Uma Implementação MPI Tolerante a Falhas do Algoritmo Paralelo de Ordenação Quickmerge

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

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

[2]Universidade Federal do Paraná (UFPR) – Departamento de Informática
Caixa Postal 19018 – 81531-980 – Curitiba – PR – Brasil

felipecamargoxav@hotmail.com, edson@utfpr.edu.br, elias@inf.ufpr.br

Abstract. *Quickmerge is a parallel sorting algorithm that combines the efficient Quicksort strategy with merge operations of subsets which are created from key elements called pivots. Two versions of the Quickmerge algorithm that run on a hypercube have been found in the literature, but none can deal with process failures at runtime. In this work we present a fault-tolerant MPI implementation of both Quickmerge and Modified Quickmerge based on a virtual topology called VCube. The proposed algorithms work correctly even if all processes fail but one remains correct. The algorithms are compared to a fault-tolerant implementation of the Hyperquicksort parallel sorting algorithm also for the VCube. Results show the efficiency of the implementation to sort up to 1 billion integers in faulty and fault-free scenarios.*

Resumo. *O algoritmo de ordenação paralelo Quickmerge combina a estratégia do algoritmo Quicksort com operações de fusão de subconjuntos criados a partir de elementos chaves, chamados pivôs. Duas versões do algoritmo Quickmerge que executam sobre o hipercubo foram encontradas na literatura, porém nenhuma considera falhas de processos. Este trabalho apresenta uma implementação MPI tolerante a falhas dos algoritmos Quickmerge e Quickmerge Modificado na topologia virtual denominada VCube. Os algoritmos propostos são capazes de executar a ordenação mesmo que todos menos um processo falhem. Os algoritmos são comparados a uma implementação tolerante a falhas do algoritmo paralelo Hyperquicksort. Resultados mostram a eficiência da implementação na ordenação de até 1 bilhão de números inteiros em cenários com e sem falhas.*

1. Introdução

O algoritmo *Quickmerge* [Quinn 1988] é um algoritmo paralelo de ordenação que foi projetado para arquiteturas baseadas em multiprocessadores fortemente acoplados. O algoritmo é inspirado na estratégia de ordenação eficiente do *Quicksort*. A ideia original do algoritmo prevê a sua execução em três fases. Inicialmente há n elementos para serem ordenados. Na primeira fase, cada um de p processos ordena um conjunto contíguo de não mais que $\lceil n/p \rceil$ elementos usando o próprio algoritmo Quicksort sequencial. Ao final dessa fase os n elementos iniciais formam p listas ordenadas de tamanho aproximado $\lceil n/p \rceil$. Da primeira lista são escolhidos $p - 1$ elementos que são usados como pivôs.

Por exemplo, supondo que haja 3 processos, são criadas três listas e dois pivôs (i_1, i_2). Na segunda fase, ocorre a formação de novas p listas a partir dos elementos pivôs. Os elementos e de cada lista, onde $e \leq i_1$, são unidos em uma nova lista. Da mesma forma, com $i_1 < e \leq i_2$ e com $e > i_2$. As novas p listas são novamente ordenadas localmente. A terceira fase une as p listas que agora formam uma única lista ordenada.

Uma versão do algoritmo *Quickmerge*, proposta para a topologia de hipercubo [Quinn 1989], assemelha-se a outro algoritmo baseado no *Quicksort*, o Hyperquicksort [Wagar 1987]. O hipercubo é uma estrutura largamente utilizada como topologia de interligação e comunicação e para a execução de algoritmos paralelos e distribuídos [Parhami 1999]. A grande diferença entre esses algoritmos está em como os elementos pivôs são escolhidos e comunicados. O *Quickmerge* exige menos comunicação uma vez que cada processo recebe seus pivôs somente no início da ordenação. O *Hyperquicksort* exige que o pivô seja selecionado e transmitido durante a ordenação. No entanto, o *Quickmerge* pode gerar uma distribuição desigual de elementos entre os processos.

Em [Quinn 1989], com o objetivo de contornar a distribuição desigual de carga, uma versão modificada do algoritmo *Quickmerge* foi proposta. Ambas as versões foram comparadas com algoritmo *Hyperquicksort*. Em seus experimentos, Quinn concluiu o *Hyperquicksort* supera levemente o *Quickmerge* em termos de tempo de execução (desempenho). Os experimentos foram realizados com 16.384 números inteiros considerando uma arquitetura de 64 processadores e a tolerância a falhas não é considerada. Apesar de algoritmo *Quickmerge* possuir diversas variantes [Evans 1990, Shi and Schaeffer 1992, Träff 2018], não se encontrou na literatura qualquer implementação que considera falhas.

