Uma Implementação MPI Tolerante a Falhas do Algoritmo *Bitonic Sort*

Lucca Gabriel M. dos Santos[1], **Edson Tavares de Camargo**[1], **Elias P. Duarte Jr.**[2]

[1] Universidade Tecnológica Federal do Paraná - 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

luccagabriel12@hotmail.com, edson@utfpr.edu.br, elias@inf.ufpr.br

Abstract. *The Bitonic Sort algorithm performs sorting by executing splitting and merging operations on a bitonic sequence. A bitonic sequence is a sequence of numbers which is first strictly increasing then after a point strictly decreasing. Despite of the fact that many variants and implementations have been proposed, none of the existing parallel implementation is capable of tolerating process failures at runtime. This paper presents an MPI implementation of Bitonic Sort that tolerates up to $n-1$ process failures, where n is the total of processes. The implementation is based on the most recent MPI-Forum specification for MPI to deal with failures. Experimental results show the efficiency of the implementation to sort up to 1 billion integers.*

Resumo. *O algoritmo Bitonic Sort executa a ordenação através de operações de divisão e mesclagem de sequências bitônicas. Uma sequência é dita bitônica se seus elementos crescem e depois, a partir de algum ponto, decrescem. Apesar de muitas variantes do algoritmo encontradas na literatura, não se conhece uma implementação paralela capaz de tolerar falhas de processos em tempo de execução. Este trabalho apresenta uma implementação MPI do Bitonic Sort capaz de suportar até $n-1$ falhas de processos, onde n é o total de processos. A mais recente especificação de tolerância a falhas proposta pelo MPI-Fórum é empregada no desenvolvimento da solução. Resultados experimentais 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 [Quinn 2003]. Algoritmos paralelos de ordenação são utilizados, por exemplo, em processamento de imagens, geometria computacional e teoria dos grafos [Quinn 2003, Durad et al. 2014]. Geralmente, algoritmos de ordenação paralelos empregam uma estratégia de ordenação sequencial em uma topologia de rede física ou virtual. Uma topologia de rede conecta fisicamente ou virtualmente os processos ou nodos, que são organizados em malha, anel, estrela ou em um hipercubo, por exemplo. Entre os algoritmos clássicos de ordenação paralela, destaca-se o *Bitonic Sort* [Batcher 1968, Lee and Batcher 1994].

O *Bitonic Sort* foi proposto inicialmente por Batcher em 1968 para ordenadar um sequência de 2^m elementos, onde m é o tamanho da sequência. O algoritmo é baseado na

ideia de uma rede de ordenação [Pacheco 1997]. Em uma rede de ordenação, a sequência e a direção das comparações é determinada previamente e independe da sequência de entrada, sendo assim adequada para implementação em hardware e em máquinas de processamento paralelo [Kumar 2002]. No *Bitonic Sort* a ordenação se baseia em uma sequência bitônica. Uma sequência bitônica é uma sequência monotonicamente crescente seguida de uma sequência monotonicamente decrescente (ou uma rotação cíclica dessa sequência) [Kumar 2002]. Por exemplo, a sequência $s = (2, 3, 6, 8, 7, 5, 4, 1)$ é uma sequência bitônica, pois os primeiros 4 elementos são crescentes e os demais decrescentes. Sequências bitônicas podem ser formadas a partir de quaisquer elementos pela concatenação de sequências bitônicas menores. Ou seja, a partir de qualquer entrada é possível gerar uma sequência bitônica.

O *Bitonic Sort* foi adaptado para executar em uma variedade de arquiteturas paralelas, tais como redes em malha (*mesh*) [Nassimi and Sahni 1979] e o hipercubo [Johnson 1984]. Na sua versão para o hipercubo, o *Bitonic Sort* assume que a sequência a ser ordenada é igualmente distribuída entre os *n* nodos de um hipercubo. Cada nodo então ordena individualmente sua sequência e então repetidamente executa a mesclagem das sequências em subcubos sucessivamente maiores. Ao final da ordenação o hipercubo possui a sequência ordenada nos seus consecutivos nodos de forma que o nodo n_i possui elementos menores ou iguais ao nodo n_j, onde $i < j$. O *Bitonic Sort* ainda é ainda alvo de vários estudos e implementações [Chen et al. 2015, Al-Hashimi et al. 2017, Velusamy et al. 2018]. Em [Sheu et al. 1992], uma estratégia de tolerância a falhas é aplicada ao *Bitonic Sort* que permite a ocorrência de até r falhas, onde r é a dimensão do hipercubo; as falhas são permanentes e de processos de máquinas paralelas nas quais os processadores são organizados em hipercubo. Além do contexto distinto, a solução introduzida no presente trabalho tolera até $n - 1$ falhas de processos em tempo de execução, onde n é o número total de processos, independentemente da arquitetura das máquinas paralelas.

