MECANISMO DE TOLERÂNCIA A FALHAS TRANSIENTES PARA APLICAÇÕES BAG-OF-TASKS EM AMBIENTES DE ALTO DESEMPENHO

Rodrigo Coacci

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: Diego Leonel Cadette Dutra
Claudio Luis de Amorim

Rio de Janeiro
Fevereiro de 2025

MECANISMO DE TOLERÂNCIA A FALHAS TRANSIENTES PARA APLICAÇÕES BAG-OF-TASKS EM AMBIENTES DE ALTO DESEMPENHO

Rodrigo Coacci

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Orientadores: Diego Leonel Cadette Dutra
Claudio Luis de Amorim

Aprovada por: Prof.ª Priscila Machado Vieira Lima
Prof.ª Maria Clicia Stelling de Castro
Prof. Eugene Francis Vinod Rebello

RIO DE JANEIRO, RJ – BRASIL
FEVEREIRO DE 2025

Coacci, Rodrigo

Mecanismo de Tolerância a Falhas Transientes para Aplicações Bag-Of-Tasks em Ambientes de Alto Desempenho/Rodrigo Coacci. – Rio de Janeiro: UFRJ/COPPE, 2025.

XIII, 48 p.: il.; 29, 7cm.

Orientadores: Diego Leonel Cadette Dutra
Claudio Luis de Amorim

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2025.

Referências Bibliográficas: p. 43 – 47.

1. tolerância a falhas. 2. mestre-trabalhador. 3. replicação de tarefas. 4. especulação de tarefas. 5. falha transiente. I. Leonel Cadette Dutra, Diego *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Dedico este trabalho a meus pais,
cujos esforços incalculáveis
culminaram nesta conquista.

Agradecimentos

Agradeço à Petróleo Brasileiro S.A., na figura de seus gerentes Jonilton e Milena que me deram a oportunidade de realizar este trabalho com total dedicação e suporte financeiro.

Também agradeço ao coordenador Elton, por sua ajuda e suporte na obtenção das autorizações necessárias junto à Companhia e demais formalidades administrativas em conjunto com o colega Ricardo.

Agradeço a todos os colegas da gerência de Tecnologia Geofísica da Petrobrás, pelo total apoio na realização deste mestrado. Agradeço especialmente a Carlos Cunha, Daniel Thomé e Fernanda Thedy que me incentivaram a fazer este mestrado e ao Bruno Pereira Dias pelos dados de teste e ajuda com a aplicação usada nos experimentos.

Agradeço ao Alan Albano, Alexandre Sardinha e Nelson Hargreaves pelas inestimáveis conversas e sugestões que muito me ajudaram a realizar este trabalho. Agradeço também o importante trabalho da equipe de Infraestrutura que cuida dos clusters da Petrobrás por toda sua ajuda e apoio.

Agradeço especialmente aos Professores Claudio Luis de Amorim e Diego Leonel Cadette Dutra, por terem me aceito no Programa de Engenharia de Sistemas e Computação da COPPE e por todos os ensinamentos e orientações nestes 3 anos de curso.

Finalmente, agradeço à minha esposa Verônica e às minhas filhas Amanda e Júlia pela companhia e apoio durante esta jornada.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

MECANISMO DE TOLERÂNCIA A FALHAS TRANSIENTES PARA APLICAÇÕES BAG-OF-TASKS EM AMBIENTES DE ALTO DESEMPENHO

Rodrigo Coacci

Fevereiro/2025

Orientadores: Diego Leonel Cadette Dutra
Claudio Luis de Amorim

Programa: Engenharia de Sistemas e Computação

As falhas aumentaram significativamente nos sistemas de computação de alto desempenho recentes, tornando a tolerância a falhas crucial para a eficiência e confiabilidade das aplicações. Falhas transitórias podem gerar atrasos consideráveis na execução das tarefas, comprometendo a eficiência do sistema. Soluções tradicionais podem ser ineficazes no tratamento de falhas frequentes, gerando uma sobrecarga significativa.

Este trabalho propõe e avalia a replicação parcial de tarefas com especulação em um *framework* genérico de mestre-trabalhador para mitigar os efeitos de falhas transitórias em ambientes de computação de alto desempenho (HPC). A proposta combina replicação parcial de tarefas com especulação para explorar o paralelismo inerente e reduzir o tempo ocioso em caso de falhas. A solução é implementada em um *framework* genérico e avaliada em dois programas paralelos, um real e outro sintético, em um ambiente HPC.

Os resultados dos experimentos mostram que a especulação de tarefas pode reduzir significativamente a variância dos tempos de execução das tarefas na presença de falhas transitórias, diminuindo o tempo total de execução em até 4 vezes e aumentando a previsibilidade do tempo de execução.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

TRANSIENT FAULT TOLERANCE MECHANISM FOR BAG-OF-TASKS APPLICATIONS IN HIGH-PERFORMANCE ENVIRONMENTS

Rodrigo Coacci

February/2025

Advisors: Diego Leonel Cadette Dutra
Claudio Luis de Amorim

Department: Systems Engineering and Computer Science

The failures have increased significantly in recent high-performance computing systems, making fault tolerance crucial for application efficiency and reliability. Transient faults can generate considerable delays in task execution, compromising system efficiency. Traditional solutions may be inefficient in dealing with frequent failures, generating significant overhead.

This work proposes and evaluates partial task replication with speculation in a generic master-worker framework to mitigate the effects of transient failures in high-performance computing (HPC) environments.

The proposal combines partial task replication with speculation to exploit inherent parallelism and reduce idle time in case of failures. The solution is implemented in a generic framework and evaluated on two parallel programs, one real and one synthetic, in an HPC environment. The results of the experiments show that task speculation can significantly reduce the variance of task execution times in the presence of transient failures, decreasing the total execution time by up to 4 times and increasing the predictability of the execution time.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Listagens

Lista de Abreviaturas

AMT	Asynchronous Many-Task	40
API	Application Programming Interface	3
CPU	Central Processing Unit	18
DAG	Directed Acyclic Graph	40
E/S	Entrada/Saída	19
HPC	High Performance Computing	1
MPI	Message Passing Interface	2
MTTF	Mean Time To Failure	2
RAID	Redundant Array of Independent Disks	7
RAM	Random Access Memory	18
TCO	Total Cost of Ownership	22

Capítulo 1

Introdução

1.1 Contexto

Em sistemas de computação de alto desempenho (do inglês *High Performance Computing* – HPC), um dos modelos de programação utilizados é o modelo mestre-trabalhador. Esse modelo organiza os processos paralelos em um processo mestre e vários processos trabalhadores. Por simplicidade, neste trabalho o termo processo será implícito quando se utilizar os termos mestre e trabalhador. Todo o trabalho a ser realizado por uma aplicação que utiliza este modelo é divido em tarefas, o mestre define quais tarefas serão executadas, seleciona quais trabalhadores devem realizá-las e em que ordem, finalmente enviando as tarefas e seus requisitos para os trabalhadores. Em muitos casos, os trabalhadores precisam enviar respostas ao mestre para subsequente processamento.

Neste tipo de aplicação, o tempo total de execução é fortemente influenciado pela tarefa mais lenta, e qualquer desbalanceamento nas tarefas ou lentidão causada por fatores externos pode reduzir drasticamente os benefícios obtidos pela execução paralela. Tais ocorrências são conhecidas como tarefas atrasadas (do inglês *straggler tasks*) [1, 2].

Assumindo a inexistência de desbalanceamento, diversas razões podem causar variabilidade nos tempos de execução das tarefas: compartilhamento de recursos globais ou locais, atividades de manutenção em segundo plano, falhas de software ou hardware, entre outras [3], podendo ser temporárias e intermitentes. Com o aumento da escala dos supercomputadores e sistemas distribuídos em geral, cresce a probabilidade de ocorrência de problemas, sejam eles intermitentes ou não. Estima-se que em sistemas *Exascale* o tempo médio até a falha (*mean time to failure* – MTTF) pode atingir o intervalo de um minuto, tornando impraticável o uso de *checkpoint/restart* [4]. Além disso, falhas transientes e erros silenciosos se tornarão cada vez mais frequentes [5]. Devido à complexidade inerente às razões da varia-

bilidade nos tempos de execução das tarefas, exemplificadas anteriormente, neste trabalho todas as causas externas de variabilidade são consideradas falhas transientes.

Nesse contexto, o uso da replicação de tarefas é uma alternativa importante para aprimorar a tolerância a falhas. Embora em aplicações baseadas em passagem de mensagens (MPI) a replicação de tarefas ainda apresente desafios, como a manutenção da consistência das mensagens entre as réplicas [5], a replicação é amplamente adotada em aplicações do tipo mestre-trabalhador e *fork-join*. Dean and Ghemawat [1] demonstraram a eficácia da replicação para tolerância a falhas no MapReduce, um *framework* implementando com o modelo mestre-trabalhador, enquanto Xu *et al.* [6] e Zhang *et al.* [7] exploraram seu uso para aprimorar a tolerância a falhas e reduzir a latência.

1.2 Motivação

Em ambientes de computação de alto desempenho, é comum que os recursos computacionais fiquem alocados exclusivamente para o aplicação paralela até que este termine completamente seu processamento. Isto significa que qualquer aumento significativo no tempo de execução de qualquer tarefa em um aplicação mestre-trabalhador pode causar subutilização dos recursos computacionais devido à necessidade de que todas as tarefas terminem com sucesso para conclusão do aplicação. Assim a presença de falhas transientes e tarefas atrasadas pode comprometer significativamente a eficiência global. Com o avanço da escala dos sistemas HPC, a probabilidade desses eventos aumenta, tornando crucial o aprimoramento da tolerância a falhas transientes.

Um caso concreto de uma aplicação real de migração sísmica com tarefas atrasadas em um supercomputador, utilizada em uma grande empresa do setor de óleo e gás, ilustra a motivação para este trabalho. A aplicação, implementada originalmente usando MPI para distribuição de tarefas, apresenta tempos de execução que variam de alguns minutos a algumas horas, com os mesmo dados de entrada e parâmetros, sem que fosse possível identificar uma causa única e específica. Esta elevada variabilidade gera enormes impactos para a empresa como a falta de previsibilidade nos prazos, atrasos nas execuções de outras aplicações e redução da eficiência energética e financeira do supercomputador. Partindo-se da premissa que esses atrasos seriam causados por falhas transientes, propõe-se para esta aplicação a substituição do MPI por um outro *framework* de distribuição de tarefas em conjunto com uma solução de replicação de tarefas para mitigar os efeitos destas falhas transientes.

A replicação parcial de tarefas tem apresentado bons resultados como mecanismo de tolerância a falhas, particularmente para falhas transientes, tendo sido implemen-

tada com sucesso em sistemas como *MapReduce*, cujos autores estimam uma redução na latência em até 66% [1], e Hadoop, onde se obteve melhoria de até 2 vezes nos tempos de resposta [8]. Finalmente, é possível reduzir o tempo de computação em até uma ordem de grandeza dependendo do nível de replicação utilizado [9].

O principal padrão de comunicação utilizado em sistemas de computação de alto desempenho é o padrão MPI, que propõe um modelo de comunicação entre processos e uma API (do inglês *Application Programming Interface*) flexíveis e genéricos, sem impor nenhum modelo de computação específico, como mestre-trabalhador. Qualquer replicação ou especulação de tarefas precisa ser implementada diretamente pelo programador da aplicação, ou em algum *framework* implementado sobre o MPI.

A maior dificuldade em se utilizar replicação em MPI é a natureza estática do conjunto de processos controlados pelo MPI, e a ausência de tolerância a falhas em processos, inexistente tanto no padrão MPI quanto na maioria das implementações [10]. Adicionalmente, algumas classes de aplicações implementadas com o paradigma mestre-trabalhador tem pouca necessidade de comunicação e sincronização, tornando o uso de MPI pouco atrativo quando considerados os seus custos, como a complexidade do modelo de programação baseado em passagem de mensagens. Por outro lado, *frameworks* não baseados em MPI podem suportar conjuntos de processos dinâmicos e tolerância a falhas de processos, tornando a implementação da replicação e especulação de tarefas uma extensão natural.

1.3 Objetivos e Contribuições

O presente trabalho propõe e avalia a implementação de replicação parcial de tarefas utilizando especulação em um *framework* mestre-trabalhador genérico, não baseado em MPI, que não siga o modelo *MapReduce* e seja usado em ambientes de computação de alto desempenho científicos, com o objetivo de verificar sua eficácia em mitigar os efeitos as falhas transientes.

As principais contribuições deste trabalho são:

- Descrição de alguns dos principais problemas existentes em ambientes de computação de alto desempenho quando estes ambientes apresentam falhas. É dado foco especial para as falhas transientes e o problema das tarefas atrasadas;
- Discussão de algumas das soluções existentes para a tolerância a falhas transientes, em particular a replicação e especulação de tarefas, bem como suas aplicações no contexto de computação de alto desempenho;
- Desenvolvimento e avaliação de uma implementação de replicação parcial, utilizando especulação de tarefas, com sua aplicação em duas aplicações paralelas

mestre-trabalhador, uma real e uma sintética, em um ambiente de computação de alto desempenho em produção;

- Comparação da proposta implementada com uma solução pré-existente para o tratamento de tarefas atrasadas.

1.4 Organização

No Capítulo 2 são introduzidos os principais conceitos necessários para a compreensão desta dissertação, como o modelo mestre-trabalhador, as definições de erros, falhas e tolerância a falhas, replicação e especulação de tarefas, e é apresentado o *framework* mestre-trabalhador utilizado como base para a implementação da especulação de tarefas. No Capítulo 3 é detalhada a proposta de solução e sua implementação. A descrição dos experimentos realizados e a discussão dos resultados podem ser encontradas no Capítulo 4. No Capítulo 5 são discutidos os trabalhos relacionados com o conteúdo desta dissertação. Finalmente no Capítulo 6 são resumidos os principais resultados e apresentadas as conclusões, com as propostas de trabalhos futuros.

Capítulo 2

Conceitos básicos

O objetivo deste capítulo é apresentar os principais conceitos e definições usados neste trabalho. É apresentado também o *framework* Work Queue que é utilizado como base para a proposta de solução.

2.1 Modelo mestre-trabalhador

O modelo mestre-trabalhador é um paradigma de programação amplamente utilizado em aplicações de computação de alto desempenho. Caracteriza-se por uma estrutura hierárquica, centralizando o controle do processamento das tarefas em um único processo denominado mestre. Este mestre é responsável por definir as tarefas a serem executadas, sua ordem de execução e os trabalhadores associados. Os trabalhadores, por sua vez, são processos que executam as tarefas e comunicam-se com o mestre para solicitar tarefas e reportar resultados.

