ULFM está em desenvolvimento

[2]Disponível em https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/ft/run_through_stabilization

[3]Disponível em http://fault- tolerance.org/ulfm/ulfm-specification/

[4]Disponível em http://meetings.mpi-forum.org/MPI_4.0_main_page.php

como um subprojeto do projeto Open MPI[5] [Gabriel et al. 2004]. Existe a expectativa de que a adotação da ULFM pelo padrão MPI se dê a partir das próximas versões da norma MPI[6].

A ULFM oferece um conjunto mínimo de interfaces para recuperar a capacidade do MPI de continuar transportando suas mensagens após uma falha. Não há uma estratégia de recuperação específica. O objetivo é permitir que o desenvolvedor escolha a técnica de tolerância a falhas que melhor se adeque ao programa. A compatibilidade de código com as versões anteriores do MPI também está entre os requisitos observados.

A aplicação é notificada da falha de um processo ao tentar se comunicar diretamente (comunicação ponto-a-ponto, por exemplo através de um `MPI_Send()` e um `MPI_Recv`) ou indiretamente (por exemplo através de um `MPI_Bcast()`) com o processo falho. A ULFM adota o modelo de falhas *fail-stop* e os manipuladores de erros propostos na norma MPI são os meios para informar a aplicação sobre as falhas de processos. A Figura 2 apresenta um exemplo com três processos (A, B e C) que realizam uma comunicação ponto-a-ponto.

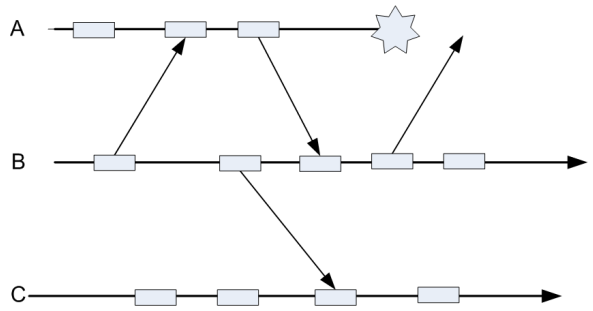

Figura 2. Detecção de falhas ULFM - Processo B detecta a falha em A.

Conforme apresenta a Figura 2, o processo B detecta a falha do processo A após enviar uma mensagem para A. No entanto, o processo C não identifica a falha em A. Essencialmente, a ocorrência de uma falha faz com que a comunicação não seja executada com sucesso. Por razões de desempenho, não há a notificação automática sobre a ocorrência de falhas. Dessa forma, é possível que somente alguns processos a identifiquem. Ao todo, a ULFM disponibiliza ao usuário cinco funções para lidar com as situações de falhas. Entre essas, algumas permitem estabelecer uma visão consistente entre os processos. As primitivas da ULFM são descritas a seguir.

A operação de revogação, `MPI_Comm_revoke`, notifica todos os processos que o comunicador MPI a que pertencem está inválido. Dessa forma, evita a inconsistência entre os processos associados a um comunicador. O comunicador torna-se inválido e as comunicações futuras, ou as comunicações pendentes, são interrompidas e marcadas com um código de erro.

A primitiva `MPI_Comm_agree` é empregada para determinar uma visão consistente entre os processos. Essa função executa uma operação coletiva e faz com que os

[5]http://www.open-mpi.org/faq/?category=ft e http://fault-tolerance.org/

[6]Um rascunho da nova versão, incluindo um capítulo para tolerância a falhas está disponível em https://svn.mpi-forum.org/trac/mpi-forum-web/attachment/ticket/323/mpi31-ticket323-r252-20140518.pdf

processos concordem com um valor lógico. Para fazer uso dessa primitiva o processo que identifica a falha deve antes revogar o comunicador. O construtor `MPI_Comm_shrink` permite à aplicação criar um novo comunicador, eliminando todos os processos falhos de um comunicador inválido. Essa operação é coletiva e executa um algoritmo de consenso para assegurar uma visão consistente no novo comunicador [Herault et al. 2015]. As primitivas `MPI_Comm_failure_ack` e `MPI_Comm_failure_get_acked` são usadas para informar quais processos dentro do comunicador se encontram falhos.