O MPI (*Message-Passing Interface*) é o padrão de *facto* para o desenvolvimento de aplicações [Fagg and Dongarra 2000] paralelas de memória distribuída. 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] e portanto não oferece suporte à falhas. A *User Level Failure Mitigation* (ULFM) [Bland et al. 2013] é mais recente proposta do MPI-Fórum para lidar com as falhas da aplicação em nível de programação. Ao usar a ULFM o próprio desenvolvedor da aplicação é o responsável por programar a estratégia de tolerância a falhas.

Este trabalho apresenta uma implementação MPI tolerante a falhas do algoritmo paralelo de ordenação *Bitonic Sort*. A implementação parte do príncipio que um processo correto, ou seja, sem-falha, assume as tarefas de um processo que é detectado como falho. A abordagem utilizada se baseia na implementação tolerante a falhas do algoritmo *Hyperquicksort* [Camargo and Duarte, Jr. 2016], onde uma função de mapeamento define quais processos sem-falha assumem a tarefa do processo que falhou. Essa abordagem é capaz de tolerar até $n - 1$ falhas de processos em tempo de execução. Ao final do processo de ordenação as sequências inicialmente distribuídas a cada processo estarão ordenadas conforme prevê o *Bitonic Sort*. Resultados experimentais são apresentados para a ordenação

de até 1 bilhão de inteiros e confirmam a eficiência da implementação.

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 *Bitonic Sort* e a sua versão tolerante a falhas é apresentada na Seção 4. A implementação é descrita 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 transmitindo e recebendo 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 com identificadores dos nodos em números inteiros e bits. 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.

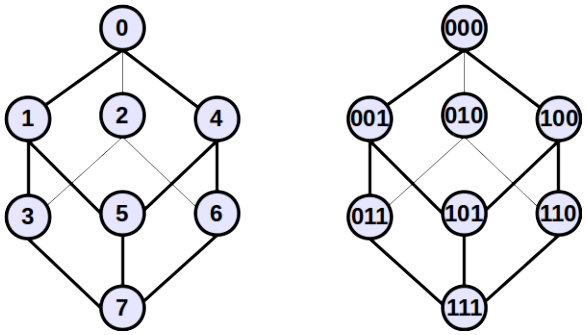


Figura 1. Hipercubo de 3 dimensões.

3. Tolerância a Falhas em MPI

O *Message Passing Interface* (MPI) [MPI Forum 2015] é um padrão para a comunicação de dados em computação paralela. O padrão especifica um conjunto de interfaces para troca de mensagens entre processos geralmente distribuídos em uma rede. O objetivo é tornar a comunicação entre diferentes plataformas transparente. Há primitivas para comunicação ponto-a-ponto, coletiva, gerencia de grupos, leitura e escrita paralela, entre outras. A primeira versão do padrão, chamada de MPI-1, foi proposta em 1994 e, atualmente, encontra-se na sua terceira versão. A especificação é mantida pelo MPI-Fórum que organiza a evolução do padrão [Forum 2019a].

Uma das questões que giram em torno do padrão é se o mesmo deve ou não incorporar o tratamento de falhas. Desde a sua primeira versão, o MPI parte do princípio que a infraestrutura responsável por executar a aplicação paralela é confiável. Dessa forma, as mais tradicionais implementações do padrão, como a [Gropp et al. 1996] e a Open MPI [Gabriel et al. 2004], simplesmente abortam toda a aplicação mesmo se um único processo falhar.

Consequentemente, como alternativa, delega-se às implementações específicas ou ferramentas externas a responsabilidade da detecção e recuperação da aplicação. A técnica de tolerância a falhas mais empregada em MPI é o *rollback-recovery* [Elnozahy et al. 2002]. O principal objetivo dos protocolos de *rollback-recovery* é restaurar o sistema a partir de um estado consistente salvo previamente. Periodicamente a aplicação, ou seja, cada processo, salva o seu estado em um dispositivo confiável enquanto realiza a computação. Após a ocorrência de uma falha, a aplicação usa o estado salvo previamente para reiniciar a sua computação. Embora o *rollback-recovery* seja capaz de lidar com falhas no ambiente MPI, a técnica desperdiça tempo de computação ao exigir que informações de recuperação sejam salvas de tempos em tempos.

Diversos trabalhos buscaram incorporar e padronizar a tolerância a falhas na especificação do MPI para permitir que a própria aplicação seja capaz de lidar com suas falhas [Camargo and Duarte, Jr. 2018]. A especificação ULFM é o mais recente esforço do MPI-Fórum para padronizar a semântica de tolerância a falhas em MPI [Forum 2019b]. A ULFM não determina uma estratégia de recuperação específica. O objetivo é permitir que o desenvolvedor escolha a técnica de tolerância a falhas que melhor se adeque ao seu programa. Atualmente, uma implementação da ULFM está disponível na implementação Open MPI [MPI-Forum 2019].