O modelo mestre-trabalhador é popular em ambientes de computação de alto desempenho devido à sua eficiência ao lidar com problemas computacionais intensivos. Ele distribui a carga de trabalho entre vários processadores de forma eficiente, com o mestre atuando como coordenador para facilitar a execução paralela e garantir a sincronização geral da computação. A natureza descentralizada dos processos trabalhadores permite execução simultânea e independente, otimizando o desempenho global. O fraco acoplamento possibilita a inclusão ou remoção dinâmica de trabalhadores, maximizando a utilização dos recursos e a tolerância a falhas nos trabalhadores. A estrutura hierárquica do modelo também facilita a implementação de algoritmos com estruturas semelhantes, como dividir-e-conquistar e árvores de busca.

No entanto, o principal desafio reside na vulnerabilidade à falha do mestre. Sobrecarga, erros de execução ou perda de comunicação com os trabalhadores podem exigir reinicialização da execução. Esses problemas podem ser atenuados com mestres redundantes, distribuindo cooperativamente a carga de trabalho, ou uma hi-

erarquia de mestres que atuam como trabalhadores para o nível superior e como mestres para o nível inferior, reduzindo recursivamente a carga.

2.2 Insucessos, Erros e Falhas

Neste trabalho são utilizadas as definições de insucesso (do inglês *failure*), erro (do inglês *error*) e falha (do inglês *fault*) propostas por Avizienis *et al.* [11].

Insucessos são eventos em que um programa ou sistema passa a funcionar de maneira incorreta, apresentando um resultado incorreto ou tempo de resposta superior ao esperado. Um erro é um estado (ou parte dele) incorreto de um programa ou sistema. Por fim, uma falha é a causa, hipotética ou não, do erro. Assim uma falha (ex.: cabo de rede quebrado) desencadeia um erro (comunicação de rede perdida), que por sua vez origina um insucesso (o programa espera pela resposta indefinidamente). Não é sempre que falhas causam erros, e nem sempre erros causam insucessos.

Existem diversas categorias de insucessos, erros e falhas, sendo o foco deste trabalho uma categoria bastante específica: as falhas transientes.

2.2.1 Falhas transientes

Falhas transientes são falhas que ocorrem de forma intermitente e por um tempo finito, geralmente curto. São difíceis de diagnosticar devido à sua natureza e podem passar despercebidas quando causam apenas degradação no sistema, geralmente observada como lentidão na execução das operações sem causar erros ou insucessos. Se sinalizadas ou se causarem um erro, como um erro na leitura em um arquivo, devido a uma falha de *hardware*, o sistema pode tentar a operação novamente. Se a nova tentativa for bem-sucedida, caracteriza a primeira ocorrência como transiente. Em casos mais graves, a falha pode ser comunicada ao usuário ou administrador para correção, se possível. O desafio é maior quando as falhas não são sinalizadas, causando apenas degradação temporária no sistema, sem uma identificação imediata do problema.

Neste trabalho são consideradas falhas transientes as ocorrências externas à aplicação paralela que resultem em aumento na variabilidade dos tempos de execução e degradação no sistema, sem a ocorrência de erros ou insucessos. As degradações são particularmente problemáticas em ambientes de computação de alto desempenho. Dado que os programas frequentemente operam em modo não interativo por longos períodos, torna-se difícil identificar falhas transientes e aplicar correções. Além disso, devido à natureza paralela dos programas nesses ambientes, uma falha transiente em um processo pode causar o surgimento de tarefas atrasadas (do inglês *straggler tasks*) [2, 12, 13], impactando significativamente o término do programa e

a eficiência dos recursos computacionais.

2.3 Tolerância a Falhas

É um meio de se prevenir o insucesso de programas ou sistemas na presença de falhas. Nos sistemas de computação de alto desempenho, os métodos de tolerância a falhas mais empregados, quando aplicáveis, incluem o *checkpoint/restart* e a replicação [5, 14, 15], sendo esta última viabilizada por *hardware* (ex: discos em RAID) ou por *software* (ex.: replicação de processos).

2.3.1 Replicação

A replicação, como método de tolerância a falhas em *software*, consiste em utilizar uma ou mais cópias redundantes do mesmo processo, executando em paralelo. Isso permite que, em caso de erros em uma das cópias, outra possa concluir a computação com sucesso, evitando assim o insucesso do programa paralelo.

O principal desafio dessa abordagem reside na redução da eficiência do aplicação e diminuição da escalabilidade, uma vez que as réplicas consomem recursos que poderiam ser direcionados para aumentar o paralelismo. Como forma de mitigar esse custo, propõe-se a replicação parcial, evidenciada por estudos que demonstram até mesmo melhoria no desempenho dos programas, especialmente na presença de tarefas atrasadas [13, 16–18]. Surge, então, o desafio de identificar quais réplicas devem ser criadas para otimizar a utilização dos recursos computacionais disponíveis sem comprometer a tolerância a falhas.

2.3.2 Especulação de tarefas

A especulação de quais tarefas são propensas a falhas é um critério para a replicação parcial, reduzindo assim os custos da tolerância a falhas. Na presença de falhas intermitentes que resultam em degradação, observam-se melhorias significativas de desempenho ao esperar apenas pela réplica que termina com sucesso mais rapidamente, cancelando as demais [19].

A especulação de tarefas foi amplamente estudada em *frameworks* de nuvem [6, 7, 20], sendo implementada no MapReduce com o nome de "*Backup Tasks*" [1].

Para facilitar a replicação e especulação de tarefas, é importante que as tarefas sejam idempotentes, ou seja, sua execução repetida não deve causar efeitos adicionais externamente visíveis ou produzir resultados diferentes. A falta desta característica não impossibilita a utilização de replicação, mas dificulta sua implementação, pois é necessário tratar os efeitos colaterais de sua execução, por exemplo realizando um *rollback*, ou tratando as diferenças nos resultados.

2.4 Work Queue

O Work Queue é um *framework* flexível voltado à construção de aplicações científicas mestre-trabalhador em larga escala, abrangendo *clusters*, *grids* e nuvens. Diferenciando-se de sistemas tradicionais baseados em MPI, o Work Queue permite a utilização de conjuntos dinâmicos de trabalhadores, possibilitando ajustes conforme a demanda da aplicação. Além disso, incorpora mecanismos de tolerância a falhas, como o tratamento transparente de falhas nos trabalhadores e o cancelamento rápido (*Fast Abort*) de tarefas identificadas como atrasadas [21, 22]. Uma das motivações para escolha deste *framework* é a utilização da linguagem C em sua implementação, o que garante maior eficiência e portabilidade, e permite maior interoperabilidade com outras linguagens de programação bastante utilizadas em ambientes de computação de alto desempenho como Fortran. Outro critério importante para a escolha é a manutenção contínua ao longo do tempo pela sua equipe, além da utilização em diversos sistemas de produção, indicando ser um software estável e perene a longo prazo.

2.4.1 Fast Abort

O mecanismo de *Fast Abort* atua como uma solução para o problema das tarefas atrasadas, identificando aquelas que excedem o tempo médio de execução bem-sucedida. Seu funcionamento é simples: o Work Queue utiliza o tempo médio de execução bem-sucedida e um multiplicador definido pelo usuário para calcular o tempo máximo esperado para a execução de tarefas. Quando uma tarefa ultrapassa esse limite em um trabalhador, o Work Queue presume que o trabalhador está enfrentando problemas e cancela proativamente a execução dessa tarefa, retornando-a à fila para ser realizada por outro trabalhador. Adicionalmente, se um trabalhador for frequentemente identificado como lento em diversas execuções, é colocado em quarentena, ficando sem receber tarefas por um intervalo predefinido [22].

2.4.2 A fila de tarefas

O núcleo do Work Queue é a fila de tarefas mantida no mestre do *framework*. O processamento dessa fila pelo Work Queue começa quando o mestre invoca a rotina `work_queue_wait()`. Esta rotina executa um laço que apenas retorna quando uma tarefa foi concluída (com sucesso ou não), indo para o estado COMPLETO (fig. 2.2), ou quando uma quantidade de tempo opcionalmente especificada na chamada é atingida (*timeout*). Na Figura 2.1 são descritos resumidamente os passos executados nesse laço.

Este laço lida principalmente com três ações: retornar o controle para o usuário

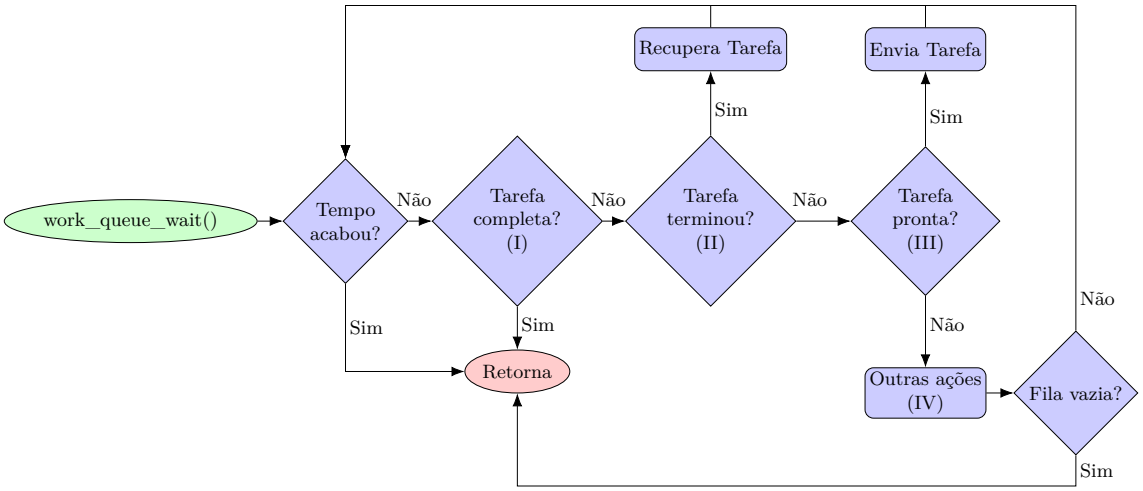

Figura 2.1: Laço principal do Work Queue

informando tarefas terminadas (I), processar o término de tarefas dos trabalhadores (II), e submeter tarefas novas para os trabalhadores (III). Essas ações são consideradas prioritárias no laço de execução sendo a importância delas demonstrada pela ordem delas no fluxo e pelos retornos para o início do laço ou para a saída da rotina. Se nenhuma destas ações é necessária, então é feito o processamento de outras ações menos importantes (IV) como coleta de estatísticas, remoção de trabalhadores perdidos, etc. Finalmente, retorna-se ao início do laço caso existam tarefas aguardando na fila. Do contrário, a execução do laço é terminada levando à saída da rotina.

2.4.3 O ciclo de vida de uma tarefa

Quando uma tarefa é submetida para a fila do Work Queue pelo mestre, ela entra na fila no estado PRONTO, indicando que está pronta para ser enviada para um trabalhador, permanecendo neste estado até que um trabalhador esteja disponível para sua execução. Quando isso ocorre a tarefa é enviada com seus requisitos para o trabalhador e este responde para o mestre que a tarefa iniciou sua execução. Caso haja algum problema durante o envio, a tarefa pode ser retornada para o estado PRONTO para aguardar uma nova tentativa ou diretamente para o estado COMPLETO, sendo marcada como uma tarefa com erro. Se não houver problemas, o mestre então altera suas estruturas internas para refletir a mudança para o estado EXECUTANDO. Neste estado há uma diferença importante caso ocorra um erro: se o problema ocorrer na tarefa em si, isto é, a tarefa retornar com um erro (ex.: código de retorno diferente de 0), a tarefa é marcada como com erro, e segue para o estado RECEBENDO, para que o mestre recupere quaisquer saídas que tenham sido geradas. Já se ocorrer um problema no trabalhador em si ou houver falha de comunicação entre o mestre e o trabalhador, a tarefa é retornada para o estado

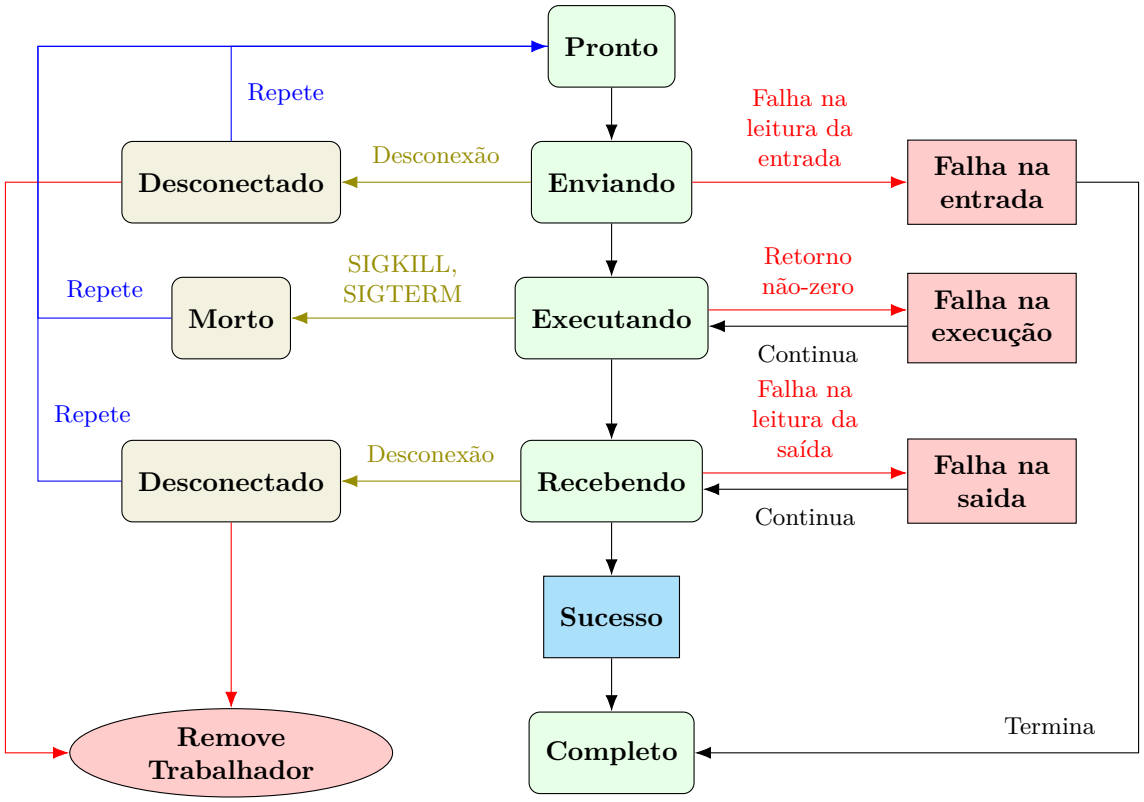

Figura 2.2: Ciclo de vida básico de uma tarefa. Adaptado de [21].

PRONTO para ser repetida. Esta diferenciação é fundamental para permitir a tolerância a falhas, pois o *framework* não tem como determinar a ação correta, por exemplo, no caso de algum parâmetro incorreto, ou se é esperado que aplicação retorne um valor diferente de 0 em algumas situações. Apenas o mestre (e seu programador) tem informações suficientes para determinar a ação correta em cada caso.