Por exemplo, na Figura 2, o processo B identifica a falha do processo A e então executa a função de revogação para que todos os processos corretos, no caso o processo C, marquem o seu comunicador inválido. Após isso, todos os processos executam a operação de acordo (`MPI_Comm_agree`) para garantir uma visão consistente sobre o estado do comunicador. Então, um novo comunicador válido, somente com processos corretos, pode ser criado por meio da função `MPI_Comm_shrink`. Se houver a necessidade de identificar qual processo falhou (no caso o A), as primitivas `MPI_Comm_failure_ack` e `MPI_Comm_failure_get_acked` podem ser usadas.

As falhas temporárias, tanto de rede quanto de processo, não fazem parte do escopo da ULFM, mas podem ser tratadas em nível de implementação. Uma falha temporária seria promovida a uma falha permanente (conforme o modelo *fail-stop*). Ou seja, se um processo sem-falha detecta que um processo deixa de responder, mesmo que temporariamente, o processo correto classifica esse processo como falho e continuamente ignora e descarta qualquer comunicação com o processo falho. Nesse caso, como dito anteriormente, para evitar que os processos tenham uma visão diferente sobre o estado de algum processo, as rotinas da ULFM (`MPI_Comm_revoke`, `MPI_Comm_agree` e `MPI_Comm_shrink`) podem ser usadas.

4. Algoritmo *Hyperquicksort*

Os algoritmos paralelos de ordenação são utilizados, por exemplo, em processamento de imagens, geometria computacional e teoria dos grafos [Quinn 2003]. Proposto por [Wagar 1987] o algoritmo *Hyperquicksort* combina a topologia e as características do hipercubo com a estratégia de ordenação empregada no *Quicksort* [Hoare 1962]. O algoritmo *Quicksort* é um dos algoritmos de ordenação sequenciais mais rápidos. É intuitivamente recursivo e baseia-se na comparação de chaves. Seu tempo de execução está estimado em $O(n \log n)$ no melhor e no médio caso e n^2 para o pior caso [Cormer et al. 2009].

O problema da ordenação no hipercubo consiste em empregar um conjunto P de processos, $P = \{p_0, p_1, ..., p_{2^{dim}-1}\}$, para ordenar uma lista K de números, $K = \{a_0, a_1, ..., a_{k-1}\}$. Os processos são organizados na forma de um hipercubo virtual de dimensão dim. Inicialmente os $|K|$ números são divididos igualmente entre os $|P|$ processos. Cada processo é responsável por ordenar uma lista de $\frac{|K|}{|P|}$ números. A ordenação é executada em rodadas, na quais os processos p_i e p_j trocam entre si parte da sua lista, determinada com base em um *número pivô* distribuído por um dos processos. Ao final de dim, rodadas de ordenação as listas estão ordenadas de forma que em cada processo p_i o maior número é menor ou igual ao menor número no processo $p_i + 1$, onde $0 \leq i \leq |P| - 2$. O algoritmo 1 a seguir apresenta o pseudocódigo do *Hyperquicksort*.

Inicialmente, cada processo ordena localmente a sua lista (linha 8). Os processos são organizados em *clusters* virtuais de tamanho regressivo (potência de 2) a cada rodada

Algorithm 1 Pseudocódigo do algoritmo *Hyperquicksort*

1: ***Hyperquicksort*** (para cada processo p executando em paralelo)
2: Initialization
3: $dim \leftarrow \log_2 |P|$ {*Dimensão do hipercubo*}
4: $rank \leftarrow process\_id$ {*Cada processo tem um valor único entre* $0..2^{dim}-1$}
5: $lista \leftarrow K$ {*lista de números inicial em cada processo*}
6: $n \leftarrow |K|$ {*tamanho da lista de números em cada processo*}
7: Begin
8: quicksort($lista, n$)
9: **while** $dim > 0$ **do**
10: $cluster_i \leftarrow$ processes($rank, dim$)
11: $processo\_raiz \leftarrow$ root($rank, dim$)
12: **if** $rank == processo\_raiz$ **then**
13: $pivo \leftarrow$ medium($lista$)
14: broadcast($processo\_raiz, pivo, cluster_i$)
15: create_lists($higher\_list, lower\_list, list, pivo$)
16: $partner \leftarrow rank \oplus 2^{(dim-1)}$ {*ou exclusivo*}
17: **if** $rank > partner$ **then**
18: send($lower\_list, partner$)
19: receive($new\_higher_list, partner$
20: $list \leftarrow$ merge($higher\_list, new\_higher\_list$)
21: **else if** $rank < parter$ **then**
22: send($higher\_list, partner$)
23: receive($new\_lower\_list, partner$
24: $list \leftarrow$ merge($lower\_list, new\_lower\_list$)
25: $dim \leftarrow dim - 1$
26: quicksort($lista, n$)
End

de ordenação (linha 10). A Figura 3 representa o tamanho dos *clusters* para um hipercubo de 3 dimensões. Inicialmente, na primeira rodada de ordenação há oito processos agrupados em um único *clusters*. Para a segunda rodada, há dois *clusters* que agrupam quatro processos cada. Por último, há quatro *clusters* para cada dois processos. A partir de então, durante dim rodadas de ordenação o algoritmo executa os seguintes passos.