Este trabalho apresenta uma implementação MPI tolerante a falhas do algoritmo paralelo de ordenação *Quickmerge* e *Quickmerge Modificado*. O algoritmo é baseado em uma topologia virtual denominada VCube [Duarte et al. 2014], que é idêntica ao hipercubo quando todos os processos estão sem falhas e mantém diversas propriedades logarítmicas quando há processos falhos. Os algoritmos são capazes de tolerar até $p - 1$ falhas de processos em tempo de execução, de um total de p processos. Na implementação, utiliza-se a mais recente especificação de tolerância a falhas em MPI, a *User Level Failure Mitigation* (ULFM) [Bland et al. 2013] proposta pelo MPI-Fórum. A ULFM permite que o próprio desenvolvedor da aplicação desenvolva a estratégia de tolerância a falhas mais adequada para sua aplicação. Resultados experimentais apresentam o desempenho em termos do balanceamento de carga entre processos e o tempo de execução dos algoritmos para ordenar 1 bilhão de números inteiros em até 32 processos em cenários com e sem falhas de processos. Os algoritmos são comparados com a versão tolerante a falhas do algoritmo *Hyperquicksort* [Camargo and Duarte, Jr. 2016].

Este trabalho segue organizado da seguinte forma. A Seção 2 descreve o modelo de sistema e definições básicas. A Seção 3 apresenta brevemente o padrão MPI e a ULFM. Os algoritmos citados e a versão tolerante a falhas do *Quickmerge* são apresentados na Seção 4. A implementação e os resultados experimentais são descritos na Seção 5. A conclusão é apresentada na Seção 6.

2. Modelo de Sistema e Definições

Este trabalho assume o modelo de falhas *fail-stop*: um processo falha por parada permanente e os demais processos detectam a falha [Freiling et al. 2011]. Um processo possui

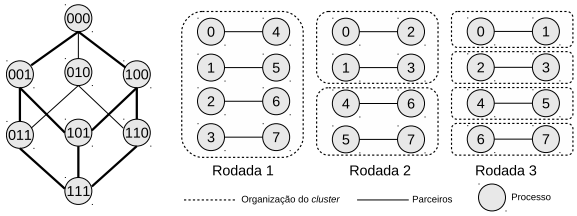


Figura 1. Hipercubo de 3 dimensões.

um entre dois estados possíveis: falho ou sem-falha. Um processo que nunca falha é considerado correto ou sem-falha. O sistema inclui um serviço de detecção de falhas perfeito, isto é, não há enganos quanto ao estado dos processos. Os processos têm acesso a armazenamento compartilhado confiável e se comunicam transmitindo e recebendo mensagens através de canais de comunicação confiáveis, ou seja, mensagens trocadas entre dois processos não são perdidas, corrompidas ou duplicadas. O sistema é representável por um grafo completo, isto é, cada processo pode comunicar diretamente com qualquer outro sem utilizar intermediários. Os processos estão organizados em uma topologia de hipercubo virtual denominada VCube [Duarte et al. 2014]. Quando todos os processos estão sem falhas o VCube corresponde a um hipercubo. Um hipercubo de dim dimensões possui 2^{dim} processos. A Figura 1 apresenta um hipercubo de 3 dimensões com identificadores dos nodos em números *bits* (esquerda). Cada processo p_i é identificado pelo código binário $(p_0, ..., p_{2^{dim}-1})$ de i. Dois processos estão conectados se seus endereços diferem em apenas um *bit*.

Nos algoritmos apresentados, a ordenação ocorre em dim rodadas de ordenação. A cada rodada, os processos são organizados em *clusters* de tamanhos progressivamente menores (iguais a potências de 2). Pares de processos são formados para trocar elementos com base em um pivô. A Figura 1 (direita) apresenta um exemplo com 8 processos e portanto três rodadas. Na primeira rodada há um único *cluster* e os seguintes pares de processos são estabelecidos: $(0, 4)$, $(1, 5)$, $(2, 6)$ e $(3, 7)$. Na segunda rodada, há dois *clusters* cada um com 4 processos e novos pares de processos são formados e um novo número pivô é escolhido. Finalmente, na terceira rodada, há dois processos em cada *cluster* e todo o processo é repetido. Os algoritmos *Quickmerge*, *Quickmerge Modificado* e *Hyperquicksort* diferem basicamente na forma como escolhem o número pivô. A Seção 4 detalha os algoritmos.

3. O Padrão MPI e a Especificação ULFM

O padrão MPI (*Message-Passing Interface*), mantido pelo MPI-Fórum [Forum 2019], define interfaces para troca de mensagens em um modelo de computação paralela e distribuída onde geralmente cada processo tem acesso somente a sua memória local [Forum 2015]. Nesse modelo, dados são comunicados entre processos através de operações entre dois processos como `MPI_Send()` e `MPI_Receive()`, ou coletivas, como `MPI_Bcast()`. O MPI conta ainda com operações de criação dinâmica de processos, gerência de grupos e operações paralelas de entrada e saída (I/O).

Entre outras premissas, o MPI assume a transmissão confiável de mensagens. Ou seja, uma mensagem enviada é sempre recebida corretamente e o usuário não precisa verificar erros de transmissão, *timeouts*, ou qualquer outra condição de erro. Na

verdade, o MPI original não oferece mecanismos para lidar com falhas, que devem ser tratadas pelo sistema subjacente. No entanto, a probabilidade de falhas de processos não pode ser desprezada, especialmente considerando que o tempo médio entre falhas diminui conforme aumenta o tamanho dos sistemas de computação de alto desempenho [Gamell et al. 2017]. A *User Level Failure Mitigation* (ULFM) [Bland et al. 2013] é a mais recente proposta do MPI-Fórum para lidar com as falhas da aplicação.