Seguindo o fluxo, se a tarefa terminou, com erros ou não, o trabalhador envia ao mestre os resultados da tarefa como arquivos de saída, conteúdo da saída padrão, etc. Neste estado, temos um tratamento semelhante ao do envio das tarefas: problemas podem direcionar a tarefa de volta para o estado de PRONTO para uma nova tentativa, ou serem marcadas como com erro e seguir o fluxo. Não há limite para o número de vezes que uma tarefa retorna para a fila em caso de problemas no trabalhador ou na comunicação entre o mestre e o trabalhador. Independente do seu estado interno de erro ou não, a tarefa é considerada bem-sucedida, pois foi executada até seu término pelo trabalhador, indo para o estado COMPLETO para ser devolvida para o usuário. Esse ciclo é ilustrado na Figura 2.2.

Se durante o ciclo de vida houver uma falha de conexão com um trabalhador, ou o trabalhador falhe em responder às requisições dentro do tempo estipulado (*keepalive*), o trabalhador é marcado como DESCONECTADO e é removido da lista

de trabalhadores disponíveis. Outros detalhes do ciclo de vida podem ser vistos em [21].

Capítulo 3

Especulação de Tarefas no Work Queue

Neste capítulo a implementação de especulação de tarefas no Work Queue é descrita em detalhes, tanto no modo *Backup Tasks* quanto no modo especulativo por tempo.

3.1 Especulação de Tarefas

Neste trabalho são propostos dois modos de especulação de tarefas para o Work Queue:

- Modo *Backup Tasks*: neste modo, inspirado pela implementação do *MapReduce*, quando o sistema identifica que não há mais tarefas na fila aguardando execução e existem trabalhadores disponíveis, especula-se que as tarefas ainda em execução podem ser tarefas atrasadas. Assim uma réplica é criada para cada tarefa e submetida à fila para execução pelos trabalhadores ociosos. A justificativa para essa especulação reside na ideia de que as tarefas são balanceadas e portanto terminam aproximadamente ao mesmo tempo. Se algumas tarefas ainda estão em execução quando a maioria delas termina, é razoável presumir que as tarefas em execução estão atrasadas;

- Modo Especulação por Tempo: neste modo o sistema armazena o tempo de execução das tarefas, definido como o tempo entre o instante em que o trabalhador recebe a tarefa do mestre até o instante em que o trabalhador notifica o mestre que a tarefa foi terminada. Ao observar o término bem sucedido de pelo menos 5 tarefas, o que reflete a implementação do *Fast Abort*, o sistema utiliza os tempos de execução armazenados e calcula a média aritmética de tempo de execução de todas as tarefas bem sucedidas, recalculando a média cada nova tarefa terminada com sucesso. Um fator de tolerância chamado MULTIPLICADOR, sempre maior que 1.0, é aplicado a esta média e o resultado é

chamado de GATILHO. Como exemplo, dado um tempo médio calculado de 30 segundos e um MULTIPLICADOR de 1.5, o GATILHO será de $30 \times 1.5 = 45$ segundos. Considerando que as tarefas sejam balanceadas corretamente e não haja fatores externos de desbalanceamento, é possível especular que as tarefas que ainda não terminaram após o tempo GATILHO devem ser consideradas atrasadas. O sistema então cria réplicas apenas destas tarefas, submetendo-as na fila com maior prioridade que as demais tarefas, de maneira que estas réplicas sejam executadas assim que houver trabalhadores ociosos. O fator de tolerância MULTIPLICADOR deve ser interpretado como representando uma porcentagem acima do tempo de execução médio, desta forma um MULTIPLICADOR de 1.5 indica que as tarefas são consideradas atrasadas quando o tempo de execução ultrapassa 150% do valor médio.

Em ambos os modos é criada uma única réplica de cada tarefa e o sistema utiliza o resultado da tarefa mais rápida do par original-réplica, cancelando a outra tarefa do par. A utilização de múltiplas réplicas adicionaria complexidade à implementação, o que poderia aumentar os custos da replicação, por isto foi decidido que nesta implementação inicial apenas uma réplica seria feita por tarefa. A rotina `work_queue_wait`, descrita na Seção 2.4.2, foi alterada para introduzir duas novas rotinas dentro do laço principal, ilustradas na Figura 3.1. A primeira rotina verifica e cancela réplicas das tarefas que terminam com sucesso (Ia), e a segunda faz o envio de réplicas para a fila de tarefas durante o processamento de outras ações menos importantes (IVa). Isto garante que a replicação não vai atrasar nenhuma das demais ações prioritárias (I, II e III) e causar sobrecarga no mestre quando a especulação estiver ativada. Adicionalmente, ambas rotinas verificam se a especulação foi ativada e retornam imediatamente em caso negativo. Na próxima seção são discutidos os detalhes de cada rotina e demais alterações feitas no Work Queue.

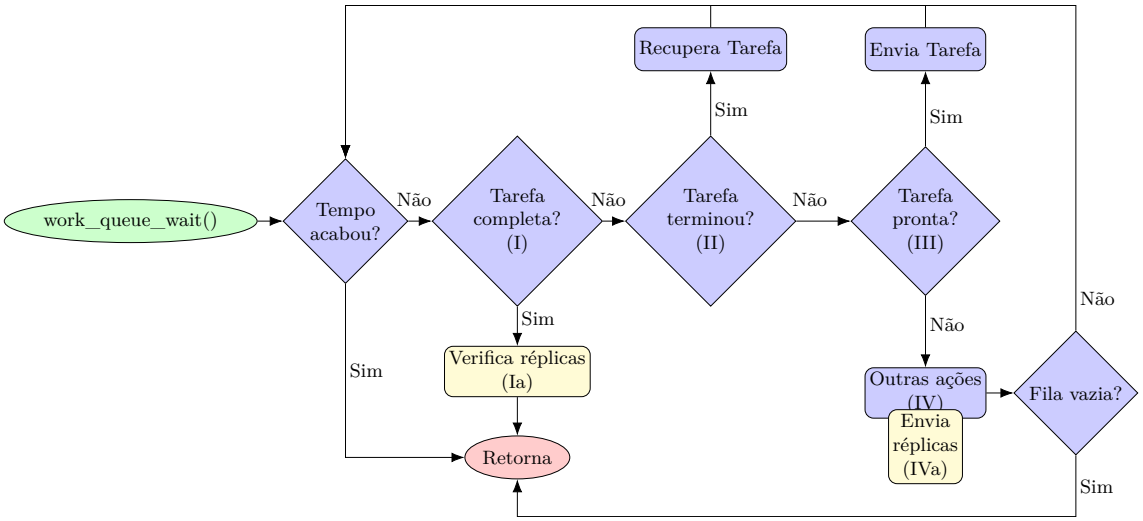

Figura 3.1: Laço principal do Work Queue modificado

3.2 Replicação de tarefas

Para implementar a replicação de tarefas, foram adicionados três campos à estrutura principal que gerencia o estado do Work Queue (Listagem 3.1): uma variável do tipo ponto flutuante de dupla precisão que armazena o MULTIPLICADOR aplicado ao tempo médio de execução para determinar se uma tarefa está atrasada (I), uma variável do tipo inteiro simples para armazenar a prioridade das tarefas replicadas (II) e finalmente uma tabela associativa que guarda a associação entre as tarefas originais e as réplicas (III). Essas alterações foram feitas no arquivo `work_queue/src/work_queue.c`, linhas 247-249. Informações detalhadas sobre como obter os arquivos podem ser encontradas no Apêndice.

```
1  struct work_queue{
2      // Demais campos omitidos
3      double speculative_multiplier; // (I)
4      int speculative_priority;      // (II)
5      struct itable* replicas;       // (III)
6  }
```

Listagem 3.1: Campos adicionados na `struct work_queue` para replicação de tarefas.

Adicionalmente, foi incluída uma nova rotina (Listagem 3.2) na interface pública que configura os dois primeiros parâmetros e ativa a especulação por tempo, o *Backup Tasks* ou desativa qualquer replicação, conforme o valor do parâmetro `multiplicador`. Se o parâmetro for negativo a replicação é desativada, se maior que 1.0 a especulação por tempo é ativada usando o valor do parâmetro como MULTIPLICADOR e se o valor do parâmetro for zero é ativado o modo *Backup Tasks*. A implementação foi feita no arquivo `work_queue/src/work_queue.c` a partir da

linha 6118, e a interface modificada no arquivo `work_queue/src/work_queue.h`, linha 1197.

```
void work_queue_activate_speculation(struct work_queue *q, double multiplier, int priority);
```

Listagem 3.2: Nova rotina incluída para ativação e configuração da especulação.

A prioridade é usada apenas no modo especulação por tempo (i.e. multiplicador maior que 1.0), para permitir maior controle sobre quando as tarefas especulativas são executadas em relação às tarefas normais. Se este valor for menor que a prioridade usada nas tarefas regulares, as réplicas podem demorar muito para serem executadas, eliminando as possíveis vantagens da especulação. Para que a especulação por tempo seja efetiva, a prioridade das réplicas precisa ser maior que a prioridade das tarefas originais, de forma que as réplicas sejam executadas assim que houver recursos livres, e sejam executadas antes de outras tarefas regulares na fila. No modo *Backup Tasks*, a prioridade das réplicas é a menor possível, para que elas sejam executadas apenas quando não houver mais tarefas originais aguardando na fila.

O algoritmo de replicação depende do modo utilizado, se especulação por tempo ou *Backup Tasks*, e foi dividido em 2 rotinas. A primeira, `check_replicas` (Figura 3.2) é executada no momento em que uma tarefa vai para o estado COMPLETO e está pronta para ser retornada para o usuário na rotina `work_queue_wait` (arquivo `work_queue/src/work_queue.c`, linha 7055). A rotina `check_replicas` verifica se existe um mapeamento para a tarefa concluída na tabela de replicação e, se houver, remove este mapeamento. Caso a tarefa concluída tenha terminado com êxito, cancela a outra tarefa associada, removendo-a do sistema. Caso contrário, a tarefa associada continua na fila, para ter oportunidade de terminar com sucesso. A rotina retorna imediatamente no início caso a especulação esteja desligada para evitar processamentos desnecessários no mestre. Sua implementação se encontra no arquivo `work_queue/src/work_queue.c` a partir da linha 4750.

A segunda rotina se chama `submit_replicas` (Figura 3.3) sendo a responsável por criar e submeter novas réplicas para a fila quando a especulação por tempo ou *Backup Tasks* é ativada. Ela é chamada na rotina `work_queue_wait` após a execução das demais rotinas administrativas do Work Queue (linha 7177 do arquivo `work_queue/src/work_queue.c`). Primeiro a rotina verifica se alguma forma de especulação foi ativada, retornando imediatamente caso nenhuma especulação esteja ativada. Isso evita qualquer sobrecarga no mestre quando a especulação não for usada. Sua implementação reside nas linhas 4776 até 4819 do arquivo `work_queue/src/work_queue.c`.

Em seguida é verificado se as condições para ativação da especulação foram

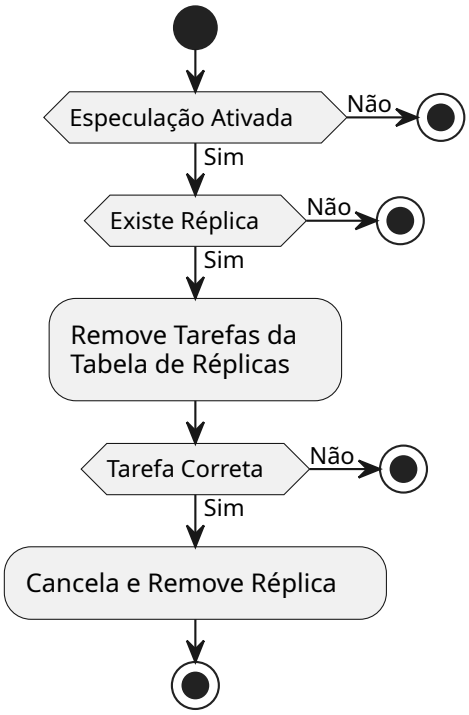

Figura 3.2: Algoritmo de verificação das réplicas.

satisfeitas: pelo menos 5 tarefas foram concluídas e, no caso do *Backup Tasks*, se há recursos ociosos. Caso as condições tenham sido atendidas, calcula o tempo médio de execução das tarefas e o GATILHO da replicação. Em seguida a lista de tarefas originais em execução é percorrida, verificando se cada tarefa excedeu o GATILHO e submetendo uma nova réplica em caso positivo, desde que já não haja outra réplica submetida previamente. Caso a réplica seja submetida, é incluído na tabela de réplicas um mapeamento bidirecional entre a réplica e a original. A implementação usa o fato de o MULTIPLICADOR ter valor zero no modo *Backup Tasks* para que o GATILHO seja sempre zero, implicando que todas as tarefas em execução excedem o GATILHO e terão réplicas submetidas se as demais condições forem satisfeitas. A prioridade forçadamente inferior no modo *Backup Tasks* vai garantir que as réplicas fiquem no final da fila, mesmo que tarefas originais retornem à fila ou novas tarefas originais sejam incluídas.

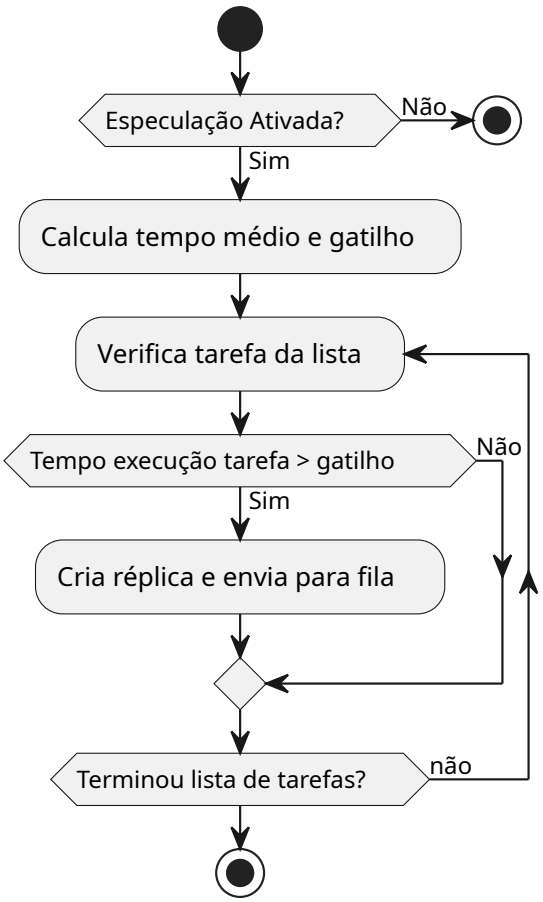

Figura 3.3: Algoritmo de submissão das réplicas.