A ULFM adota o modelo de falhas *fail-stop*. Uma falha é detectada durante a comunicação entre um processo correto com um processo que falhou. A partir de então, a especificação define que uma indicação de erro deve ser retornada ao processo que detectou a falha. Para que todos os demais processos corretos sejam notificados da falha, considerando que ainda não se comunicaram com o processo que falhou, é possível usar uma das primitivas definidas pela ULFM. Se o desenvolvedor desejar que a partir da detecção da falha a aplicação continue sua computação então é possível recuperar o comunicador MPI também usando uma das primitivas da ULFM. O comunicador MPI é um elemento fundamental do MPI 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, respectivamente, 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 pontencial 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 envolvendo todos os processos corretos. Os processo devem concordar com o valor contido em `flag`. Essa primitiva permite que todos os processos sem-falha tenham uma única visão sobre os processos que falharam.
- `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.

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 pode concretamente ser promovida a 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. O trabalho desenvolvido em [Camargo and Duarte, Jr. 2017] apresenta mais detalhes sobre a implementação de tolerância a falhas em MPI usando a ULFM.

4. O Algoritmo *Bitonic Sort*

O algoritmo *Bitonic Sort* é um dos algoritmos clássicos de ordenação paralela. O algoritmo define uma rede de ordenação, ou seja, compara elementos em uma sequência predefinida, de forma que a sequência de comparação não depende dos dados de entrada. Sua complexidade é estimada em $O(\log^2 m)$ para ordenar m elementos [Kumar 2002], onde m é uma potência de 2. Por isso, o algoritmo é adequado para implementações em hardware e computadores de arquitetura paralela. O método de operação do *Bitonic Sort* tem como base a ordenação de uma sequência bitônica em uma sequência ordenada.

Uma sequência bitônica é uma sequência de elementos $(a_0, a_1, ..., a_{m-1})$ com a propriedade de que tanto (1) existe um índice i, $0 \leq i \leq m-1$, tal que $(a_0, ..., a_i)$ é monotonicamente crescente e $(a_{i+1}, ..., a_{m-1})$ é monotonicamente decrescente, ou (2) existe uma rotação cíclica que satisfaça (1). Por exemplo, $seq = (2, 3, 6, 8, 7, 5, 4, 1)$ é uma sequência bitônica para $i = 3$, formada pela sequência monotonicamente crescente $(2, 3, 6, 8)$ e pela monotonicamente decrescente $(7, 5, 4, 1)$. Como $(6, 8, 7, 5, 4, 1, 2, 3)$ é uma rotação cíclica da sequência original, também e bitônica. Qualquer subsequência de uma sequência bitônica também é considerada bitônica.

A partir de uma sequência bitônica, uma sequência ordenada, ou seja, monotonicamente crescente, pode ser obtida da forma seguinte. A sequência seq é dividida em sua metade, gerando as sequências seq_1 e seq_2 ambas de tamanho $m/2$. A partir de então, compara-se o primeiro elemento da sequência seq_1 com o primeiro elemento da sequência seq_2. O menor elemento fica na sequência seq_1 e o maior elemento na sequência seq_2, ou seja: $seq_1 = (min\{a_0, a_{m/2}\}, min\{a_1, a_{m/2+1}\}, ..., min\{a_{m/2-1}, a_{m-1}\})$ e $seq_2 = (max\{a_0, a_{m/2}\}, max\{a_1, a_{m/2+1}\}, ..., max\{a_{m/2-1}, a_{m-1}\})$. Considerando a sequência seq, descrita anteriormente o resultado seria o seguinte: $seq_1 = (2, 3, 4, 1)$ e

$seq_2 = (7, 5, 6, 8)$. Tanto seq_1 quanto seq_2 também são sequências bitônicas. Observe que todos os elementos de seq_1 são menores do que os contidos em seq_2. O próximo passo é aplicar novamente o método nas novas sequências e repeti-lo recursivamente até que haja apenas 2 elementos em cada sequência. A partir de então, basta unir as subsequências.

O procedimento de dividir uma sequência de tamanho m em duas sequências bitônicas é chamado de divisão bitônica (*bitonic split*). O procedimento de gerar uma sequência ordenada a partir da divisão bitônica é chamado de mesclagem ou fusão bitônica (*bitonic merge*). Qualquer sequência bitônica pode ser ordenada usando os procedimentos de divisão, comparação dos elementos e fusão das sequências ordenadas. A rede de ordenação do Bitonic Sort pode ainda ser adaptada e usada para ordenação em algoritmos paralelos usando uma topologia em malha ou a topologia do hipercubo, por exemplo. O *Bitonic Sort* adaptado para o hipercubo é descrito a seguir.

Os processos são organizados na forma de um hipercubo virtual de dim dimensões, onde $dim = \log n$ e n é a quantidade de processos disponíveis para a ordenação. Dessa forma P representa o conjunto de processos $P = \{p_0, p_1, ..., p_{n-1}\}$. A ordenação é executada em s rodadas, onde $0 < s \leq dim$. A cada rodada, pares de processos são formados de acordo com a operação $p_i \oplus 2^{(s-1)}$. O símbolo $\oplus$ representa a operação binária de OU exclusivo (XOR). Por exemplo, considerando um hipercubo de 3 dimensões o par do processo p_0 seria o processo p_4. Os processos p_i e p_j trocam elementos entre si e realizam comparações com base nos elementos trocados em cada rodada s. O processo p_i, onde $i < j$, mantém os menores elementos, o processo p_j, onde $j > i$ mantém os maiores elementos. Ao final os elementos em cada processo estão ordenados de forma que em cada processo p_i o maior elemento é menor ou igual ao menor elemento no processo $p_i + 1$.