Os *clusters* são formados na respectiva rodada de ordenação (linha 10). O processo com menor *rank* em cada *cluster* é definido como o *processo raiz* (linha 11). Esse processo é o responsável por distribuir um número pivô aos demais processos do seu *cluster* (linhas 12-14). O número pivô é o número médio obtido a partir da sua lista de números (linha 13). Esse número pivô tem a mesma função da chave empregada no algoritmo *Quicksort*. Após receber o número pivô, cada processo divide a sua lista em duas outras listas: uma lista com números maiores que o número pivô e outra lista com os números menores que o número pivô (linha 15). Então, cada processo encontra um parceiro no seu *cluster* usando uma operação de ou exclusivo aplicada no seu próprio *rank* (linha 16). O processo de maior *rank* envia a sua lista de números menores que o número pivô ao seu parceiro (que possui *rank* maior) e recebe deste a lista com números maiores

que o número pivô (linhas 17-24). Após a troca, cada processo une a lista recebida com a lista que não foi trocada (linha 20 e 24) e realiza a ordenação nessa nova lista (linha 26).

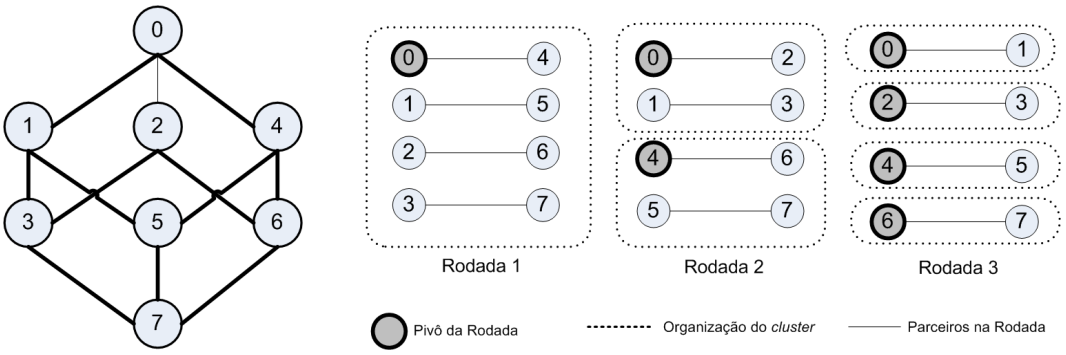

Figura 3. Algoritmo paralelo *Hyperquicksort* com 8 processos

A Figura 3 apresenta um exemplo de execução do algoritmo *Hyperquicksort* para 8 processos. A ordenação é realizada em 3 rodadas. Na primeira rodada, há um *cluster* com 8 processos e o processo raiz é o 0. O processo 0 distribui o seu número pivô aos demais processos do seu *cluster*. Os seguintes pares de processos são estabelecidos e trocam as suas listas de acordo com o número pivô recebido do processo 0: $(0, 4)$, $(1, 5)$, $(2, 6)$ e $(3, 7)$. Todos os processos reorganizam as suas listas e as ordenam localmente. Na segunda rodada de ordenação há dois *clusters* cada um com 4 processos. Os processos 0 e 4 são os processos raízes de seus respectivos *clusters*. Cada processo raiz envia o seu número pivô aos outros processos do seu *cluster*. Então as listas são trocadas entre os processos pares na rodada 2. Finalmente, na terceira rodada, todo o processo é repetido considerado os *clusters* e os processos em cada *cluster* de acordo com a terceira rodada de ordenação. A seguir, descreve-se a implementação tolerante a falhas do algoritmo *Hyperquicksort*.