Capítulo 4

Experimentos e Resultados

Neste capítulo são apresentados os experimentos realizados com Work Queue usando a implementação de especulação de tarefas descrita anteriormente. Os experimentos foram realizados no supercomputador Atlas da Petróleo Brasileiro S.A. e consistem de uma aplicação sintética originalmente criada e descrita pela equipe do Work Queue e uma aplicação da área de óleo e gás.

4.1 Ambiente experimental

O supercomputador Atlas [23, 24] tem 136 nós, onde cada nó consiste em 2 processadores Intel(R) Xeon(R) Gold 6240 CPU 2.60GHz com 18 núcleos cada, 768 GB de memória RAM, 8 aceleradores NVIDIA V100 32GB, disco local com 3,4 Tb para *scratch* das aplicações, rede de interconexão Infiniband EDR 100 Gbps e sistema operacional CentOS 7 com o gerenciador de recursos e agendador de trabalhos *Slurm* [25] na versão 20.11.9. É disponibilizado um sistema de arquivos compartilhado Lustre [26] dedicado ao supercomputador, conectado a este via Infiniband, e compartilhado entre os nós. Os nós de computação são distribuídos igualmente em 17 bastidores, resultando em 8 nós por bastidor e compartilhando um *switch* Infiniband de interconexão. Neste trabalho é também utilizado o termo *cluster* para se referir ao supercomputador utilizado nos experimentos.

4.2 Aplicações experimentais

Para demonstrar a aplicabilidade da nossa proposta em cenários reais da indústria de óleo e gás, onde a migração sísmica é um problema comum [27], implementamos e avaliamos a proposta em uma aplicação de migração sísmica de alto desempenho chamada Least-Squares Migration with Hessian Filter (LSM_HF). Adicionalmente a proposta foi avaliada no Wavefront, uma aplicação de exemplo e *benchmark* do

Work Queue, para verificar que a solução não introduz regressões no *framework*. A seguir, cada aplicação é explicada brevemente, destacando as modificações feitas para os experimentos a fim de alinhar com nosso foco de pesquisa em processamento paralelo e tolerância a falhas usando replicação de tarefas.

4.2.1 Least-Squares Migration with Hessian Filter - LSM_-HF

A aplicação LSM_HF realiza uma migração sísmica utilizando o método de mínimos quadrados com filtro Hessiano [28], sendo este algoritmo específico massivamente paralelo e intenso em E/S. A implementação original usa MPI apenas para distribuição de tarefas em um modelo *bag-of-tasks*. Para este trabalho, a aplicação foi executada com dados baseados em um modelo sísmico sintético, comumente utilizados para validação do algoritmo de migração. Os dados são lidos do sistema de arquivos compartilhado pelos trabalhadores durante a execução. Os resultados também são escritos no mesmo sistema de arquivos compartilhado. Esta aplicação foi escolhida para os experimentos por ter apresentado tarefas atrasadas em diversas ocasiões, sem que uma causa interna à aplicação tenha sido identificada.

Foi feita a adaptação do código original que utilizava um *framework* interno de gerenciamento de tarefas baseado em MPI para a utilização do Work Queue, sendo todas as modificações efetuadas apenas nas rotinas de controle da aplicação que lidam com o gerenciamento de tarefas. Não foi feita nenhuma modificação no núcleo da aplicação e a mesma não utiliza nenhuma rotina MPI em seu núcleo, o que tornou a aplicação efetivamente independente do MPI.

Os *scripts* invocadores do aplicação, que são usados durante a submissão no agendador de trabalhos foram também modificados para invocar o aplicação `work_queue_worker` que age como trabalhador do Work Queue e o aplicação LSM_HF é invocado em modo mestre através de um parâmetro de linha de comando em apenas uma invocação. O aplicação em modo mestre lê um arquivo extra que contém os parâmetros específicos do Work Queue relacionados à replicação de tarefas, e configura o *framework* de acordo. O mestre então constrói as tarefas como invocações do aplicação LSM_HF em modo trabalhador, informando os parâmetros específicos de cada tarefa via linha de comando. As tarefas são então submetidas aos trabalhadores e o mestre entra em um laço de espera ocupada enquanto aguarda a realização das tarefas.

4.2.2 Wavefront

O Wavefront é uma abstração para computação de relações de recorrência em duas ou mais dimensões. Essa abstração pode representar a simulação de problemas que

ocorrem geralmente em economia, teoria de jogos e genômica. O Wavefront foi escolhido como a segunda aplicação experimental por ser uma das aplicações de referência do Work Queue, o que nos permite verificar que não houve regressões no *framework* causadas pela implementação da proposta e também por ser uma das aplicações utilizadas no artigo onde o *Fast Abort* foi introduzido. Na implementação de Wavefront disponibilizada pelos autores[1], a função de exemplo computa o equilíbrio de Nash em um jogo, com dados suficientes para um problema de tamanho 500x500 [22].

Yu *et al.* [22] utilizaram 180 CPUs para executar um Wavefront de tamanho 1000x1000, com uma função F não informada. Assim, para aproximar ao máximo o artigo original utilizando os dados disponíveis, foi executado o Wavefront em um problema de tamanho 500x500 com 90 CPUs nas configurações de 4, 8 e 16 nós, dividindo os 91 processos (90 trabalhadores e 1 mestre) aproximadamente igualmente entre os nós. No Wavefront, os dados são lidos do sistema de arquivos compartilhado pelo mestre e enviados para os trabalhadores como parte de cada tarefa.

Foi efetuada uma única modificação no aplicação original: inclusão de opções para parametrizar a replicação de tarefas. Essa modificação foi estritamente necessária para realização do experimento, uma vez que o código original do Work Queue não contempla nenhuma forma de replicação de tarefas nativamente.

Foi desenvolvido um *script* para invocação do aplicação mestre `wavefront_manager` e do aplicação `work_queue_worker` para os trabalhadores através do agendador de tarefas. Este *script* configura os trabalhadores para usarem uma CPU cada e invoca o aplicação mestre e os demais trabalhadores corretamente nos nós.

4.3 Configurações experimentais

Nas Tabelas 4.1 e 4.2 são apresentadas as configurações experimentais usadas em ambas aplicações. A LSM_HF tem um requisito de memória elevado em relação aos nós do *cluster*, não havendo memória suficiente para mais de 1 trabalhador por nó, sendo um nó compartilhado por um trabalhador e o mestre. Já a Wavefront usa um número total de trabalhadores fixo, de forma que em cada configuração o número de trabalhadores por nó varia. Cada execução individual de uma das aplicações é chamada de execução, e cada execução é composto de várias de tarefas que serão executadas pelos trabalhadores. Cada trabalhador executa uma tarefa de cada vez. O número total de tarefas por execução é pré-determinada pelos parâmetros e dados de entrada utilizados nas aplicações, sendo 249001 tarefas no Wavefront, distribuídas de acordo com um grafo acíclico direcionado, e 25 tarefas no LSM_HF, distribuídas usando o modelo *bag-of-tasks*, em todas as configurações. Apesar

[1] https://github.com/cooperative-computing-lab/work-queue-examples/

de os nós serem sempre alocados de forma exclusiva pelo agendador de trabalhos (*Slurm*), existem diversos processos administrativos e de monitoração sendo executados constantemente nos nós, além de outras aplicações em execução nos demais nós do *cluster*.

Tabela 4.1: Configurações experimentais: processos por nó. O formato `AxB` significa A processos em B nós.

Número de Nós	Processos no Wavefront	Processos na LSM_HF
4	23x3, 22x1	2x1, 1x3
8	12x3, 11x5	2x1, 1x7
16	6x11, 5x5	2x1, 1x15

Tabela 4.2: Configurações experimentais: tipo de execução.

Tipo de execução	Multiplicador	Prioridade	Abreviação
Referência	N/A	0	`baseline`
Backup Tasks	0.0	`INT_MIN`	`backup`
Especulativa 1.4	1.4	5	`spec_1.4`
Especulativa 1.5	1.5	5	`spec_1.5`
Especulativa 1.6	1.6	5	`spec_1.6`
Fast Abort 1.4	1.4	0	`fa_1.4`
Fast Abort 1.5	1.5	0	`fa_1.5`
Fast Abort 1.6	1.6	0	`fa_1.6`

Os tipos de execuções são explicados a seguir:

1. Referência: Execução usando Work Queue na sua configuração padrão, isto é, sem nenhum tipo de execução especulativa, ou *Fast Abort*. Neste trabalho, ela também é chamada de *baseline*. Não há multiplicador a ser especificado;

2. *Backup Task*: Execução usando a implementação de *Backup Tasks* proposta neste trabalho. A prioridade das réplicas é fixada pela implementação, não sendo uma opção de usuário. O valor escolhido neste caso é o menor valor negativo inteiro suportado pela plataforma, dado pela *macro* `INT_MIN` na linguagem de programação C, de forma que nenhuma outra tarefa tenha prioridade inferior à das réplicas. O multiplicador especificado é 0.0 que indica para a implementação que se deseja utilizar *Backup Tasks*;

3. Especulativa X.Y: Execução usando a implementação de Especulação por Tempo proposta neste trabalho. A prioridade das réplicas foi escolhida arbitrariamente como 5, sendo este um valor maior que a prioridade padrão do Work Queue, de forma que as réplicas especulativas tenham maior prioridade

que as demais tarefas originais que ainda não iniciaram e assim sejam executadas em paralelo às tarefas replicadas, conforme a disponibilidade dos recursos. Foram escolhidos como multiplicadores os valores 1.4 ,1.5, e 1.6 para especular tarefas que demorem, respectivamente 40%, 50% e 60% acima do tempo médio observado;

4. *Fast Abort* X.Y: Execução usando a implementação de *Fast Abort* original do Work Queue. De forma semelhante à execução especulativa, foram escolhidos os valores 1.4, 1.5 e 1.6 para os multiplicadores;

Em todos os modos a prioridade das tarefas originais não foi especificada, sendo utilizado o padrão do Work Queue cujo valor é zero. A escolha dos valores para os multiplicadores do modo Especulativo por Tempo e *Fast Abort* foi feita através de uma análise de sensibilidade preliminar nas aplicações que mostrou ser necessário um equilíbrio entre o aumento do custo causado pela replicação e os ganhos obtidos pela execução possivelmente rápida de tarefas que estão atrasadas, bem como no reinício de tarefas que podem estar próximas a terminar. Estes multiplicadores não representam as melhores escolhas possíveis, sendo apenas escolhas com boa generalidade. Um estudo específico pode ser necessário para determinar os valores ótimos para cada caso.

Para garantir a reprodutibilidade e considerar variações na carga do *cluster*, cada configuração foi executada de forma independente 20 vezes, sendo uma execução no *cluster* chamada de execução. Isso resultou em 480 execuções por aplicação e um total de 960 execuções em diferentes dias e horários.

4.4 Resultados

Uma das medidas mais importantes em um *cluster* de alto desempenho é a latência das execuções, isto é, quanto tempo uma execução leva para terminar após seu início. Essa latência é determinante em diversas outras métricas, como a eficiência energética do *cluster* e seu Custo Total de Propriedade (usualmente conhecido como TCO ou *Total Cost of Ownership*). Quanto menor a latência, mais execuções são realizadas em determinada unidade de tempo e menos energia é gasta em ociosidade.

O diagrama de caixa (*boxplot*) na Figura 4.1 mostra a dispersão de todas as tarefas bem-sucedidas em todas as execuções na configuração de referência LSM_-HF em 16 nós. Apesar de a caixa ser pequena, indicando que a maioria das tarefas termina em menos de 20 minutos, a presença de valores discrepantes acima dos 40 minutos, alguns chegando até 9 vezes acima da mediana, faz com que a execução toda seja atrasado pois a execução só pode ser concluída quando a todas as tarefas

terminam com sucesso. Este atraso gera uma enorme ineficiência pois os demais recursos alocados à execução podem ficar ociosos aguardando o término da tarefa atrasada.

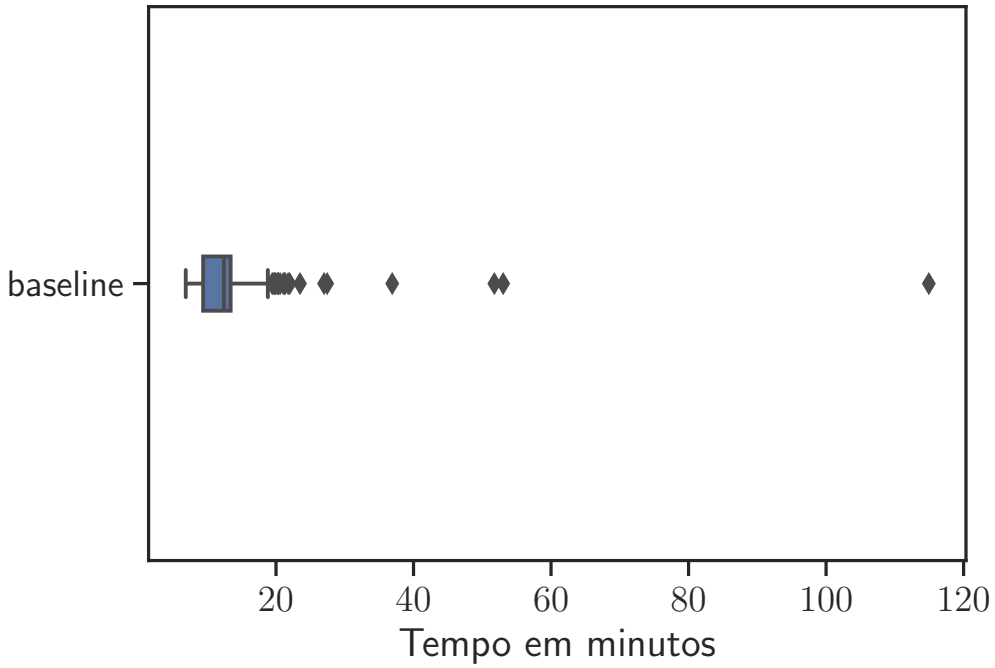

Figura 4.1: Latência da execução de referência das tarefas do LSM_HF em 16 nós.

Uma outra dimensão do problema pode ser vista na Tabela 4.3, onde é apresentada a quantidade percentual de execuções do LSM_HF de referência que tem tarefas cujo tempo de execução é maior que a mediana da respectiva execução multiplicada por um fator. Assim, 20% das execuções em 4 nós tiveram tarefas com tempo de execução maiores que 3 vezes a respectiva mediana. Desta tabela observa-se que há tarefas mais lentas em uma quantidade significativa das execuções com a configuração de 4 nós tendo os maiores percentuais. Consequentemente, assumimos neste trabalho que reduzir a latência da execução equivale a reduzir a latência da tarefa mais lenta, que serve como a principal métrica de avaliação para nossa abordagem proposta.

4.4.1 Latência das execuções no LSM_HF

Referência

São apresentados os tempos de execução referência medianos e máximos das execuções do LSM_HF na Tabela 4.4, enquanto a Figura 4.2 apresenta as latências das execuções de referência de cada execução agrupados por número de nós. Cada ponto representa uma execução e as linhas pontilhadas representam as medianas de cada grupo.