Na ULFM o próprio desenvolvedor da aplicação pode lidar com falhas de processos ou de comunicação e assim definir o comportamento da aplicação após uma falha. O desenvolvedor pode decidir a estratégia de tolerância a falhas mais adequada para sua aplicação, que pode variar, por exemplo, desde o lançamento de novos processos para assumir as tarefas de um processo que falhou ou simplesmente a delegação de tarefas de um processo falho a outro processo sem-falha [Camargo and Duarte, Jr. 2018].

A ULFM atua sobre o comunicador MPI, um elemento fundamental que reúne todos os processos participantes da computação e viabiliza a comunicação entre eles. Ao todo, a ULFM apresenta três constantes e cinco primitivas, descritas a seguir:

- `MPIX_ERR_PROC_FAILED` se ao usar uma das primitivas de comunicação do MPI não foi possível completar a troca de mensagens, uma indicação de erro `MPIX_ERR_PROC_FAILED` é retornada.
- `MPIX_ERR_PROC_FAILED_PENDING` quando um potencial emissor se comunica com um receptor não-bloqueante que aguarda uma mensagem. Uma recepção não bloqueante define que o receptor continua sua execução mesmo se ainda não recebeu a mensagem.
- `MPIX_ERR_REVOKED` quando um dos processos da aplicação invocou a primitiva `MPI_Comm_revoke` no comunicador MPI.
- `MPIX_Comm_revoke(MPI_Comm comm)` interrompe qualquer comunicação que esteja ocorrendo dentro do comunicador `comm`.
- `MPIX_Comm_shrink(MPI_Comm comm, MPI_Comm* newcomm)` cria um novo comunicador `newcomm` sem os processos falhos de `comm`.
- `MPIX_Comm_agree(MPI_Comm comm, int *flag)` executa uma operação de consenso com todos os processos corretos. Os processos devem concordar com o valor contido em `flag` para que uma única visão sobre os processos que falharam seja estabelecida.
- `MPIX_Comm_failure_get_acked(MPI_Comm, MPI_Group*)` identifica os processos que falharam dentro do grupo.
- `MPIX_Comm_failure_ack(MPI_Comm)` permite que a aplicação identifique que ocorreu uma falha.

São diversos os trabalhos que adotam a ULFM para lidar com falhas em aplicações MPI. Em [Shahzad et al. 2019] os autores implementam uma biblioteca chamada CRAFT para realizar *checkpoint-restart* em nível de aplicação através da ULFM. O trabalho em [Ashraf et al. 2018] lida com falhas de processos em tempo de execução sem descartar os dados de processos após a ocorrência de uma falha. Os autores exploram estratégias de recuperação que usam a ULFM aliada ao *checkpointing* em memória. A aplicação continua sua execução após uma falha usando tanto os processos que sobreviveram quanto novos processos para substituir os falhos. O trabalho de Bland et. al. descreve as lições aprendidas sobre a incorporação da ULFM à biblioteca MPICH [mpich.org 2017]. Bland

et. al. demostram que, embora a implementação da ULFM realizada na biblioteca MPICH não seja otimizada, o custo do tempo de execução das novas chamadas de API introduzidas é relativamente baixo. O trabalho em [Camargo and Duarte, Jr. 2017] apresenta mais detalhes sobre a implementação de tolerância a falhas em MPI usando a ULFM.

4. Algoritmos Paralelos Baseados no *Quicksort* e no Hipercubo

Na versão do ***Quickmerge*** para o hipercubo proposta por Quinn [Quinn 1989], $x * 2^{dim}$ elementos estão igualmente distribuídos entre os 2^{dim} processos de um hipercubo de dimensão dim, sendo x a quantidade de elementos de cada processo. Vale destacar que neste trabalho usamos o termo "processo", mas Quinn se referia a "processador". A lista local mantida por um processo é denominada $a[]$. O Algoritmo 1 inicia com cada processo ordenando localmente sua lista de elementos usando o algoritmo sequencial *Quicksort* (linha 5). A particularidade do *Quickmerge* está nas linhas de 7 a 10: o processo p_0 gera uma lista de elementos, chamados por Quinn de *splitters* e neste trabalho de pivôs, que servem de base para a ordenação (linha 9). São escolhidos $2^{dim} - 1$ elementos que dividem uma lista em 2^{dim} sublistas de tamanhos iguais. Por exemplo, supondo um hipercubo de 3 dimensões e que o p_i possui uma lista $a[]$ com 1024 elementos, o vetor $splitter[]$ conterá 7 elementos referentes aos índices 128, 256, 384, 512, 640, 768 e 896 de $a[]$. Importante observar que os elementos do vetor $splitter[]$ estão em ordem crescente. O vetor $splitter[]$ é então disseminado aos demais processos (linha 10).