4.1. *Hyperquicksort* Tolerante a Falhas

No início de cada rodada de ordenação, cada processo possui uma lista com os processos falhos. A partir de então, uma função de mapeamento é invocada em cada processo. Tal função auxilia a formar os pares de processos em uma rodada de ordenação e a definir qual processo sem-falha assume as tarefas do processo falho. Um processo p_i pode assumir até $n - 1$ processo falhos (no caso de somente um processo permanecer sem-falha). O processo p_i, além de executar normalmente as suas funções no algoritmo, deve também executar as funções que seriam executadas pelo processo falho. Cada processo p_j também precisa conhecer o processo p_i que assume as funções do processo falho. Para essas duas importantes tarefas da versão tolerante a falhas, a função $c_{i,s}$, definida no algoritmo *Hi-ADSD* [Duarte, Jr. and Nanya 1998], é utilizada para auxiliar no mapeamento. A função $c_{i,s}$ é descrita a seguir:

$$c_{i,s} = (i \oplus 2^{s-1}, c_{i\oplus 2^{s-1},1}, ..., c_{i\oplus 2^{s-1},s-1})$$

A função $c_{i,s}$ considera que os processos estão organizados logicamente em um hipercubo: i representa o processo p_i e s está relacionado a uma determinada rodada de ordenação. Inicialmente $s = dim$. O símbolo $\oplus$ representa a operação binária de OU exclusivo (XOR). A Tabela 1 apresenta um exemplo da função $c_{i,s}$ aplicada a um

Tabela 1. $c_{i,s}$ para um sistema com 8 nodos.

s	$c_{0,s}$	$c_{1,s}$	$c_{2,s}$	$c_{3,s}$	$c_{4,s}$	$c_{5,s}$	$c_{6,s}$	$c_{7,s}$
1	1	0	3	2	5	4	7	6
2	2 3	3 2	0 1	1 0	6 7	7 6	4 5	5 4
3	4 5 6 7	5 4 7 6	6 7 4 5	7 6 5 4	0 1 2 3	1 0 3 2	2 3 0 1	3 2 1 0

hipercubo de dimensão 3, ou seja, com 8 processos. O processo p_0 quando $s = 3$ tem o seguinte resultado: 4, 5, 6 e 7.

Os pares de processos são formados de acordo com o Algoritmo 2. Conforme apresenta o algoritmo, o parceiro de um processo p_i na rodada de ordenação s é o primeiro processo sem-falha na $c_{i,s}$. Portanto, para a primeira rodada de ordenação o processo p_0 deve trocar a sua lista com o primeiro processo sem-falha resultante de $c_{0,3}$. Se p_4 está sem-falha então p_0 e p_4 formam um par e trocam suas lista de acordo com as linhas 17-24 do Algoritmo 1 (considerando que p_0 também está sem-falha). Se p_4 se encontra falho então p_0 deve trocar a sua lista com p_5 e assim por diante. Se tanto p_5 e p_6 estiverem falhos então p_0 não troca a sua lista com nenhum processo nessa rodada de ordenação.

Algorithm 2 Função para encontrar um parceiro no cluster dim

```
1: function parceiro(rank, rodada)
2: Begin
3:     nodes ← c_{rank,rodada}
4:     j ← 0
5:     while j ≤ size(nodes) do
6:         if nodes[j] ∉ faults then
7:             return nodes[j]
8:         j ← j + 1
9:     return ⊥
   End
```

O processo que substitui um processo falho p_i em uma rodada de ordenação é escolhido de acordo com o Algoritmo 3. O substituto de um processo será o primeiro processo sem-falha de $c_{i,s}$, onde inicialmente $s = 1$ e i é o identificador do processo falho. Se não há processo sem-falha naquele *cluster*, s é incrementado até que um processo sem-falha seja encontrado. Considerando a primeira rodada de ordenação e os processos 0 e 4, se o processo p_4 está falho então p_5 assume as funções de p_4 na respectiva rodada de ordenação ($c_{4,1} = 5$ ver Tabela 1). Se p_5 também está falho então p_6 substitui p_4 pois é o primeiro processo sem-falha em $c_{4,2}$.