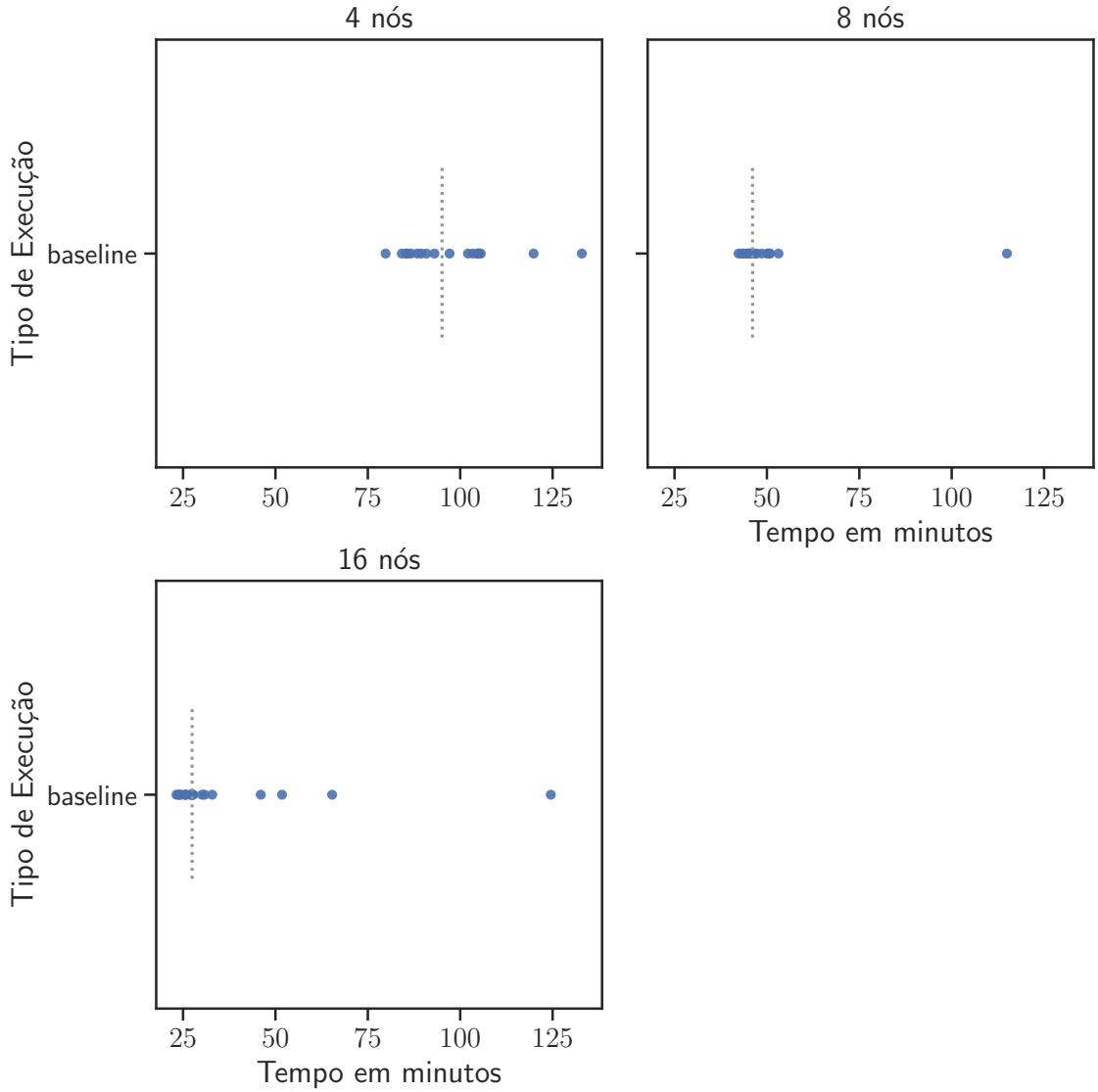

Figura 4.2: Latência da execução de referência do LSM_HF.

Como antecipado, o LSM_HF apresenta uma boa escalabilidade devido às suas tarefas completamente independentes e ao escalonamento linear dos processos com o número de nós, de forma que o paralelismo obtido pelos processos extras é melhor aproveitado. Há uma dispersão significativa com 4 nós, sendo bem menor com 8 e 16 nós, onde a maioria das execuções fica muito próxima à mediana. Numericamente, 75% das execuções demoraram no máximo 4 minutos além da mediana nas configurações de 8 e 16 nós, enquanto na configuração de 4 nós, 75% demoraram no máximo 9 minutos extras. A quantidade relevante de valores atípicos superiores indica que há, neste caso, espaço para melhorias através da especulação de tarefas, o que de fato ocorre como veremos a seguir.

A existência de valores discrepantes no LSM_HF se dá principalmente pelo fato de o LSM_HF fazer muitos acessos a arquivos em sistemas de arquivos remotos compartilhados. Isto aumenta a chance de falhas transientes e lentidão por sobrecarga temporária da rede ou do sistema de arquivos [3]. O número de execuções atípicas parece aumentar com o número de nós, devido à maior probabilidade de se encontrar um nó com alguma falha transiente.

Especulação por Tempo e *Backup Tasks*

Tanto *Backup Tasks* quanto Especulação por Tempo produziram ganhos significativos no LSM_HF (Figuras 4.3, 4.4 e 4.5): as medianas são todas menores que a mediana da configuração referência, os valores discrepantes superiores, quando existem, são sempre menores que os discrepantes da configuração referência e a dispersão diminuiu significativamente. O efeito da especulação diminui inversamente com a quantidade de nós, com as maiores reduções relativas observadas com 4 nós usando *Backup Tasks*: a mediana reduziu de 95.7 minutos para 83.14 minutos, uma redução de 12%, enquanto a execução mais longo foi reduzido em quase 30%, de 132.91 para 93.62 minutos. Atribuímos isso a quanto menor o grau de paralelismo, maior a chance de uma falha ou degradação transiente criar um gargalo. Adicionalmente, o número médio de réplicas criadas pelo *Backup Tasks* no LSM_HF (Tabela 4.5) é

	Nós		
Multiplicador da mediana	4	8	16
1.4	100%	100%	95%
1.5	100%	90%	85%
1.6	60%	15%	25%
2.0	90%	70%	60%
3.0	20%	5%	15%

Tabela 4.3: Percentual de execuções com tarefas acima da mediana no LSM_HF de referência.

Tabela 4.4: Latência em minutos da execução de referência dos jobs LSM_HF

Nós	Mediana	Máximo	Desvio
4	95.0771	132.91	13.2615
8	46.1076	114.946	15.5216
16	27.4487	124.513	23.4795

da mesma ordem de grandeza que na Especulação por Tempo.

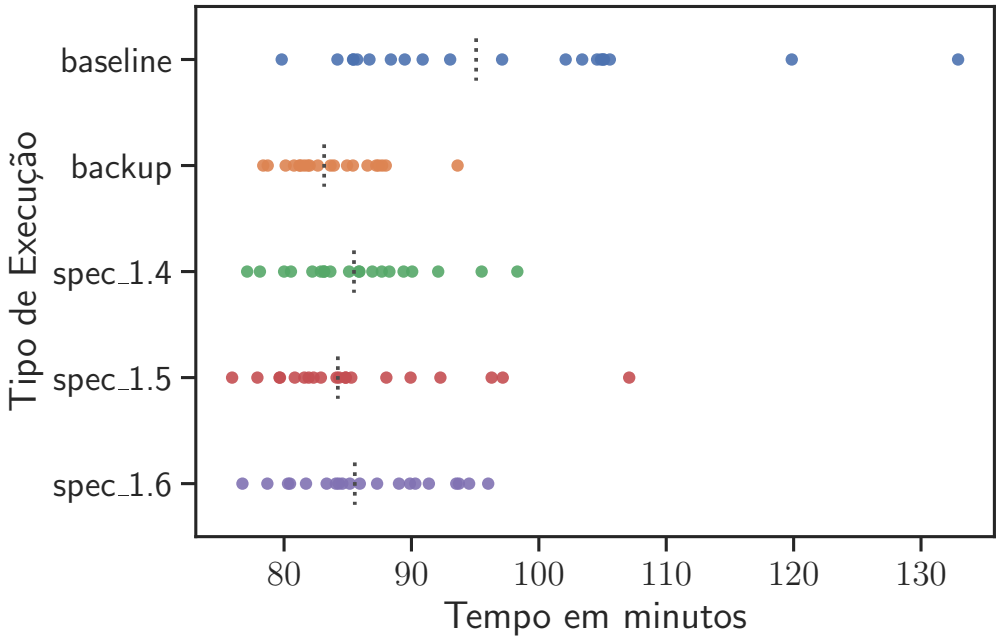

Figura 4.3: Latência usando Backup Tasks e Especulação por Tempo no LSM_HF em 4 nós.

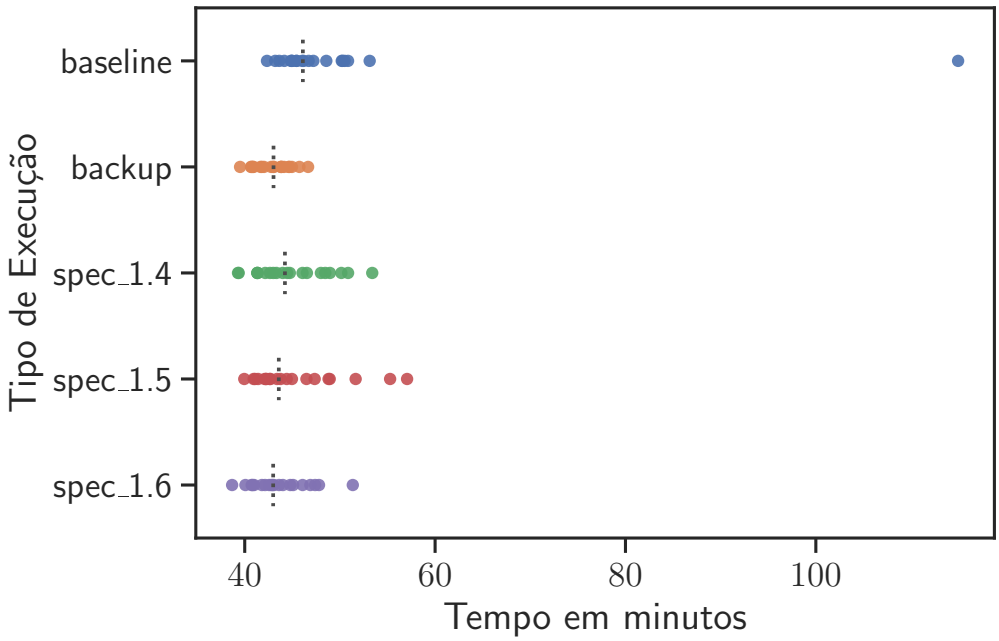

Figura 4.4: Latência usando Backup Tasks e Especulação por Tempo no LSM_HF em 8 nós.

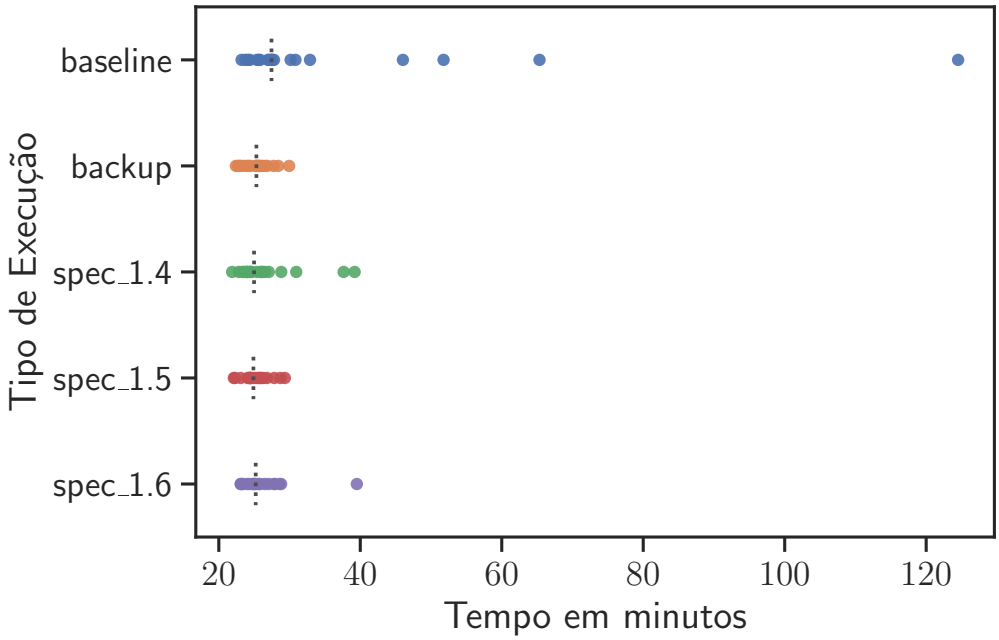

Figura 4.5: Latência usando Backup Tasks e Especulação por Tempo no LSM_HF em 16 nós.

Tabela 4.5: Número médio de réplicas geradas em 8 nós no LSM_HF

	Nr. total de tarefas	% replicas	% canceladas
Backup Tasks	32.00	21.8750	97.1429
Especulativa 1.4	27.80	10.0719	96.4286
Especulativa 1.5	27.25	8.2569	95.5556
Especulativa 1.6	26.90	7.4349	94.7368

Fast Abort

O *Fast Abort* produziu os piores resultados no LSM_HF, sendo o efeito bastante pronunciado, possivelmente devido ao maior tempo de execução de cada tarefa, como pode ser visto nas Figuras 4.6, 4.7 e 4.8. Cada cancelamento de uma tarefa no LSM_HF aproximadamente dobrou o tempo total médio por tarefa. No pior caso, *Fast Abort* 1.5 em 4 nós, a mediana aumentou de 95.07 minutos para 109.83 minutos, enquanto a execução mais longo demorou 51% mais tempo que a execução mais longo na configuração de referência.