Algoritmo 1 Pseudocódigo do Algoritmo *Quickmerge* [Quinn 1989]

```
 1: dim ← log2(p)                                   {dim é a dimensão do hipercubo}
 2: rank ← i                                        {Identificador do processo 0..2^dim − 1 }
 3: x ← n/p                                         {Quantidade de elementos em cada processo i}
 4: Início (para cada processo p_i em paralelo)
 5:    Ordene x elementos locais usando o algoritmo sequencial Quicksort
 6:    if dim > 0 then
 7:       if rank = 0 then
 8:          for i ← 1 to 2^dim − 1 do
 9:             splitter[i] ← a[(i * x)/2^dim]
10:       Processo 0 broadcast splitter[] para os processos 1..2^dim − 1
11:       for i ← dim − 1 downto 0 do
12:          parceiro ← rank ⊗ 2^i                 {Operação ou exclusivo bit a bit}
13:          index ← rank ⊙ (2^D − 2^(D−i)) ⊕ 2^(D−i)   {operações and e or bit a bit, }
14:          Use splitter[index] para particionar os elementos em 2 listas (maiores e menores)
15:          if rank > parceiro then
16:             Envie lista de elementos menores para o processo parceiro
17:             Receba a lista de elementos maiores do processo parceiro
18:          else
19:             Envie lista de elementos maiores para o processo parceiro
20:             Receba a lista de elementos menores do processo parceiro
21:          Una as duas listas em uma única lista ordenada
    Fim
```

Apesar de $2^{dim} - 1$ pivôs serem escolhidos e disseminados, apenas dim deles são empregados por cada processo (linhas 11 a 21). Conforme apresentado na Seção 2, em

cada rodada, pares de processos são formados para a troca de listas (linha 12). Então, de acordo com o seu identificador, cada par de processos escolhe um dos elementos de $splitter[]$ como pivô (linha 13) para dividir sua lista em duas: uma lista com elementos maiores que o pivô e a outra com elementos menores que o pivô (linha 14). Embora não seja listada entre as variáveis do algoritmo, D (linha 13), de acordo com Quinn, refere-se à dimensão do hipercubo. As linhas 15 a 20 realizam a troca das listas: um processo p_i, onde $i > j$, envia sua lista com elementos menores para p_j e recebe de p_j a lista com elementos maiores que o pivô. O processo p_i mantém sua lista com elementos maiores que o pivô e na linha 21 a une com a lista recebida de p_j e a reordena. Dessa forma, ao final da rodada p_i possui elementos maiores que o pivô. Por sua vez, p_j mantém os elementos menores ou iguais ao pivô.

Destacamos que, conforme o algoritmo está especificado no artigo original de Quinn [Quinn 1989], detectamos um erro na seleção dos pivôs (linha 13), descrito a seguir. Para a ordenação ocorrer de forma correta, cada *cluster* deve selecionar o mesmo pivô em cada rodada. Por exemplo, de acordo com a Figura 1, na primeira rodada há 1 *cluster* com 8 processos. Ao final dessa rodada os elementos menores ou iguais ao pivô estão nos processos 0, 1, 2 e 3 e os elementos maiores que o pivô nos processos 4, 5, 6 e 7. No entanto, da forma como está a linha 13, pivôs diferentes são escolhidos por cada processo. Além disso, na segunda rodada o mesmo pivô é selecionado para os 2 *clusters*. E na terceira rodada também há um único pivô para os 4 *clusters*. Por exemplo, na primeira rodada o processo p_0 seleciona o pivô $splitter[2]$, mas p_4 seleciona $splitter[6]$. Na rodada 2 o pivô será $splitter[4]$ e na terceira rodada $splitter[8]$.

Algoritmo 2 Pseudocódigo do Algoritmo *Quickmerge* com a sugestão de correção

```
11: for i ← dim − 1 downto 0 do
12:     ...
13:     index ← rank ⊙ (2^dim − 2^(dim−k)) ⊕ 2^(dim−k−1)   {operações and e or bit a bit, }
14:     k = k + 1                                          {k inicia com 0}
15:     ...
```

Uma forma de corrigir a linha 13 é a descrita no Algoritmo 2. Assumindo o mesmo exemplo anterior, os seguintes pivôs agora são selecionados: todos os processos na primeira rodada selecionam $splitter[4]$. Na segunda rodada, os processos do primeiro *cluster* selecionam $splitter[2]$ e os processos do segundo *cluster* selecionam $splitter[6]$. Então, na última rodada os 4 *clusters* selecionam os seguintes pivôs: $splitter[1]$, $splitter[3]$, $splitter[5]$ e $splitter[7]$, respectivamente.