Um outro exemplo é fornecido a seguir. Suponha que o processo 2 está falho (Figura 3) na primeira rodada de ordenação. Essa rodada corresponde ao maior *cluster* ($s = 3$). Nessa rodada, os processos 2 e 6 forma um par quando ambos estão sem-falha. Entretanto, o primeiro processo sem-falha em $c_{6,3}$ é o processo 3, isto é, $c_{6,3} = 3$. O processo 3 é responsável pelo processo 2 pois é o primeiro nodo sem-falha de $c_{2,1}$. O processo 3 então se torna responsável pelas tarefas do processo 2. O processo 3 então realiza a leitura da lista de números do processo 2 e interage com o processo 6 substituindo o processo 2. Vale lembrar que o processo 3 também realiza normalmente a sua tarefa

Algorithm 3 Função para encontrar um parceiro no cluster dim

```
 1: function substituiFalho(rankFalho, dim)
 2: Begin
 3:     s ← 1
 4:     while s ≤ dim do
 5:         j ← 0
 6:         nodes ← c_{rankFalho,s}
 7:         while j ≤ size(nodes) do
 8:             if nodes[j] ∉ faults then
 9:                 return nodes[j]
10:             j ← j + 1
11:         s ← s + 1
12:     return ⊥
    End
```

com o processo 7, pois ambos estão sem-falha na rodada. Ao fim de cada rodada de ordenação cada processo salva a sua lista ordenada (linha 26 do Algoritmo 1) em disco compartilhado.

5. Implementação

A implementação foi realizada através das primitivas fornecidas na especificação ULFM. Vale lembrar que, por padrão, a detecção de falhas na ULFM é local, isto é, somente os processos que efetivamente se comunicam com o processo que falhou detectam a falha. No entanto, a ULFM permite, através das suas primitivas, implementar uma abordagem global de detecção de falhas, descrita a seguir.

Uma função chamada `detectaFalhos()` para detectar todos os processos que falharam foi implementada e inserida no começo de cada rodada de ordenação. Essa função inicia invocando a primitiva `MPI_Barrier` a fim de criar um ponto de sincronização entre os processos e verificar se houve falhas. Se ao menos um processo estiver falho, a função `MPI_Barrier` retorna um código de erro: `MPI_ERR_PROC_FAILED` ou `MPI_ERR_REVOKED`. A partir de então, todos os processos chamam uma função de acordo (`MPI_Comm_agree()`). A função `MPI_Comm_agree` executa uma operação coletiva entre os processos corretos no comunicador. No caso de falha, essa função notifica os processos que o comunicador está inválido. Na sequência o comunicador MPI é revogado usando a primitiva `MPI_Comm_revoke()`.

A partir de então são empregadas as rotinas `MPI_Comm_failure_ack()` e `MPI_Comm_failure_get_acked()` para identificar quais processos dentro do comunicador estão falhos. Após isso, a rotina `MPI_Comm_shrink()` cria um novo comunicador, eliminando todos os processos que falharam. Operações de grupo de processos em MPI são ainda realizadas para manter os processos no novo comunicador com o mesmo *rank* que tinham antes da falha.

Um vetor com o estado (falho ou sem-falha) dos processos é mantido em cada processo usando a função `detectaFalhos()`. Uma vez que cada processo possui a lista de processos falhos, as funções apresentadas nos Algoritmos 2 e 3 são utilizadas para permitir que os processos sem-falha continuem a execução.

Em relação as listas de números de posse dos processos falhos duas abordagens foram implementadas: 1) os processos sem-falha mantém a lista em nome de cada processo falho e; 2) os processos sem-falha incorporam a lista do processo falho à sua lista. Na primeira possibilidade se há 8 processos, então ao final haverá 8 listas de números, mesmo se $n-1$ processos falharem. Essa versão permite - caso um processo pudesse retornar após uma falha -, que um processo que deixou de participar de uma rodada de ordenação específica retorne em uma rodada posterior. Por exemplo, o processo 0 participa somente da primeira e da última rodada de ordenação, supondo 3 rodadas de ordenação. Na segunda possibilidade, o número de listas será igual ao número de processos que não falharam. Os resultados a seguir apresentam o desempenho considerando a primeira abordagem.

Uma função `injetaFalhas()` foi codificada para injetar falhas durante a execução. Cada processo recebe o número total de processos que devem falhar e o número total de rodadas de ordenação. A partir de então, a função aleatoriamente define em qual rodada um determinado processo deve falhar. Um processo pode vir a se tornar falho em qualquer rodada de ordenação. Um processo finaliza a si mesmo através do sinal `SIGKILL` se seu identificador havia sido sorteado para falhar naquela rodada. Uma vez que os processos executam a função `injetaFalhas()` em paralelo a mesma semente é utilizada na função `srand()` em cada processo.