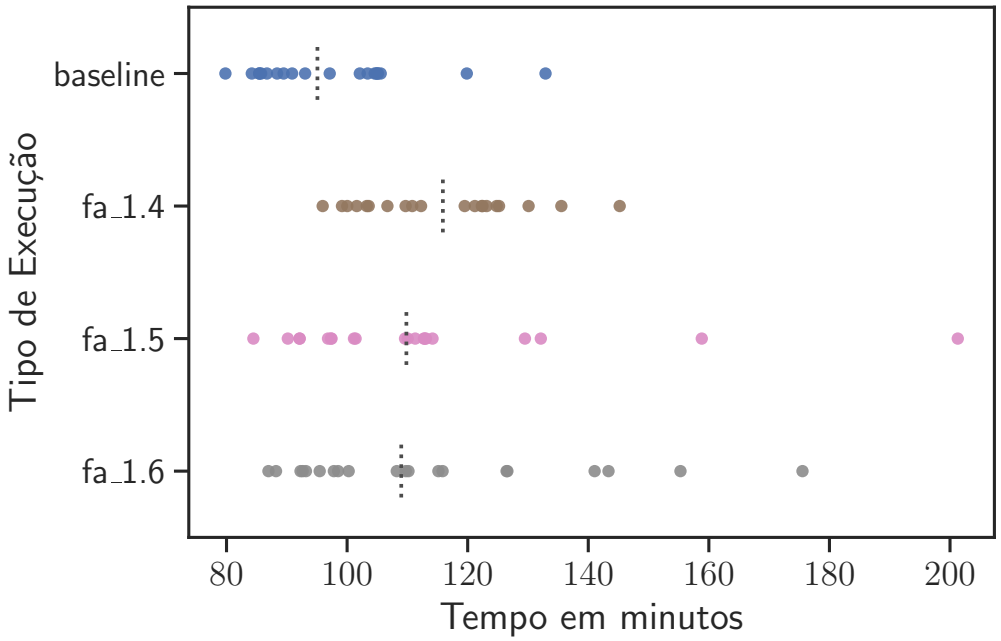

Figura 4.6: Latência usando Fast Abort no LSM_HF em 4 nós.

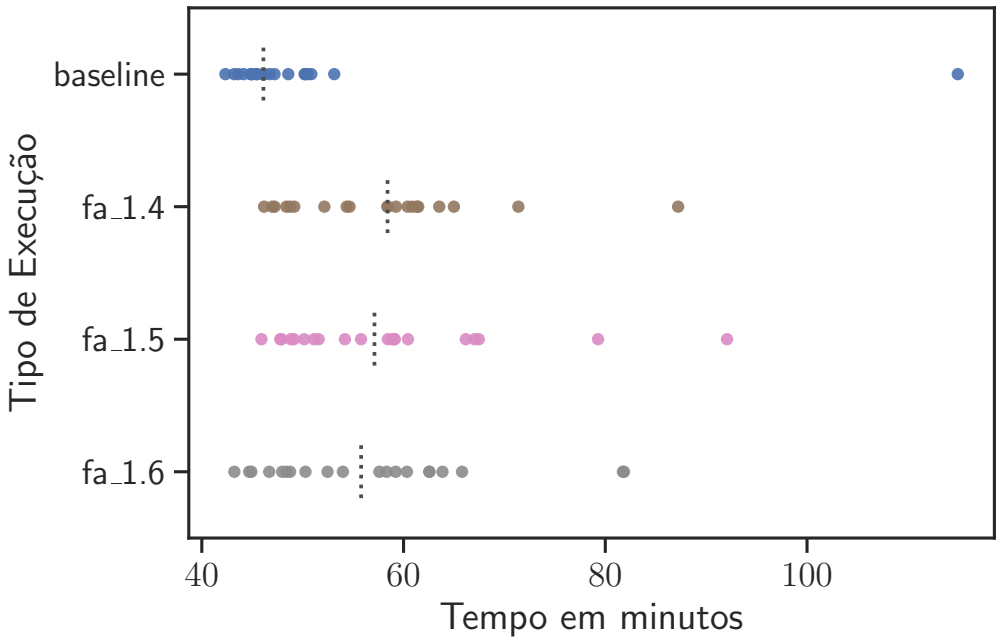

Figura 4.7: Latência usando Fast Abort no LSM_HF em 8 nós.

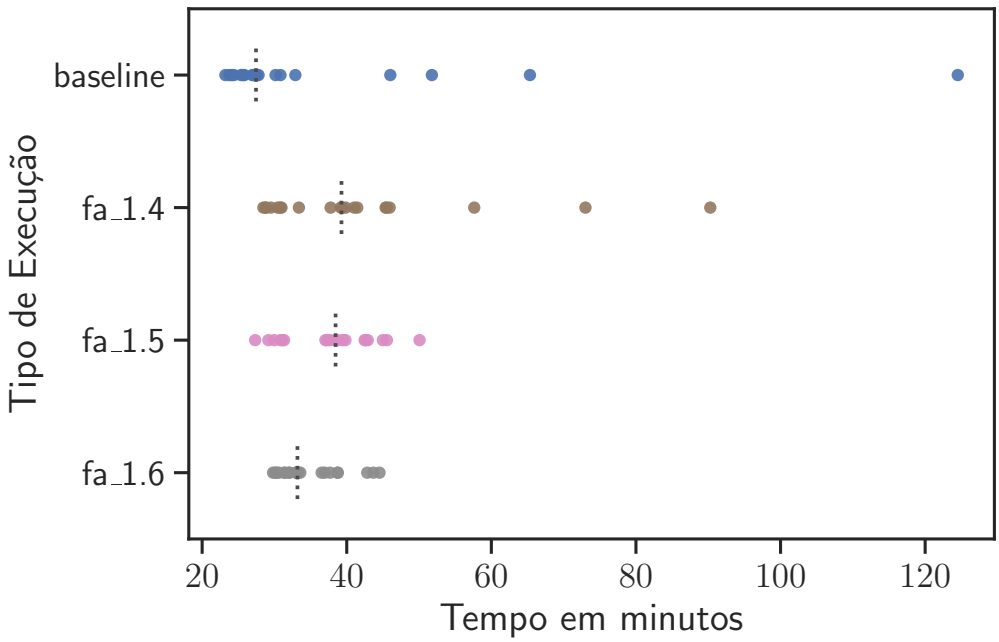

Figura 4.8: Latência usando Fast Abort no LSM_HF em 16 nós.

Como o *Fast Abort* aborta as tarefas lentas e as reexecuta, potencialmente em um trabalhador diferente, é preciso que o tempo de execução restante da tarefa lenta seja maior que o tempo de execução total médio da tarefa para que o *Fast Abort* apresente ganhos. Entretanto, nossos resultados indicam que o *Fast Abort* cancelava tarefas muito próximas de completar. Para quantificar este comportamento, acumulamos

os tempos de execução de todas as tentativas de cada tarefa como apresentado na Tabela 4.6. Esta tabela apresenta a média do tempo total de execução por tarefa agrupado pelo número de cancelamentos do LSM_HF em todas as configurações do *Fast Abort*.

Tabela 4.6: Média do tempo total (min) por tarefa por número de cancelamentos no LSM_HF

	Sem cancel.	1 cancel.	2 cancel.	3 cancel.	4 cancel.
baseline	12.9251	13.3055	15.6201	12.7702	11.8836
backup	12.0164	0	0	0	0
fa_1.4	12.2600	31.9679	62.8606	0	0
fa_1.5	12.5470	32.5841	64.8560	120.3000	0
fa_1.6	12.2796	34.0216	72.3517	0	0
spec_1.4	12.0854	0	0	0	0
spec_1.5	12.0128	18.7519	0	0	0
spec_1.6	12.0200	0	0	0	0

Nossa análise revela que a maioria dos cancelamentos observados nos modos Referência, Backup Tasks e Especulação por Tempo são espúrios, isto é, não representam cancelamentos reais, apenas transições entre os estados EXECUTANDO e PRONTO dentro do Work Queue, sem que de fato a tarefa tenha começado sua execução no trabalhador, devido a um pequeno defeito no Work Queue. Isto pode ser confirmado pela média do tempo total relativamente constante entre as diferentes contagens de cancelamentos das tarefas. A exceção é o *Fast Abort*, onde a média do tempo total aumenta com o aumento no número de cancelamentos. Este efeito é particularmente proeminente na configuração de *Fast Abort* com multiplicador 1.5, onde a média de tempo total de tarefas com 3 cancelamentos (ou seja a tarefa foi reiniciada 3 vezes até executar com sucesso) foi de 12.2796 minutos para 120.30 minutos, uma ordem de grandeza maior que a média de tempo total de tarefas bem sucedidas sem nenhum cancelamento.

Considerando que os valores discrepantes na execução de referência do LSM_HF levaram mais do que o dobro do tempo médio de execução, é possível que esperar por 1.6 vezes o tempo médio seja muito pouco para o LSM_HF (sendo ainda pior para 1.5 e 1.4), e um valor maior para o parâmetro produza resultados melhores. Embora reiniciar as tarefas do início ainda resulte em um tempo de execução de pelo menos a soma do tempo da tarefa cancelada e do tempo da nova execução, aumentar o fator de tolerância potencialmente reduziria o número de cancelamentos para tarefas próxima da conclusão.

4.4.2 Latência das execuções no Wavefront

Referência

Na Tabela 4.7 são apresentados os tempos de referência medianos e máximos de execução das execuções do Wavefront, enquanto a Figura 4.9 apresenta as latências das execuções de referência de cada execução agrupados por número de nós. Cada ponto representa uma execução e as linhas pontilhadas representam as medianas das execuções.

Tabela 4.7: Latências em minutos da execução de referência dos jobs Wavefront

Nós	Mediana	Máximo	Desvio
4	64.9830	65.0994	0.0566
8	64.9220	65.0963	0.1697
16	64.8793	65.3604	0.1751

A primeira observação importante é que para o Wavefront o número de nós afeta de forma insignificante o tempo total de execução: a mediana do tempo diminuiu de 64.9830 minutos com 4 nós para 64.8793 minutos com 16 nós. Isso é esperado, pois além de o número total de processos ser constante, o paralelismo é bastante limitado, devido às dependências entre as tarefas. No entanto, o aumento do número de nós, e a consequente diminuição do número de processos por nó, permite alguns ganhos, especialmente de 4 para 8 nós, com uma parte significativa das execuções com 8 nós abaixo do tempo mediano obtido pelas execuções de 4 nós. Com 16 nós observa-se uma redução da variabilidade nos tempos de execução e o surgimento de alguns valores atípicos mais rápidos, embora a melhoria geral tenha sido marginal, com menos de um minuto separando a execução mais rápida da mais lenta.

Ao contrário do que foi observado em [22], há poucos valores discrepantes nas execuções do Wavefront, indicando que há pouco espaço para melhorias obtidas através da especulação de tarefas. Atribuímos isso à natureza dedicada do ambiente utilizado neste trabalho, diferentemente do utilizado no referido artigo, e ao fato de a maioria das tarefas no Wavefront serem de rápida execução e computacionalmente intensivas, sem comunicação ou acesso à discos, ao contrário do que ocorre no LSM_HF que utiliza sistemas de arquivos remotos. Estas características do Wavefront reduzem a probabilidade de falhas transientes, consequentemente reduzindo a probabilidade da ocorrência de tarefas atrasadas. Isto torna difícil avaliar a proposta de especulação de tarefas deste trabalho no Wavefront, levando à necessidade de outra aplicação para uma avaliação mais completa. Por outro lado, é possível verificar que nenhuma regressão foi introduzida na execução normal do Work Queue e nos permite avaliar se algum dos modos de especulação é desaconselhável para aplicações com muitas dependências ou com geração de dinâmica de tarefas.

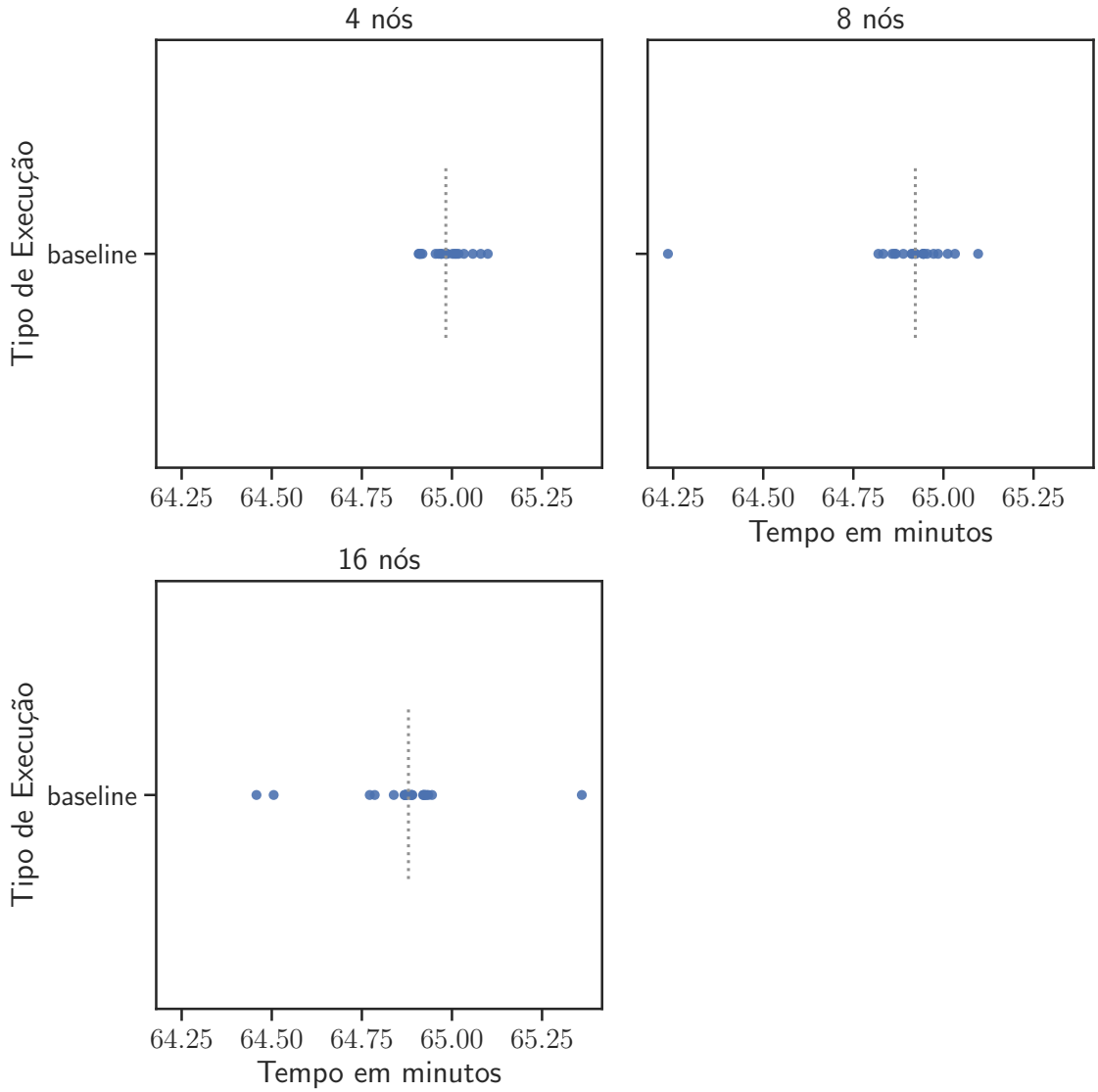

Figura 4.9: Latência da execução de referência do Wavefront.