O mapeamento de uma sequência bitônica pode considerar dois cenários. No primeiro, cada elemento é mapeado para um processo do hipercubo, ou seja, o elemento a_i é mapeado para o processo p_i. O tamanho da sequência m é igual o número de processos n ($m = n$). No segundo cenário, o tamanho da sequência é maior que o número de processos disponíveis ($m > n$). O Algoritmo 1 descreve o primeiro cenário.

Algorithm 1 Ordenação de uma sequência bitônica em um hipercubo de d dimensões.

```
1: procedure Bitonic_Sort(p_i, dim) (para cada processo p_i executando em paralelo)
2: Begin
3:    for s ← dim − 1 downto 0 do
4:       p_j ← p_i ⊕ 2^s)                                   {ou exclusivo}
5:       if p_i < p_j then
6:          comp_exchange min(p_i, p_j)
7:       else
8:          comp_exchange max(p_i, p_j)
   End
```

A função **comp_exchange** do Algoritmo 1 é responsável por enviar e receber o elemento contido em um processo ao seu parceiro e compará-los. A Figura 2 apresenta o processo de ordenação para a sequência $seq = (2, 3, 6, 8, 7, 5, 4, 1)$, descrita anteriormente, em um hipercubo de 3 dimensões. Cada elemento a_i da sequência seq é distribuído para o processo p_i. Na primeira rodada de ordenação os pares são formados de acordo

com a linha 4 do Algoritmo 1: (0, 4), (1, 5), (2, 6) e (3, 7). O processo de menor identificador mantém o menor elemento e processo de identificador maior mantém o maior elemento (linhas 5 e 8, respectivamente). Considerando que os pares (0, 4) contêm os elementos 2 e 7, respectivamente, após a troca e comparação ambos permanecem com os seus elementos. O mesmo não acontece com os pares de processos (2, 6) e (3, 7). O processo p_2 mantém o elemento 4 e p_6 o elemento 6. Na próxima rodada o hipercubo é dividido em 2 e novos pares de processos são formados de acordo com a linha 4. Os elementos são novamente trocados e comparados. É possível perceber que os elementos contidos em um subcubo são todos menores do que outro subcubo. Na última rodada a sequência está ordenada com cada processo p_i contendo um elemento que é menor ou igual ao elemento contido em p_j, onde $i < j$.

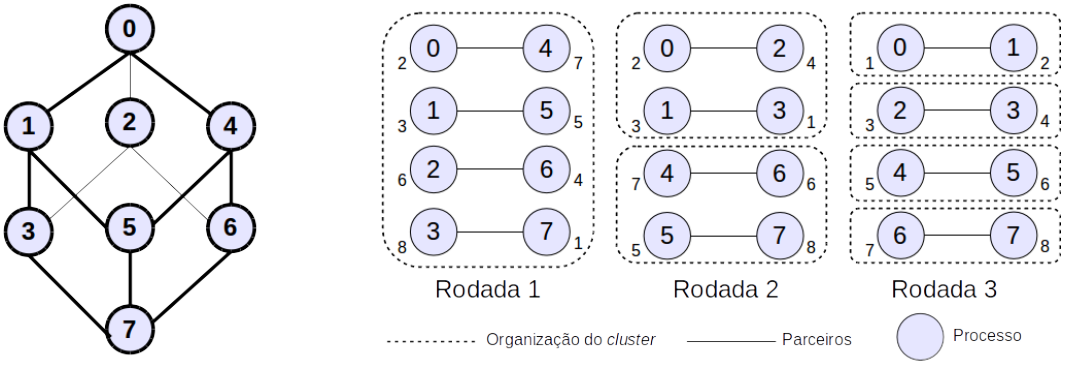


Figura 2. *Bitonic Sort* em um hipercubo com 8 processos.

Para o segundo cenário, no qual $m > n$, a sequência seq de tamanho m é dividida pelo número de processos (m/n). O processo p_0 recebe a primeira parte da sequência, ou seja, seq_0, o processo p_1 recebe a segunda parte, seq_1, e assim por diante. Neste cenário, a função **comp_exchange** do Algoritmo 1 troca as sequências seq_i e seq_j contidas nos processos p_i e p_j, respectivamente. A partir de então cada processo faz comparações entre os índices das sequências. O processo p_i compara $seq_i[k] \leftarrow min(seq_i[k], seq_j[k])$ e p_j compara $seq_j[k] \leftarrow max(seq_i[k], seq_j[k])$, ou seja, os elementos menores são mantidos em seq_i e os maiores em seq_j, para $i < j$.