6. Resultados

Os experimentos foram executados no sistema operacional *Linux Kernel* 3.2.0 em 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%. O código MPI foi escrito em liguagem C. A biblioteca MPI utilizada foi a Open MPI versão 1.7 estendida com a ULFM (1.7ft_b4)[7].

São apresentados resultados de desempenho para quatro cenários: 1) sem-falhas; 2) somente um processo falho; 3) metade dos processos falham; 4) $n-1$ processos falham. O objetivo não é apresentar o *speedup*, mas a capacidade do algoritmo de se manter em execução mesmo perante falhas. As falhas são inseridas no início de cada rodada de ordenação.

A Figura 4 apresenta o desempenho do *Hyperquicksort* tolerante a falhas para ordenar 1 bilhão de números inteiros usando 16 processos MPI e aplicando os quatro cenários de falhas. O tempo de execução do algoritmo sem falhas é de aproximadamente 420 segundos e a variação no desempenho é pequena.

No cenário com uma falha, o tempo de execução do algoritmo é ligeiramente menor. Isso se deve ao momento em que a falha ocorre. Uma falha que acontece logo na primeira rodada de ordenação prejudica mais o desempenho do algoritmo do que uma falha que ocorre na última rodada de ordenação. Uma falha na primeira rodada obriga o algoritmo a se reconfigurar logo no começo, fazendo um processo acumular tarefas desde o início. Por outro lado, uma falha na última rodada de ordenação faz com o processo que substitui o processo falho economize uma troca de listas: o processo substituto é o processo parceiro do processo falho. Por exemplo, supondo que na última rodada de

[7]http://fault-tolerance.org/ulfm/downloads/

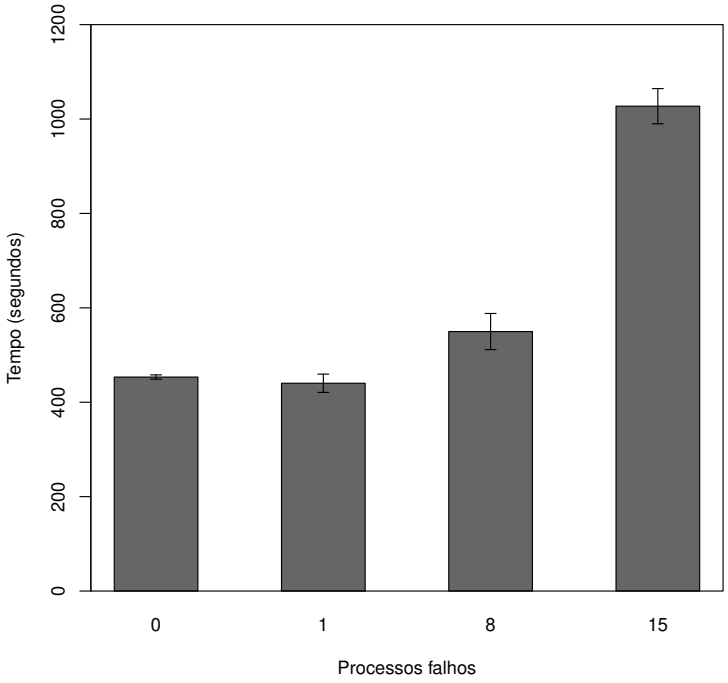

Figura 4. Desempenho de 16 processos perante falhas ordenando 1.024 x 10^6 números inteiros.

ordenação o processo 1 falhe (Figura 3, rodada 3). O processo 0 é o processo que substitui o processo falho, mas o processo 0 também é o parceiro do processo falho. Dessa forma, o processo substituto lê a lista de números do processo falho e executa as operações previstas sem usar as primitivas `MPI_Send()` e `MPI_Recv`. Outra situação que pode aumentar o desempenho é a falha do processo raiz. Lembrando que o processo raiz é o responsável por distribuir o seu número pivô os demais processos do seu *cluster*.

Para o cenário com metade das falhas, o desempenho é prejudicado em cerca de 30%. Para esse cenário, houve casos em que apenas um processo falhou na primeira rodada de ordenação, dois processos falharam na segunda rodada de ordenação, três processos falharam na terceira rodada de ordenação e dois processos falharam na última rodada de ordenação. O cenário com $n - 1$ falhas apresenta perto de 150% de sobrecarga. No entanto, para todos os casos, a ordenação foi realizada com sucesso, apesar das falhas.