O pseudo-código do ***Hyperquicksort***, proposto por Wagar [Wagar 1987] é apresentado no Algoritmo 3. No *Hyperquicksort* a seleção dos pares (linha 8) e a troca das listas (linhas 14 a 21 do Algoritmo 1) são idênticas ao algoritmo Quickmerge. A diferença está na seleção do pivô. Há um processo chamado de raiz (linha 9), escolhido a cada rodada, responsável por selecionar o elemento mediano de sua lista (linha 11) e disseminá-lo aos demais processos do seu *cluster* (linha 12). O processo raiz é o processo de menor identificador em cada *cluster*. Por exemplo, na primeira rodada (ver Figura 1) o processo raiz é o 0. Na segunda rodada há dois processos raízes: o processo 0 e o processo 4, pois há dois *clusters*. Na última rodada, os processos 0, 2, 4 e 6 são os raízes de seus respectivos *clusters*. Como os processos raízes são escolhidos durante o processo de ordenação, o

Algoritmo 3 Pseudocódigo do Algoritmo *Hyperquicksort* adaptado de [Quinn 1989]

1: $dim \leftarrow \log_2(p)$ {*Dimensão do hipercubo*}
2: $rank \leftarrow i$ {*Identificador do processo* $0..2^{dim} - 1$ }
3: $x \leftarrow n/p$ {*Quantidade de elementos em cada processo i*}
4: **Início** (para cada processo p_i em paralelo)
5: Ordene x elementos locais usando o algoritmo sequencial Quicksort
6: **if** $dim > 0$ **then**
7: **for** $i \leftarrow dim$ downto 1 **do**
8: $parceiro \leftarrow rank \otimes 2^i$ {*Ou exclusive bit a bit*}
9: $raiz \leftarrow$ processo com o menor identificador do cluster i
10: **if** $rank = raiz$ **then**
11: $splitter \leftarrow$ elemento mediano da lista ordenada localmente
12: Processo $raiz$ broadcast $splitter$ para os outros processos do cluster i
13: Use $splitter$ para particionar os elementos em 2 listas (maiores e menores)
14: ...
Fim

Hyperquicksort gera uma distribuição mais equilibrada de elementos entre os processos, ou seja, uma ordenação mais balanceada que o *Quickmerge*. Detalhes sobre o *Hyperquicksort* e sua versão tolerante a falhas são descritos em [Camargo and Duarte, Jr. 2016].

Considerando o problema da distribuição desigual de elementos entre os processos, Quinn sugeriu o ***Quickmerge Modificado***. A ideia de Quinn é não contar apenas com um processo para seleção do pivô. Cada elemento seria a média de todos os processos. A proposta não foi descrita com pseudocódigo, então neste trabalho assumimos que antes das rodadas de ordenação, os processos selecionam o $splitter[i]$ da sua própria lista local e uma média dos elementos é armazenada $splitter[i]$. Ou seja, não há a linha 7 do Algoritmo 1 e após a linha 9 há uma operação de redução seguindo com a média do elemento pelo número de processos.

4.1. *Quickmerge* Tolerante a Falhas

Neste trabalho é proposta uma versão tolerante a falhas do algoritmo Quickmerge na qual processos sem-falha assumem a computação de processos que falham. A função $c_{i,s}$ [Duarte et al. 2014], descrita na Tabela 1, define: a) a formação dos pares perante falhas de um dos processos e; b) qual processo sem-falha substitui o processo falho. Cada processo sem-falha conhece os processos falhos. Dessa forma, o algoritmo tolera até $p - 1$ processos falhos, sendo p o número total de processos. A função $c_{i,s}$ considera que os processos estão organizados logicamente em uma estrutura hierárquica que, quando todos os nodos estão sem falhas, corresponde ao hipercubo: i representa o processo p_i e s está relacionado a uma determinada rodada de ordenação. Inicialmente $s = dim$. A Tabela 1 exibe um exemplo da função $c_{i,s}$ aplicada a um hipercubo de três dimensões. O processo p_0 quando $s = 3$ tem o seguinte resultado: 4, 5, 6 e 7.

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 (Figura 1) o processo p_0 deve trocar elementos com o primeiro processo sem-falha resultante de $c_{0,3}$. Então, se p_4 está sem-falha p_0 e p_4 formam um par. Se p_4 se encontra falho, então p_0 deve trocar elementos com p_5. Nesse caso p_5 assume as funções de p_4 pois $c_{4,1}$ resulta em 5. Se p_5

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

também estiver falho então p_0 e p_6 formam o novo par. O processo p_6 assume p_4 pois como $c_{4,1}$ resultou em 5 e 5 está falho então se incrementa s e $c_{4,2}$ resulta em 6. Se tanto p_6 e p_7 estiverem falhos então p_0 não troca elementos com nenhum processo nessa rodada de ordenação e assume a responsabilidade dos dados do processo 4.

O pseudocódigo do *Quickmerge* tolerante a falhas é descrito no Algoritmo 4. A linha 4 introduz o vetor $map[]$. Esse vetor contém todos os processos que são responsabilidade de p_i e inclui o próprio p_i na primeira posição do vetor ($map[0] = p_i$). Considerando o exemplo anterior onde o p_4 falha na primeira rodada, então p_5 tem o seguinte: $map[0] = 5$ e $map[1] = 4$. A linha 6 ordena a lista local de cada processo e havendo processos falhos, processos sem-falha assumem suas tarefas. A linha 8 também considera que se p_0 está falho então outro processo, no caso p_1 assume a tarefa de distribuir o $splitter[]$ de p_0. As linhas 14 e 30 introduzem um laço de repetição para varrer o vetor $map[]$ de cada processo. A lógica do algoritmo é semelhante ao algoritmo tradicional, com a exceção que agora o processo deve verificar se ele é o único processo sem-falha no *cluster* (linhas 19 e 20). Em cada rodada o processo lê a lista de sua responsabilidade e a salva no dispositivo de armazenamento compartilhado (linha 15 e 28, respectivamente).