Algorithm 2 Ordenação de uma sequência qualquer em um hipercubo de d dimensões.

```
1: procedure Bitonic_Sort(p_i, dim) (para cada processo p executando em paralelo)
2: Begin
3:    for s ← 0 to dim − 1 do
4:       for t ← s downto 0 do
5:          p_j ← p_i ⊕ 2^(t)                                   {ou exclusivo}
6:          if (p_i >> (s + 1)) and (p_i >> t) are both odd or even then
7:             comp_exchange max(p_i, p_j)
8:          else
9:             comp_exchange min(p_i, p_j)
   End
```

O algoritmos apresentados até o momento ordenam uma sequência bitônica. O *Bitonic Sort* permite ordenar uma sequência qualquer, porém é preciso primeiro transformá-la em uma sequência bitônica. Isso pode ser realizado ao unir repetidamente

sequências bitônicas de tamanho crescente. Como quaisquer dois elementos formam uma sequência bitônica, a fusão de sequências de dois elementos cada também gera uma sequência bitônica. Basta que a primeira sequência seja crescente e a segunda decrescente. O Algoritmo 2 recebe uma sequência qualquer e a transforma em bitônica. A Figura 3 ilustra o processo de ordenação de acordo com o Algoritmo 2 para a sequência de entrada $seq = (7, 3, 6, 8, 1, 2, 5, 4)$ em um hipercubo de 3 dimensões. Na Figura 3 os símbolos $\oplus$ e $\ominus$ representam as comparações entre os elementos da sequência. O primeiro define que as comparações devem gerar uma sequência crescente e o segundo uma sequência decrescente.

Inicialmente cada elemento a_i da sequência é mapeado para o processo p_i. Na primeira rodada, os pares de processos (0, 1), (2, 3), (4, 5), (6, 7) são formados de acordo com a linha 5 do Algoritmo 2. Cada processo envia seus elementos para o seu par. A linha 6 define se o processo mantém os menores elementos ou os maiores elementos. Há uma operação de deslocamento de *bits* a direita ($>>$). Se o identificador do processo deslocado $s + i$ *bits* à direita e t *bits* à direita são ambos pares ou ambos ímpares, então o menor elemento permanece no processo com esse identificador. Caso contrário esse processo manterá o maior elemento. Dessa forma, o elemento p_0 mantém o elemento 3 e p_1 mantém o elemento 7 uma vez que esses processos devem gerar uma sequência crescente (ver Figura 3). Os elementos contidos em p_4 e p_5 são mantidos pois já formavam uma sequência crescente. Por sua vez os pares de processos (2, 3), (6, 7) geram sequências decrescentes. Assim p_2 mantém o elemento 8 e p_3 o elemento 6. Processos p_6 e p_7 mantém seus valores originais.

Na segunda rodada, para $s \leftarrow 1$, o processo de troca e comparação se repete considerando agora dois subcubos de dimensão 2 e depois subcubos de tamanho 1. A última rodada do Algoritmo 2 equivale ao Algoritmo 1 (ver Figuras 2 e 3). Como é possível verificar a última rodada inicia o processo de ordenação com a sequência bitônica $seq = (3, 6, 7, 8, 5, 4, 2, 1)$. Ao final, o algoritmo gera a sequência $seq = (1, 2, 3, 4, 5, 6, 7, 8)$.

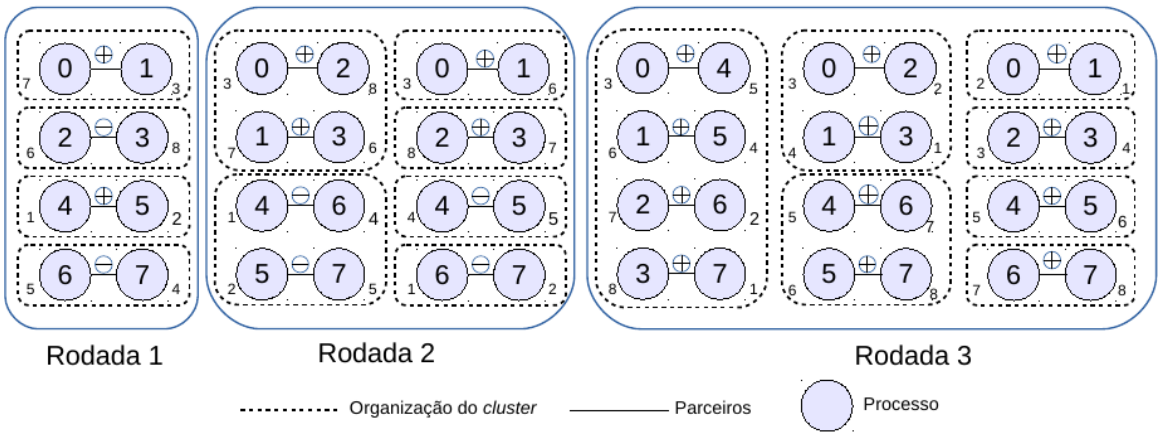


Figura 3. *Bitonic Sort* em um hipercubo com 8 processos para qualquer sequência.