Especulação por Tempo e *Backup Tasks*

A especulação de tempo alcançou resultados semelhantes à configuração de referência, com a mediana em torno de 64.9 minutos e uma ligeira redução na dispersão, além do surgimento de valores de discrepantes muito menores do que na configuração de referência (Figuras 4.10 , 4.11 e 4.12). Como previsto anteriormente, a especulação de tarefas foi ineficaz para o Wavefront e o impacto negativo do modo *Backup Tasks* no desempenho do Wavefront também não é surpreendente. A construção dinâmica de tarefas e as dependências entre tarefas inerentes ao Wavefront frequentemente resultam em trabalhadores ociosos e filas de tarefas vazias, condições que acionam a criação desnecessária de réplicas no modo *Backup Tasks*. Isso levou a um aumento no número de réplicas, a maior parte das quais foi cancelada posteriormente devido à conclusão das tarefas originais, atrasando a execução de novas tarefas originais que entravam na fila.

Comparando o número médio de réplicas criadas e canceladas do *Backup Tasks* com os mesmos números de réplicas da Especulação por Tempo em 8 nós no Wavefront (Tabela 4.8), verifica-se que o número de réplicas criadas corresponde a 4% de todas as tarefas submetidas no *Backup Tasks* sendo que a maioria delas é cancelada. Em contrapartida, na Especulação por Tempo, o número de réplicas não chega nem a 1% das tarefas totais e a maioria delas também é cancelada. Esse número expressivo de tarefas extras inúteis cria um custo adicional na execução do Wavefront com *Backup Tasks* aumentando assim o tempo total de execução para próximo de 66.4 minutos.

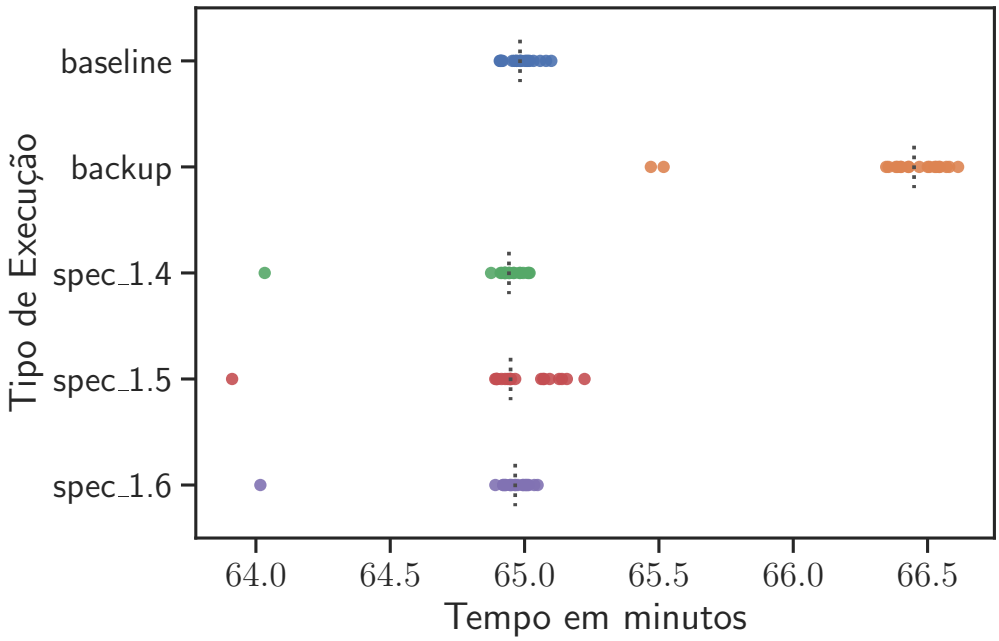

Figura 4.10: Latência usando Backup Tasks e Especulação por Tempo no Wavefront em 4 nós.

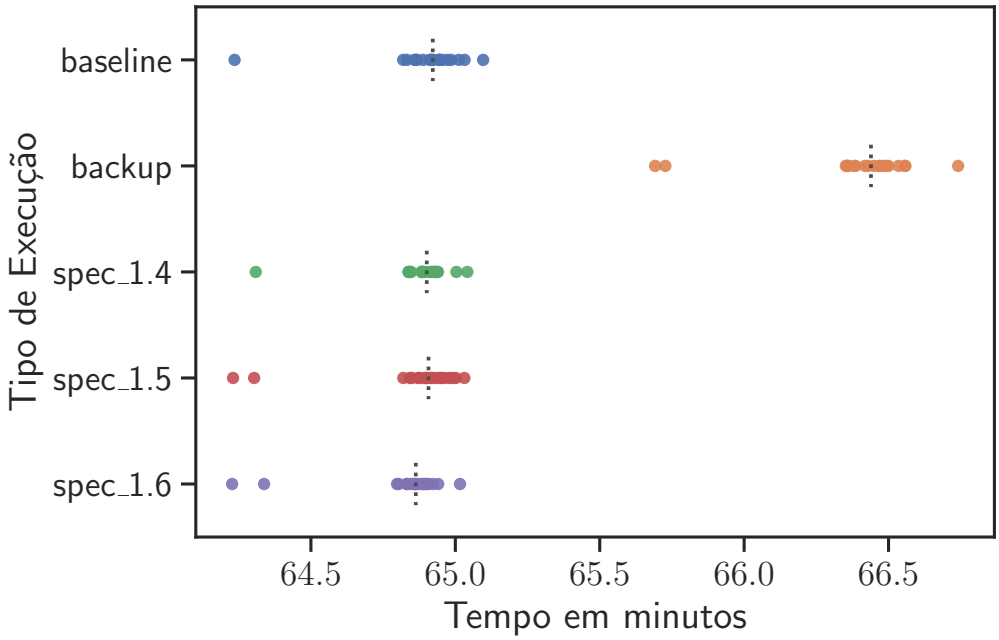

Figura 4.11: Latência usando Backup Tasks e Especulação por Tempo no Wavefront em 8 nós.

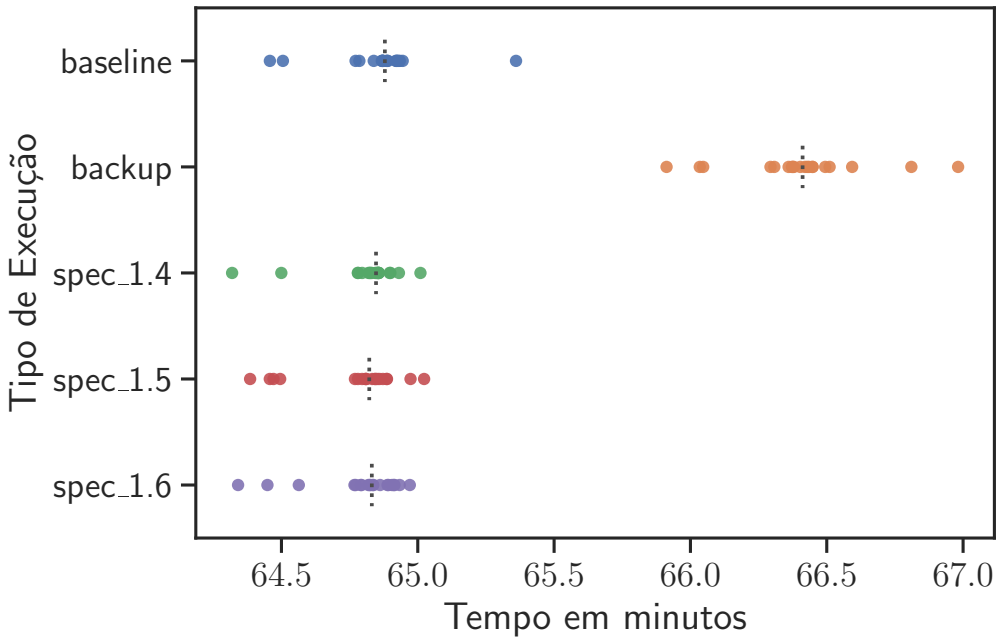

Figura 4.12: Latência usando Backup Tasks e Especulação por Tempo no Wavefront em 16 nós.

Tabela 4.8: Número médio de réplicas geradas em 8 nós no Wavefront

	Nr. total de tarefas	% replicas	% canceladas
Backup Tasks	260477.60	4.4060	78.65
Especulativa 1.4	249013.50	0.0053	95.60
Especulativa 1.5	249015.55	0.0058	87.50
Especulativa 1.6	249003.45	0.0013	100.00

Fast Abort

Nas Figuras 4.13, 4.14, 4.15, pode ser visto que a execução usando *Fast Abort* apresentou pioras significativas no tempo de execução do Wavefront. Algumas execuções ficaram muito mais lentas que a configuração referência, com a mediana indo de 64.9 minutos na execução de referência para mais de 66.5 minutos com *Fast Abort* 1.4 em 8 nós. A variabilidade também aumentou muito, com execuções variando entre 64.2 e 72.5 minutos na mesma configuração.

De forma semelhante ao que foi feito no LSM_HF, acumulamos os tempos de execução de todas as tentativas de cada tarefa como apresentado na Tabela 4.9. Esta tabela apresenta a média do tempo total de execução por tarefa agrupado pelo número de cancelamentos do Wavefront em todas as configurações do *Fast Abort*.

Também aqui a maioria dos cancelamentos fora do *Fast Abort* são espúrios, isto é, não representam cancelamentos reais. No Wavefront o efeito é particularmente

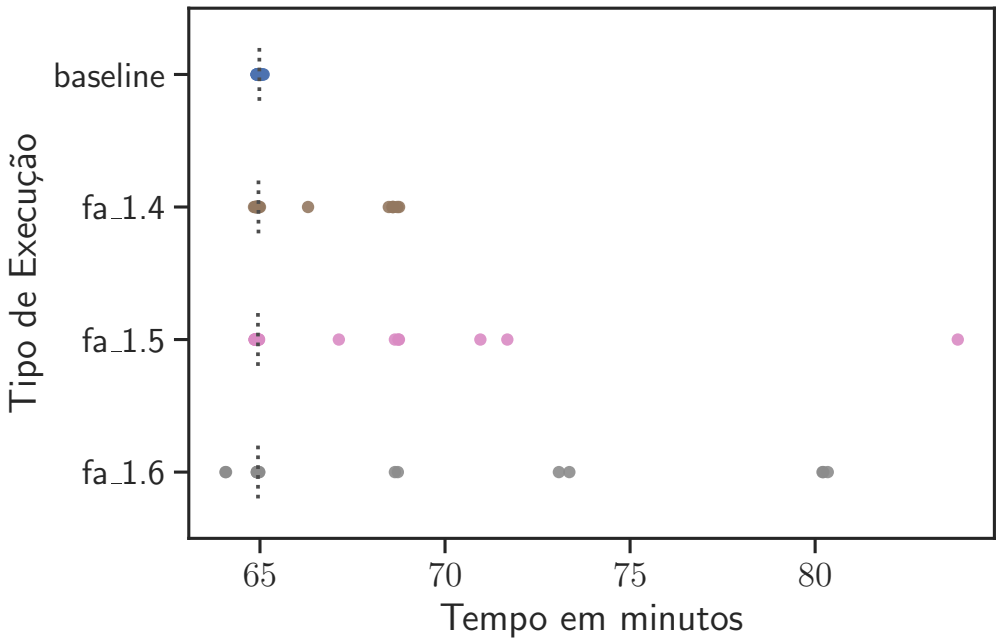

Figura 4.13: Latência usando Fast Abort no Wavefront em 4 nós.

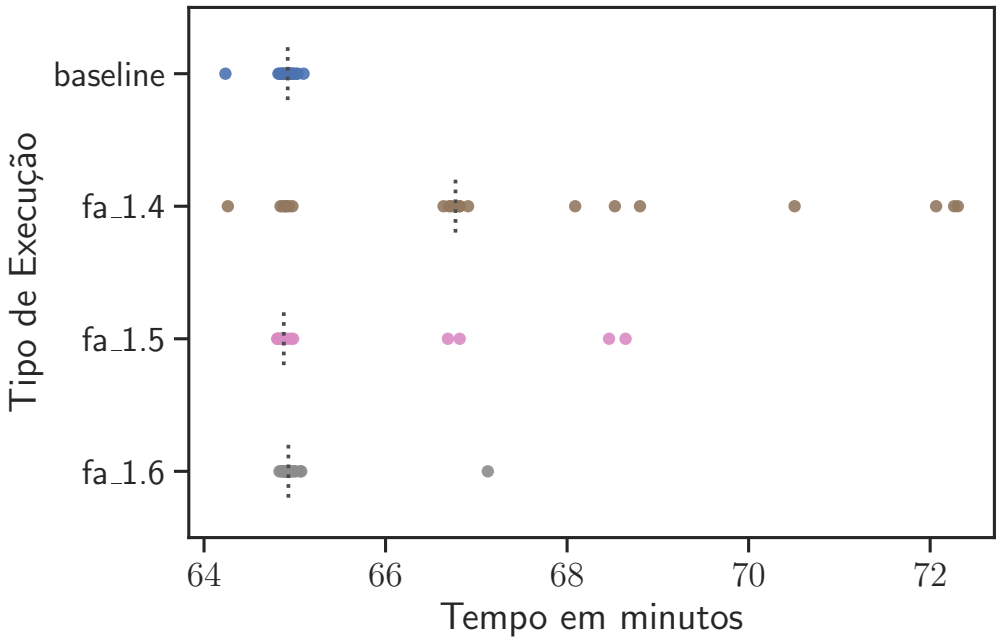

Figura 4.14: Latência usando Fast Abort no Wavefront em 8 nós.

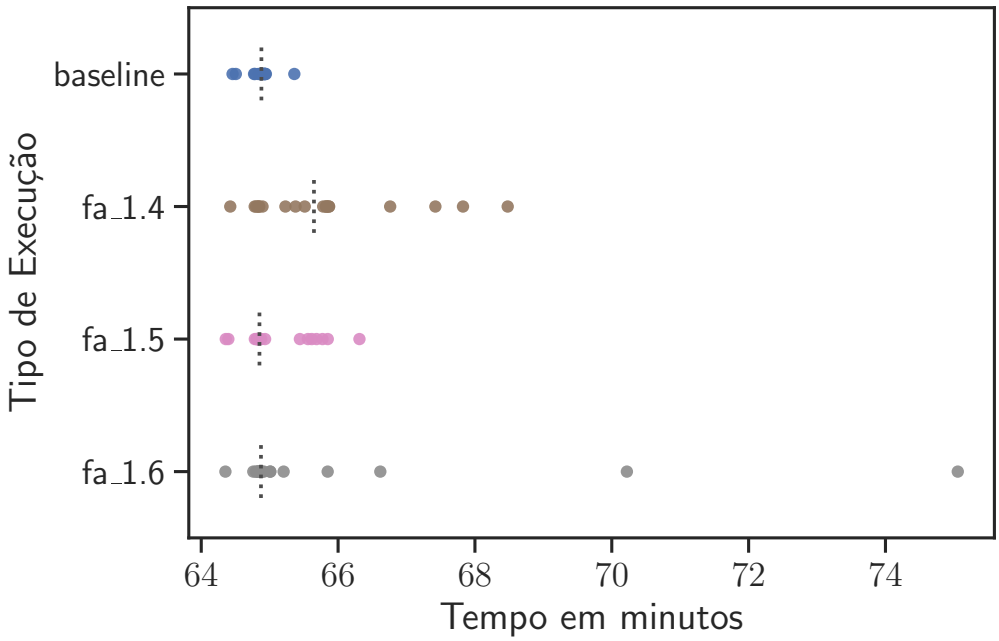

Figura 4.15: Latência usando Fast Abort no Wavefront em 16 nós.