5. Resultados

A versão tolerante a falhas de cada algoritmo foi implementada na linguagem C usando a biblioteca ULFM 2.0 [MPI-Forum 2019]. A implementação do *Hyperquicksort* utilizada para comparação é a de [Camargo and Duarte, Jr. 2016]. A função `replace()` implementa o assinalamento das tarefas de um processo falho para outro sem-falha. Os códigos fontes estão disponíveis em `https://bitbucket.org/feelipeexavieer/wscad19`.

Uma das questões mais importantes na implementação é a detecção dos processos falhos. Ao início de cada rodada de ordenação a função `MPI_myBarrier()` foi implementada com o objetivo de identificar os processos falhos. Se o código `MPI_ERR_PROC_FAILED` ou `MPI_ERR_REVOKED` for retornado significa que um processo falhou. A partir de então o comunicador MPI deve ser reconstruído retirando aqueles processos que falharam. O novo comunicador conterá apenas os processos sem-falha. No entanto, o comunicador original é mantido para preservar a configuração original. Ou seja, se o processo 7 falhou em um conjunto de 8 processos, onde cada processo é identificado de 0 a 7, a comunicação deve ocorrer como se o processo 7 ainda estivesse operacional. No novo comunicador o processo 7 não existe. Novas verificações de falha sempre devem acontecer no novo comunicador. As funções ULFM apresentadas na Seção 3 foram empregadas para implementar a função `MPI_myBarrier()`.

A partir da detecção das falhas, a lista de processo falhos alimenta o Algoritmo 4 e a função `replace()`. Uma função para injetar falhas aleatórias também foi definida

Algoritmo 4 Pseudocódigo do Algoritmo *Quickmerge* Tolerante a Falhas

1: $dim \leftarrow \log_2(p)$ {*dim é a dimensão do hipercubo*}
2: $rank \leftarrow i$ {*Identificador do processo* $0..2^{dim}-1$ }
3: $x \leftarrow n/p$ {*Quantidade de elementos em cada processo i*}
4: $map[]$ {*Processos que são responsabilidade de* p_i *devido a falha de* p_j}
5: **Início** (para cada processo p_i em paralelo)
6: Ordene x elementos locais de sua responsabilidade usando o algoritmo sequencial Quicksort
7: **if** $dim > 0$ **then**
8: **if** $rank =$ substituto de p_0 **then**
9: **for** $i \leftarrow 1$ to $2^{dim}-1$ **do**
10: $splitter[i] \leftarrow a[(i * x)/2^{dim}]$
11: Substituto de p_0 broadcast $splitter[]$ para os processos sem-falha
12: **for** $i \leftarrow dim-1$ downto 0 **do**
13: $k \leftarrow 0$
14: **repeat**
15: Ler a lista de $map[k]$
16: $parceiro \leftarrow map[k] \otimes 2^i$ {*Operação ou exclusivo bit a bit*}
17: $index \leftarrow map[k] \odot (2^D - 2^{D-i}) \oplus 2^{D-i}$ {*operações and e or bit a bit,* }
18: Use $splitter[index]$ para particionar os elementos em 2 listas (maiores e menores)
19: **if** $rank = parceiro$ **then**
20: *Mesmo processo manipula as duas listas* {*Só há um processo sem-falha no cluster*}
21: **else if** $map[k] > parceiro$ **then**
22: *Envie lista de elementos menores para o processo parceiro*
23: *Receba a lista de elementos maiores do processo parceiro*
24: **else**
25: *Envie lista de elementos maiores para o processo parceiro*
26: *Receba a lista de elementos menores do processo parceiro*
27: *Una as duas listas em uma única lista ordenada*
28: Salvar a lista de $map[k]$ em disco compartilhado
29: incrementa k
30: **until** k = tamanho de $map[]$
Fim

para realizar os testes. A função `applyFailures()` escolhe quais processos irão falhar em uma rodada de ordenação. Foram definidos quatro cenários. No primeiro cenário nenhuma falha é inserida. No segundo cenário um processo deve falhar. No terceiro, metade dos processos falham. Por último, $p-1$ processos falham.

Os experimentos foram executados em uma máquina com 32 processadores *Intel Core i7*. O sistema operacional é o *Linux Kernel 4.4.0*. O desempenho dos algoritmos foi comparado em um cenário sem falhas (Figura 2) e com falhas (Figura 3) para 4, 8, 16 e 32 processos MPI. No cenário sem falhas, foram geradas 10 amostras, cada uma com 2^{30} números inteiros no intervalo de -2^{31} e $2^{31}-1$. Cada amostra foi executada 3 vezes e a média do tempo de execução em segundos é apresentada. O cenário com falhas avaliou uma das amostras considerando 0, 1, $p/2$ e $p-1$ processos falhos (10 execuções). O comando `time` do Linux foi usado para obter o tempo de execução.