4.1. *Bitonic Sort* Tolerante a Falhas

O *Bitonic Sort* tolerante a falhas é apresentado no Algoritmo 3. O algoritmo é capaz de tolerar até $n - 1$ falhas em tempo de execução, sendo n o número de processos res-

ponsáveis pela ordenação. O algoritmo faz uso das funções `partner()` e `replace()`. A primeira é responsável por formar os pares de processos em cada rodada. A segunda é responsável por definir qual processo sem-falha substitui um processo falho em uma rodada.

A ideia central da solução proposta é fazer os processos sem-falha substituirem os processos falhos. Dessa forma, após o processo p_i realizar suas próprias tarefas em uma rodada, p_i deve verificar se há algum processo falho p_j nessa rodada que seja de sua responsabilidade. Em caso afirmativo, p_i executa as tarefas de p_j antes de seguir para a próxima rodada de ordenação.

Algorithm 3 *Bitonic Sort* tolerante a falhas para um hipercubo de d dimensões.

```
 1: procedure Bitonic_Sort_Fail(p_i, dim) (para cada processo p em paralelo)
 2: Begin
 3:    for s ← 0 to dim − 1 do
 4:       for t ← s downto 0 do
 5:          makeMap(map, faults, p_i, dim)
 6:          p_j ← p_i ⊕ 2^(t)                                   {ou exclusivo}
 7:          partner ← partner(p_j, t)
 8:          asc ← (p_i >> (s + 1)) and (p_i >> t) are both odd or even   {>> rotação de bits}
 9:          if p_i ≠ partner then
10:             if (testPartner(p_i, partner, dim)) then
11:                if (asc) then
12:                   comp_exchange max(p_i, partner)
13:                else
14:                   comp_exchange min(p_i, partner)
15:                testIfNeedTakePlace(map, partner, p_i, i, j, faults)
16:             else
17:                if (asc) then
18:                   comp_exchange max(p_i, partner)
19:                   testIfNeedTakePlace(map, partner, p_i, i, j, faults)
20:                else
21:                   testIfNeedTakePlace(map, partner, p_i, i, j, faults)
22:                   comp_exchange min(p_i, partner)
23:          else
24:             partner ← p_i ⊕ 2^(t)                            {ou exclusivo}
25:             readCheckpoint(p_i, partner)
26:             testIfNeedTakePlace(map, partner, p_i, i, j, faults)
27:          checkpoint(p_i)
    End
```

Os processos iniciam uma rodada de ordenação com o seu estado salvo em dispositivo estável compartilhado por todos os processos. Ao final de uma rodada um novo `checkpoint()` é executado (linha 27, Algoritmo 3). A primeira tarefa de um processo p_i (linha 5) em uma rodada s, sub-rodada t, é descobrir quais processos falhos são de sua resposabilidade. Isso é realizado através da função `Makemap()`. Com auxílio da função `replace()` essa função preenche o vetor `map`. O vetor `map` possui os processos falhos que agora são resposabilidade de p_i. A linha 6 encontra o processo p_j, o parceiro natural de p_i. Na linha 7, o algoritmo define o `partner` de p_i através da função `partner()`.

Se p_j estiver sem-falha, então `partner` será o próprio p_j, caso contrário `partner` será um processo sem-falha definido pela função $c_{i,s}$ descrita mais adiante.

A linha 8 define se p_i mantém o maior ou o menor elemento após trocar seus elementos com o parceiro (linhas 11 a 14 e 17 a 22). Se `partner` for diferente de p_i, então as linhas 10 e 22 são executadas. A linha 10 testa se `partner` é um parceiro natural, ou seja, que não falhou. Nesse caso, trocas e comparações são executadas de acordo com o Bitonic Sort. Após isso, a linha 15 verifica se p_i deve assumir as funções de um processo que falhou. Se sim, a função `testIfNeedTakePlace()`, basicamente, faz novo checkpoint de p_i e repete as linhas 11 a 14, porém agora p_i se passa pelo processo falho. Por exemplo, o par de processos 0 e 4 são parceiros naturais quando $t \leftarrow 2$. Se a linha 15 informar que o parceiro não é o natural, por exemplo, 0 e 5, quando $t \leftarrow 2$, então as linhas 17 e 22 são executadas: um dos processos executa as linhas 18 e 19 e o outro as linhas 21 e 22. Por exemplo, o processo 0 realiza as trocas e comparações e após isso verifica se possui mais trabalho na rodada (linhas 18 e 19). Já o processo 5 verifica se possui mais trabalho, que é possivelmente assumir o processo 4, e então realiza sua função natural (linha 22). As linhas 24 e 26 são invocadas quando p_i precisa ele mesmo assumir as tarefas de um processo falho. Nesse caso p_i recupera em disco as informações do parceiro (linhas 24 e 25) e logo após verifica se há mais trabalho a ser realizado (linha 26).

Importante destacar que cada processo possui uma lista com os processos falhos. Para saber com qual processo p_i deve formar par e qual processo substitui o processo falho, a função $c_{i,s}$ [Duarte, Jr. and Nanya 1998] é utilizada. 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 $s = dim$. A Tabela 1 apresenta 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.