7. Conclusão

Este trabalho apresentou uma implementação MPI do algoritmo de ordenação paralela *Hyperquicksort* que tolera até $n - 1$ falhas. A implementação empregou a especificação ULFM para detectar uma falha local. A partir de então uma função foi desenvolvida para permitir que todos os processos que participam da computação conheçam a falha. Uma função de mapeamento dos processos foi implementada para permitir que os processos sem-falha substituam os processos falhos. O algoritmo tolerante a falhas foi executado para ordenar 1 bilhão de números inteiros. Resultados apresentam que o algoritmo foi capaz de continuar a sua execução apesar das falhas de processos.

Na avaliação realizada, as falhas foram inseridas no início de uma rodada de ordenação. Um simples extensão no algoritmo o permitiria lidar com uma falha em qualquer momento da ordenação: nesse caso os processos que não falharam reiniciariam a ordenação a partir da rodada anterior.

Tradicionalmente, as implementações MPI abortam toda a sua execução mesmo se um único processo falhar. A ULFM delega ao programador da aplicação a tarefa de lidar com as falhas. Entre essas tarefas está escolher a estratégia de recuperação que melhor se adapta a sua aplicação. Um trabalho futuro é investigar se a mesma estratégia de recuperação empregada no *Hyperquicksort* pode ser aplicada em outros algoritmos de ordenação baseados no hipercubo. Além disso, é possível investigar se outras classes de aplicações MPI, para além dos algoritmos de ordenação, podem adotar a mesma abordagem de recuperação aplicada ao *Hyperquicksort*. Uma API de programação pode ser projetada para permitir que o programador insira de forma transparente a estratégia de recuperação nas aplicações MPI.

Referências

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.

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 HPC 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.

Bland, W., Du, P., Bouteiller, A., Hérault, T., Bosilca, G., and Dongarra, J. (2012c). A checkpoint-on-failure protocol for algorithm-based recovery in standard MPI. In *Euro-Par*.

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

Chen and Chung (2001). Improved fault-tolerant sorting algorithm in hypercubes. *TCS: Theoretical Computer Science*, 255.

Cormer, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introductions to Algorithms'*. The MIT Press.

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

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

Freiling, F. C., Guerraoui, R., and Kuznetsov, P. (2011). The failure detector abstraction. *ACM Comput. Surv.*, 43(2):9:1–9:40.

Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. (2004). Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary.

Gropp, W., Lusk, E., Doss, N., and Skjellum, A. (1996). A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828.

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

Herault, T., Bouteiller, A., Bosilca, G., Gamell, M., Teranishi, K., Parashar, M., and Dongarra, J. (2015). Practical scalable consensus for pseudo-synchronous distributed systems. In *SuperComputing conference*.

Hoare, C. A. R. (1962). Quicksort. *The Computer Journal*, 5(4):10–15.

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*.

Johnsson (1984). Combining parallel and sequential sorting on a boolean n-cube. In *ICPP: 13th International Conference on Parallel Processing*.

Leighton, F. T. (1991). *Introduction to Parallel Algorithms and Architectures: Array, Trees, Hypercubes*. Morgan Kaufmann Publishers.

MPI Forum (2015). Document for a standard message-passing interface 3.1. Technical report, University of Tennessee, http://www.mpi-forum.org/docs/mpi-3.1.

Parhami, B. (1999). *Introduction to Parallel Processing: Algorithms and Architectures*. Kluwer Academic Publishers, Norwell, MA, USA.

Plaxton, C. G. (1989). Load balancing, selection sorting on the hypercube. In *SPAA*, pages 64–73.

Quinn, M. J. M. J. (2003). *Parallel programming in C with MPI and OpenMP*. McGraw-Hill, pub-MCGRAW-HILL:adr.

Seidel, S. R. and George, W. L. (1988). Binsorting on hypercubes with d-port communication. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications - Volume 2*, C3P, pages 1455–1461, New York, NY, USA. ACM.

Sheu, Chen, and Chang (1992). Fault-tolerant sorting algorithm on hypercube multicomputers. In *ICPP: 21th International Conference on Parallel Processing*.

Suo, G., Lu, Y., Liao, X., Xie, M., and Cao, H. (2013). Nr-mpi: A non-stop and fault resilient mpi. In *ICPADS*.

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

Won, Y. and Sahni, S. (1988). A balanced bin sort for hypercube multicomputers. *The Journal of Supercomputing*, 2(4).