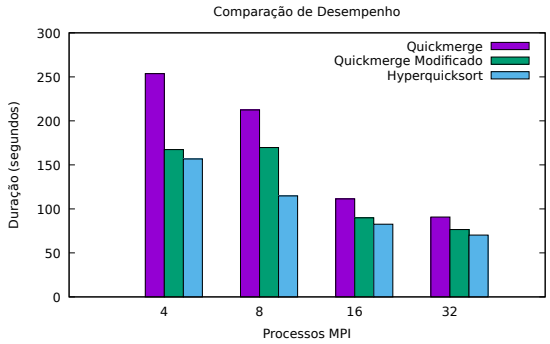


Figura 2. Desempenho dos algoritmos para ordenar 2^{30} números inteiros.

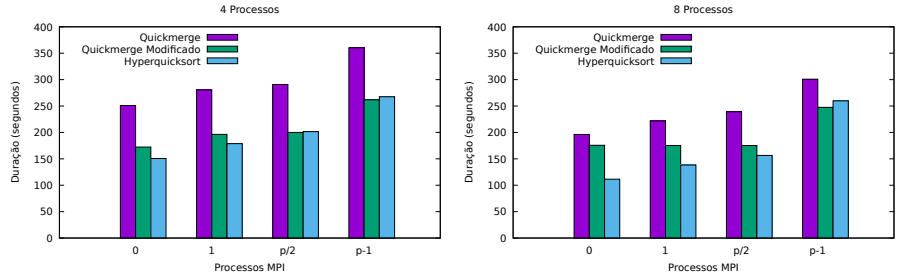
	Quickmerge				*Quickmerge Mod.*				*Hyperquicksort*			
Amostra	**4P**	**8P**	**16P**	**32P**	**4P**	**8P**	**16P**	**32P**	**4P**	**8P**	**16P**	**32P**
1	38913	26153	35849	50381	457	406	416	757	28167	31181	22115	16592
2	11207	32189	50217	52969	507	362	844	568	17797	16461	20146	22997
3	16858	48642	52836	75487	379	287	266	684	14836	36475	17725	32651
4	24574	53209	32484	62281	129	390	553	1080	28376	23486	21965	17181
5	22045	74242	33113	60720	50	431	536	569	10820	27626	14830	18664
6	46879	54687	49822	64603	197	232	487	715	27907	23583	32440	12686
7	36157	51946	48067	57307	183	295	823	978	22440	24729	16789	18827
8	32214	87790	62499	53591	213	437	509	804	23555	33736	25799	19465
9	38989	44518	62881	57282	99	250	692	860	18976	27324	20465	33117
10	52748	48903	51712	78884	303	565	524	747	43984	30395	15618	33095

Tabela 2. Diferença entre a maior lista e o tamanho ideal.

A Figura 2 compara o desempenho dos algoritmos. O eixo y apresenta o tempo de execução e o eixo x o número processos MPI. O *Quickmerge* obteve o pior desempenho entre os algoritmos. Já o *Hyperquicksort* apresentou o menor tempo de execução. Por exemplo, o *Quickmerge* leva em média 253 segundos, enquanto o *Quickmerge Modificado* e o *Hyperquicksort* levam 167 e 156 segundos, respectivamente, para 4 processos MPI.

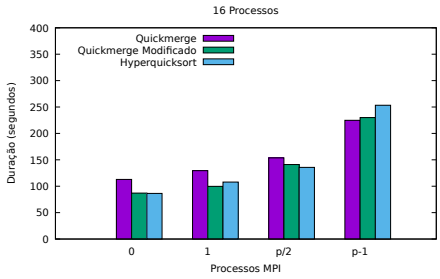
A Tabela 2 apresenta o balanceamento entre os processos. O resultado foi obtido a partir da diferença entre a maior lista encontrada nos processos ao final da ordenação e o tamanho considerado ideal. Por exemplo, para a amostra 1, no algoritmo *Quickmerge*, a maior lista tem tamanho 268.474.369 e o tamanho ideal seria 268.435.456. O algoritmo *Quickmerge Modificado* apresentou o melhor balanceamento e o *Quickmerge* o pior balanceamento ao final da ordenação. De fato, a diferença de desempenho entre o *Quickmerge Modificado* e o *Hyperquicksort* é pequena para 4, 16 e 32 processos, de acordo com a Figura 2. Importante lembrar que *Quickmerge Modificado* realiza uma operação de redução para obter as médias enquanto o *Hyperquicksort* transmite apenas um elemento (o pivô) a cada rodada. Isso deve ter contribuído para o melhor desempenho do *Hyperquicksort*.