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

De acordo com a função, 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 terceira rodada de ordenação (Figura 3) 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. Se p_5 também estiver falho então p_0 e p_6 serão o novo par. Se tanto p_6 e p_7 estiverem falhos então p_0 não troca elementos com nenhum processo nessa rodada de ordenação.

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 todos os processos onde $s = 1$ estão falhos, então s é incrementado até que um processo sem-falha seja encontrado. Considerando a terceira rodada de ordenação (Figura 3) 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 $c_{4,2}$. Importante destacar que o uso da função $c_{i,s}$ permite que as tarefas sejam distribuídas sem sobrecarregar um processo em particular. Se metade dos processos se encontrarem falhos os processos sem-falha realizam as próprias tarefas e cada um mais uma tarefa de um processo falho. Essa carecterística é importante pois o *Bitonic Sort* exige muita comunicação entre os processos.

5. Implementação

O *Bitonic Sort*tolerante a falhas foi implementado[1] em linguagem C usando a biblioteca ULFM 2.0 [MPI-Forum 2019]. A ULFM é uma implementação da biblioteca Open MPI. 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.

A função `MPI_myBarrier()` sincroniza todos os processos através de uma barreira. 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 comunicação original dos processos. 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 empregada para implementar a função `MPI_myBarrier()`.

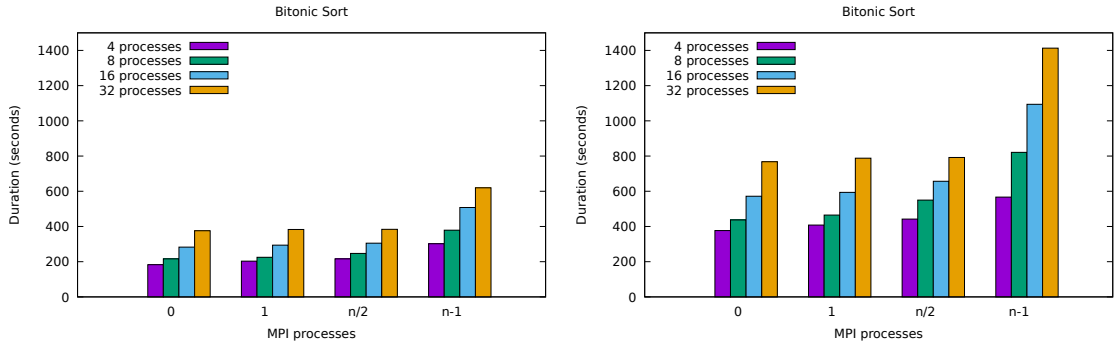
A partir da identificação das falhas, a lista de processo falhos alimenta as funções definidas na Seção 4.1. Uma função para injetar falhas aleatórias também foi definida para realizar os testes. A função `applyFailures()` escolhe quais processos irão falhar em uma rodada de ordenação particular. Foram 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 $n - 1$ processos falham.

6. Resultados

Os experimentos foram executados em uma máquina multiprocessada com 32 processadores *Intel Core i7*. O sistema operacional é o *Linux Kernel 4.4.0*. Cada experimento foi executado 10 vezes e a média do tempo de execução é apresentada. O tempo da execução, em segundos, foi extraído através do comando `time` do Linux. Dessa forma, tanto a leitura quanto a escrita de dados em disco fazem parte do tempo de execução. Importante destacar que para este trabalho o objetivo não é apresentar o *speedup*, mas a capacidade do algoritmo de se manter em execução mesmo perante falhas.

A Figura 4 apresenta o tempo de execução em segundos (eixo y) para ordendar 2^{29} (Figura 4a) e 2^{30} números inteiros (Figura 4b). Os testes foram executados com 4, 8, 16 e 32 processos MPI de acordo com a legenda apresentada no gráfico. Os experimentos consideraram o ambiente sem-falhas, 1 falha, $n/2$ falhas e $n - 1$ falhas (eixo x). Como

[1] `https://bitbucket.org/WorstOne/parallel-bitonic-sort-ulfm-crash/src`



(a) 2^{29} **números inteiros** **(b)** 2^{30} **números inteiros**

Figura 4. Desempenho do *Bitonic Sort* tolerante a falhas.

descrito anteriormente, as falhas são injetadas aleatoriamente, ou seja, podem acontecer no início ou no fim do processo de ordenação.

Como é possível observar, o comportamento do gráfico para ordenar 2^{29} e 2^{30} números inteiros é semelhante. De fato, o tempo de execução quase dobra quando são considerados 2^{30} números. Por exemplo, o tempo de execução com 4 processos em um ambiente sem-falha e com n/2 falhas para 2^{29} é 183s e 216s. Já para 2^{30} o tempo de execução é 377s e 442s. O tempo de execução ainda cresce progressivamente conforme mais falhas são inseridas. No cenário com 0, 1, $n/2$ e $n-1$ falhas para 8 processos e 2^{30} números os tempos de execução são 437s, 465s, 550s e 821s, respectivamente.