A Figura 3 compara o desempenho dos algoritmos perante falhas usando 4, 8, 16 e 32 processos MPI. O eixo x considera a execução com 0 , 1, $(p/2)$ e $p-1$ processos falhos ao final da execução. Como as falhas são injetadas aleatoriamente, é possível que o processo falhe logo na primeira rodada ou na última. O momento da falha impacta diretamente no tempo total da ordenação. Na Figura 3a o *Quickmerge* se mantém com o maior tempo de execução enquanto os outros algoritmos apresentam pequena diferença entre si. Nas Figuras 3b a 3d o tempo vai se tornando semelhante pois as falhas obrigam os processos a assumirem processamento extra.

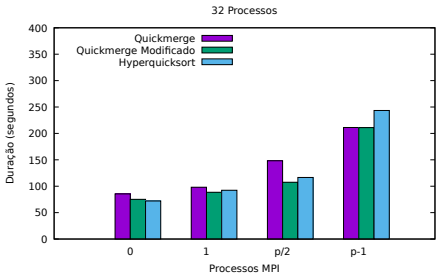


(a) 4 processos

(b) 8 processos



(c) 16 processos



(d) 32 processos

Figura 3. Desempenho dos algoritmos perante falhas.

6. Conclusão

Este trabalho apresentou uma implementação tolerante a falhas em MPI do algoritmo *Quickmerge* usando a especificação ULFM do MPI-Fórum. A implementação é capaz de executar a ordenação mesmo se todos menos um processo falharem. A implementação é baseada no VCube, uma topologia virtual que mantém diversas das propriedades do hipercubo. Um processo sem-falha do menor *cluster* do VCube assume as funções do processo falho. O *Quickmerge Modificado* também foi implementado. Ambos foram comparados ao algoritmo *Hyperquicksort*. Resultados de desempenho mostram a eficiência da implementação para ordenar 2^{30} números inteiros em cenários com falhas e sem falhas. O *Hyperquicksort* obteve melhor desempenho e o algoritmo *Quickmerge Modificado* o melhor balanceamento. Nos experimentos com falhas os algoritmos se assemelham em tempo de execução conforme aumenta o número de falhas. Quinn havia concluído que o *Hyperquicksort* era levemente superior aos dois algoritmos e que a versão modificada gerava melhor balanceamento. No entanto, avaliou os algoritmos apenas para 16.384 números inteiros. Como trabalhos futuros pretende-se implementar e avaliar o *Hyperquicksort Modificado*.

Referências

Ashraf, R. A., Hukerikar, S., and Engelmann, C. (2018). Shrink or substitute: Handling process failures in HPC systems using in-situ recovery. In *Euromicro Int. Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 178–185.

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.

Camargo, E. T. and Duarte, Jr., E. (2016). Uma implementação MPI tolerante a falhas do algoritmo hyperquicksort. In *Simp. Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC) 2016 - Workshop de Tolerância a Falhas*.

Camargo, E. T. d. and Duarte, Jr., E. P. (2017). Minicurso v: Técnicas para a construção de sistemas MPI tolerantes a falhas. In *Minicurso do Simpósio de Sistemas Computacionais de Alto Desempenho (WSCAD) 2017*.

Camargo, E. T. d. and Duarte, Jr., E. P. (2018). Running resilient MPI applications on a dynamic group of recommended processes. *J. of the Brazilian Comp. Society*, 24(1):5.

Duarte, E. P., Bona, L. C. E., and Ruoso, V. K. (2014). Vcube: A provably scalable distributed diagnosis algorithm. In *2014 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pages 17–22.

Evans, D. (1990). A parallel sorting - merging algorithm for tightly coupled multiprocessors. *Parallel Computing*, 14(1):111 – 121.

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

Forum, M. (2019). MPI forum website. `http://mpi-forum.org/`. Em 23/07/2019.

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

Gamell, M., Teranishi, K., Mayo, J., Kolla, H., Heroux, M. A., Chen, J., and Parashar, M. (2017). Modeling and simulating multiple failure masking enabled by local recovery for stencil-based applications at extreme scales. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2881–2895.

MPI-Forum (2019). User-level failure mitigation. `https://bitbucket.org/icldistcomp/ulfm/`. Acessado em 26/02/2019.

mpich.org (2017). High-performance portable mpi. `http://www.mpich.org/`. Acessado em 23/07/2019.

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

Quinn, M. J. (1988). Parallel sorting algorithms for tightly coupled multiprocessors. *Parallel Computing*, 6(3):349 – 357.

Quinn, M. J. (1989). Analysis and benchmarking of two parallel sorting algorithms: Hyperquicksort and quickmerge. *BIT Numerical Mathematics*, 29(2):239–250.

Shahzad, F., Thies, J., Kreutzer, M., Zeiser, T., Hager, G., and Wellein, G. (2019). Craft: A library for easier application-level checkpoint/restart and automatic fault tolerance. *IEEE Transactions on Parallel and Distributed Systems*, 30(3):501–514.

Shi, H. and Schaeffer, J. (1992). Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14(4):361 – 372.

Träff, J. L. (2018). Parallel quicksort without pairwise element exchange. *CoRR*, abs/1804.07494.

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