É possível ainda observar que o aumento no número de processos para ordenar uma entrada não diminuiu o tempo de execução. O cenário sem-falhas tanto para 2^{29} quanto para 2^{30} mostra que o tempo de execução aumenta conforme cresce o número de processos MPI envolvidos na comunicação. Isso também é observado nos demais cenários. Possivelmente isso se deve ao fato de o *Bitonic Sort* ter como característica a troca intensiva de informações entre processos. Conforme aumenta o número de processos mais rodadas de ordenação são necessárias e, consequentemente, mais trocas de mensagens entre os processos. Conforme descreve [Lan and Mohamed 1992], uma das desvantagens do *Bitonic Sort* é a sua previsibilidade: as operações de junção (*merging*) e troca tomam mais e mais tempo conforme o tamanho dos subcubos aumenta.

7. Conclusão

Este trabalho apresentou uma implementação MPI do algoritmo *Bitonic Sort* que é capaz de continuar a ordenação mesmo se todos menos um processo falharem durante a ordenação. A solução aproveita as propriedades do hipercubo: um processo sem-falha que pertence ao menor subcubo do processo falho assume as tarefas do processo falho em uma rodada de ordenação. A implementação MPI faz uso da especificação ULFM, escrita pelo MPI-Fórum para lidar com falhas no ambiente MPI. Através das rotinas fornecidas pela especificação, a solução proposta identifica quais processos estão falhos em uma rodada, recupera a capacidade do comunicador MPI ao excluir o processo falho e realiza o mapeamento dos processos falhos para processos sem-falha. Resultados de desempenho mostram a eficiência da implementação para ordenar entradas de 2^{29} e 2^{30} números em 4, 8, 16 e 32 processos MPI. Os experimentos foram realizados considerando 0, 1, $n/2$ e $n-1$ falhas.

Foi possível observar que a estratégia empregada na implementação do algoritmo *Hyperquicksort* também é válida para o algoritmo *Bitonic Sort*. Como trabalhos futuros é possível continuar desenvolvendo soluções de tolerância a falhas em MPI para algoritmos de ordenação paralelos que executam no hipercubo ou em qualquer outra topologia. Também é possível desenvolver uma biblioteca para automatizar a tolerância a falhas em aplicações MPI que empregam a topologia do hipercubo como, por exemplo, os algoritmos de ordenação. Além disso, após a otimização do código, será possível comparar o desempenho de diferentes algoritmos em cenários com falhas. As falhas foram inseridas sempre no início de uma rodada, outra possibilidade é programar a solução para suportar a falha durante uma rodada.

Referências

Al-Hashimi, M. A., Abulnaja, O. A., Saleh, M. E., and Ikram, M. J. (2017). Evaluating power and energy efficiency of bitonic mergesort on graphics processing unit. *IEEE Access*, 5:16429–16440.

Batcher, K. E. (1968). Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA. ACM.

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 *SBRC 2016 - WTF*.

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 WSCAD 2017*, page 30.

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

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

Chen, R., Siriyal, S., and Prasanna, V. (2015). Energy and memory efficient mapping of bitonic sorting on fpga. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 240–249, New York, NY, USA. ACM.

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

Durad, M. H., Akhtar, M. N., and ul Haq, I. (2014). Performance analysis of parallel sorting algorithms using mpi. In *2014 12th International Conference on Frontiers of Information Technology*, pages 202–207.

Elnozahy, Alvisi, Wang, and Johnson (2002). A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surveys*, 34.

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

Forum, M. (2019a). Mpi forum website. `http://mpi-forum.org/`. Acessado em 26/01/2019.

Forum, M. (2019b). User-level failure mitigation. `http://fault-tolerance.org/ulfm/ulfm-specification/`. Acessado em 26/02/2019.

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.

Johnson, S. L. (1984). Combining parallel and sequential sorting on a boolean n-cube. In *Proceedings of the 1984 International Conference on Parallel Processing*, pages 444–448.

Kumar, V. (2002). *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition.

Lan, Y. and Mohamed, M. A. (1992). Parallel quicksort in hypercubes. In *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing: Technological Challenges of the 1990's*, SAC '92, pages 740–746, New York, NY, USA. ACM.

Lee, D. and Batcher, K. E. (1994). On sorting multiple bitonic sequences. In *1994 International Conference on Parallel Processing Vol. 1*, volume 1, pages 121–125.

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.

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

Nassimi, D. and Sahni, S. (1979). Bitonic sort on a mesh-connected parallel computer. *IEEE Transactions on Computers*, C-28(1):2–7.

Pacheco, P. S. (1997). *Parallel programming with MPI*. Morgan Kaufmann.

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

Sheu, J.-P., Chen, Y.-S., and Chang, C.-Y. (1992). Fault-tolerant sorting algorithm on hypercube multicomputers. *J. Parallel Distrib. Comput.*, 16:185–198.

Velusamy, K., Rolinger, T. B., McMahon, J., and Simon, T. A. (2018). Exploring parallel bitonic sort on a migratory thread architecture. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7.