Tabela 4.9: Média do tempo total (s) por tarefa por número de cancelamentos no Wavefront

	Sem cancel.	1 cancel.	2 cancel.	3 cancel.	4 cancel.
baseline	1.3596	1.3543	1.3733	0	0
backup	1.3663	1.3509	1.3749	0	0
fa_1.4	1.3582	1.5591	1.5541	0	0
fa_1.5	1.3587	1.4467	1.3512	0	0
fa_1.6	1.3592	1.8927	10.7845	11.1711	8.5780
spec_1.4	1.3586	1.3520	1.3811	0	0
spec_1.5	1.3581	1.3573	1.3738	0	0
spec_1.6	1.3582	1.3521	1.3562	0	0

proeminente na configuração de *Fast Abort* com multiplicador 1.6, onde a média de tempo total de tarefas com 3 cancelamentos foi 8 vezes maior que a média de tempo total de tarefas bem sucedidas sem nenhum cancelamento.

O mecanismo de *Fast Abort* foi introduzido no Work Queue como uma forma de minimizar o impacto de tarefas lentas no Wavefront e estes resultados parecem indicar o oposto. Entretanto, a infraestrutura usada neste trabalho é dedicada e mais homogênea (todos os nós no *cluster* são idênticos) enquanto que a infraestrutura utilizada em [22] é um *Grid* não dedicado, sujeito a preempção de tarefas. Isto explica a dispersão no Wavefront ser da ordem de 0.1 minutos neste trabalho. Como previsto na análise das execuções de referência, esta dispersão é muito pequena para que os mecanismos de *Fast Abort* e especulação de tarefas apresentem ganhos, dada a ausência de falhas transientes e preempção de tarefas.

Capítulo 5

Trabalhos Relacionados

Dean e Ghemawat [1] introduziram o mecanismo de *Backup Tasks* no MapReduce, como técnica de especulação de tarefas para mitigar atrasos causados pelas tarefas atrasadas, alcançando uma redução de latência de até 66% em casos específicos. Pesquisas subsequentes exploraram a especulação de tarefas em *frameworks* inspirados no MapReduce, como Hadoop [2, 8] e Spark [7, 20, 29]. Os fundamentos teóricos da especulação e replicação de tarefas também foram amplamente estudados na literatura [2, 12, 19, 30].

Entretanto, o modelo *MapReduce* não atende a todas as necessidades de computação de alto desempenho, sendo o modelo mestre-trabalhador uma generalização importante que pode se beneficiar da especulação de tarefas. Além disso, neste trabalho foi introduzido o modelo de especulação por tempo como alternativa ao modelo de *Backup Tasks* para situações onde este último não apresente bons resultados.

Dentre os *frameworks* baseados em MPI que implementam replicação de tarefas ou processos, destacam-se VolpexMPI [31], rMPI [16], PAREP-MPI [32], teaMPI [33] e PartRePer-MPI [34]. Esses *frameworks* abordam principalmente as deficiências do MPI em termos de tolerância a falhas de processos e replicação, dando pouca atenção a tarefas atrasadas e falhas transitórias.

Nos modelos *Asynchronous Many-Task* (AMT), a computação é dividida em tarefas de granularidade fina com dependências explícitas. Diferentemente do paradigma hierárquico mestre-trabalhador, os modelos AMT representam computações como grafos acíclicos direcionados (DAGs), onde as tarefas podem ser executadas independentemente desde que suas dependências sejam satisfeitas. O sistema agenda dinamicamente essas tarefas entre os recursos disponíveis. *Frameworks* como OmpSs [35], HClib [36] e HPX [37] oferecem implementações de replicação e especulação de tarefas dentro do paradigma AMT. Os sistemas AMT oferecem mecanismos de especulação de tarefas mais similares ao apresentado neste trabalho, com as principais distinções sendo a granularidade da tarefa e o tratamento de falhas. Enquanto o Work Queue permite tarefas de granularidade mais grossa, na forma de progra-

mas inteiros executados em processos independentes, os sistemas AMT tipicamente gerenciam unidades de granularidade mais fina como subrotinas executadas em *threads*. Além disso, *frameworks* AMT como HClib e HPX abordam principalmente falhas locais, carecendo da capacidade de redistribuir tarefas entre trabalhadores. Em contraste, este trabalho propõe um sistema capaz de redistribuir tarefas para qualquer trabalhador disponível, melhorando a tolerância a falhas.

Capítulo 6

Conclusões e Trabalhos Futuros

Este trabalho aborda o desafio da tolerância a falhas em ambientes de computação de alto desempenho (HPC), focando em aplicações mestre-trabalhador. Ao introduzir e avaliar a especulação de tarefas como um método para mitigar o impacto de falhas transitórias e degradação de desempenho, demonstramos seu potencial para melhorar a robustez e eficiência de aplicações distribuídas. Tarefas atrasadas, que são significativamente mais lentas que outras tarefas, representam um problema substancial em ambientes HPC. Elas podem afetar severamente o tempo total de execução, especialmente em aplicações mestre-trabalhador onde o tempo de conclusão é frequentemente determinado pela tarefa mais lenta. Falhas transitórias, uma das causas de tarefas atrasadas, são difíceis de prever, detectar e corrigir devido à sua natureza. Nossa abordagem utiliza replicação e especulação parcial de tarefas para abordar proativamente esses desafios com baixo custo.

Nossos experimentos demonstram que a especulação de tarefas é uma estratégia eficaz para reduzir significativamente a latência em programas distribuídos propensos a falhas transitórias sem incorrer em sobrecarga substancial. Aplicações com alta variabilidade no tempo de conclusão de tarefas beneficiam-se mais dessa abordagem, pois a execução especulativa mitiga proativamente atrasos causados por tarefas lentas ou falhas. A redução observada de até 4 vezes no tempo máximo de execução e a redução do tempo de execução mediano em até 12% destacam o potencial da especulação de tarefas para melhorar a robustez e desempenho de sistemas distribuídos. Nossos resultados indicam que a eficácia das estratégias de especulação de tarefas e *Fast Abort* são muito dependentes da aplicação específica e do contexto operacional.

No LSM_HF usando a configuração Especulação de Tempo 1.5 em 16 nós, com apenas 8% de réplicas de tarefas extras, o tempo máximo de execução foi reduzido para menos de 30 minutos, enquanto na configuração de referência o tempo máximo de execução foi de 124.5 minutos, uma redução de aproximadamente 4 vezes. Embora essas técnicas tenham demonstrado ganhos de desempenho significativos para determinadas aplicações, o Wavefront exibiu melhorias limitadas e, em alguns casos,

a especulação até prejudicou o desempenho. No Wavefront, os tempos médios de execução ficaram próximos de 64.9 minutos em todas as configurações de referência, enquanto os tempos máximos de execução ficaram próximos de 65.0 minutos, uma diferença insignificante, e a especulação de tarefas com *Backup Tasks* piorou o desempenho, elevando os tempos médios de execução para mais de 66.5 minutos. Além disso, a estratégia de *Fast Abort* foi menos eficaz em ambientes com baixas taxas de falhas persistentes. Consequentemente, uma avaliação cuidadosa das características da aplicação e das condições operacionais é essencial antes de implementar estratégias de especulação de tarefas ou *Fast Abort*.

Pesquisas futuras devem investigar valores ótimos para os multiplicadores do modo Especulação por Tempo, explorar modos de especulação adicionais, como a replicação sem espera proposta por Ananthanarayanan et al. [2], e aumentar o número de réplicas por tarefa. Uma avaliação abrangente de outros algoritmos da literatura, incluindo [9, 18, 20, 30, 35, 38], poderia fornecer diferentes perspectivas para otimizar a especulação de tarefas. Além disso, é essencial avaliar o desempenho das estratégias de replicação em ambientes de nuvem não dedicados e voláteis para compreender sua adaptabilidade em diversos contextos operacionais. Testar a replicação em uma gama mais ampla de aplicações distribuídas e paralelas ajudará a generalizar os resultados e melhorar a robustez dos mecanismos de tolerância a falhas em sistemas HPC. Além disso, a integração de técnicas de aprendizado de máquina para prever falhas de tarefas e ajustar dinamicamente as estratégias de especulação poderia melhorar ainda mais a eficiência e confiabilidade das aplicações mestre-trabalhador.

Referências Bibliográficas

[1] DEAN, J., GHEMAWAT, S. "MapReduce: Simplified Data Processing on Large Clusters", *Communications of the ACM*, v. 51, n. 1, pp. 107–113, jan. 2008. ISSN: 0001-0782, 1557-7317. doi: 10.1145/1327452.1327492.

[2] ANANTHANARAYANAN, G., GHODSI, A., SHENKER, S., et al. "Effective Straggler Mitigation: Attack of the Clones". In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pp. 185–198, 2013. ISBN: 978-1-931971-00-3. Disponível em: <https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/ananthanarayanan>. Acesso em: 03/10/2023.

[3] DEAN, J., BARROSO, L. A. "The Tail at Scale", *Communications of the ACM*, v. 56, n. 2, pp. 74–80, fev. 2013. ISSN: 0001-0782, 1557-7317. doi: 10.1145/2408776.2408794.

[4] DONGARRA, J., BECKMAN, P., MOORE, T., et al. "The International Exascale Software Project Roadmap", *The International Journal of High Performance Computing Applications*, v. 25, n. 1, pp. 3–60, fev. 2011. ISSN: 1094-3420, 1741-2846. doi: 10.1177/1094342010391989.

[5] CAPPELLO, F., GEIST, A., GROPP, W., et al. "Toward Exascale Resilience: 2014 Update", *Supercomputing Frontiers and Innovations*, v. 1, n. 1, pp. 5–28, jun. 2014. ISSN: 2313-8734. doi: 10.14529/jsfi140101.

[6] XU, H., LAU, W. C. "Speculative Execution for a Single Job in a MapReduce-Like System". In: *2014 IEEE 7th International Conference on Cloud Computing*, pp. 586–593, jun. 2014. doi: 10.1109/CLOUD.2014.84.

[7] ZHANG, P., GUO, Z. "An Improved Speculative Strategy for Heterogeneous Spark Cluster", *MATEC Web of Conferences*, v. 173, pp. 01018, 2018. ISSN: 2261-236X. doi: 10.1051/matecconf/201817301018.

[8] ZAHARIA, M., KONWINSKI, A., JOSEPH, A. D., et al. "Improving MapReduce Performance in Heterogeneous Environments." In: *Osdi*,

v. 8, p. 7, 2008. Disponível em: <https://www.usenix.org/legacy/event/osdi08/tech/full_papers/zaharia/zaharia.pdf>. Acesso em: 27/01/2024.

[9] BEHROUZI-FAR, A., SOLJANIN, E. "Efficient Replication for Fast and Predictable Performance in Distributed Computing", *IEEE/ACM Transactions on Networking*, v. 29, n. 4, pp. 1467–1476, ago. 2021. ISSN: 1063-6692, 1558-2566. doi: 10.1109/TNET.2021.3062215.

[10] MESSAGE PASSING INTERFACE FORUM. *MPI: A Message-Passing Interface Standard Version 4.0*, jun. 2021. Disponível em: <https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf>. Acesso em: 01/08/2023.

[11] AVIZIENIS, A., LAPRIE, J.-C., RANDELL, B., et al. "Basic Concepts and Taxonomy of Dependable and Secure Computing", *IEEE Transactions on Dependable and Secure Computing*, v. 1, n. 1, pp. 11–33, jan. 2004. ISSN: 1941-0018. doi: 10.1109/TDSC.2004.2.

[12] AKTAS, M. F., PENG, P., SOLJANIN, E. "Effective Straggler Mitigation: Which Clones Should Attack and When?" *ACM SIGMETRICS Performance Evaluation Review*, v. 45, n. 2, pp. 12–14, out. 2017. ISSN: 0163-5999. doi: 10.1145/3152042.3152047.

[13] AKTAS, M. F., SOLJANIN, E. "Straggler Mitigation at Scale", *IEEE/ACM Transactions on Networking*, v. 27, n. 6, pp. 2266–2279, dez. 2019. ISSN: 1063-6692, 1558-2566. doi: 10.1109/TNET.2019.2946464.

[14] BERNHOLDT, D. E., BOEHM, S., BOSILCA, G., et al. "A Survey of MPI Usage in the US Exascale Computing Project", *Concurrency and Computation: Practice and Experience*, v. 32, n. 3, pp. e4851, fev. 2020. ISSN: 1532-0626, 1532-0634. doi: 10.1002/cpe.4851.

[15] JIA, J., LIU, Y., ZHANG, G., et al. "Software Approaches for Resilience of High Performance Computing Systems: A Survey", *Frontiers of Computer Science*, v. 17, n. 4, pp. 174105, ago. 2023. ISSN: 2095-2228, 2095-2236. doi: 10.1007/s11704-022-2096-3.

[16] FERREIRA, K., STEARLEY, J., LAROS, J. H., et al. "Evaluating the Viability of Process Replication Reliability for Exascale Systems". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pp. 1–12, New York, NY, USA,

nov. 2011. Association for Computing Machinery. ISBN: 978-1-4503-0771-0. doi: 10.1145/2063384.2063443.

[17] BENOIT, A., HERAULT, T., FÈVRE, V. L., et al. "Replication Is More Efficient than You Think". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, pp. 1–14, New York, NY, USA, nov. 2019. Association for Computing Machinery. ISBN: 978-1-4503-6229-0. doi: 10.1145/3295500.3356171.

[18] WANG, D., JOSHI, G., WORNELL, G. W. "Efficient Straggler Replication in Large-Scale Parallel Computing", *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, v. 4, n. 2, pp. 1–23, jun. 2019. ISSN: 2376-3639, 2376-3647. doi: 10.1145/3310336.

[19] WANG, D., JOSHI, G., WORNELL, G. "Using Straggler Replication to Reduce Latency in Large-scale Parallel Computing", *ACM SIGMETRICS Performance Evaluation Review*, v. 43, n. 3, pp. 7–11, nov. 2015. ISSN: 0163-5999. doi: 10.1145/2847220.2847223.

[20] SAID, S. A., HABASHY, S. M., SALEM, S. A., et al. "An Optimized Straggler Mitigation Framework for Large-Scale Distributed Computing Systems", *IEEE Access*, v. 10, pp. 97075–97088, 2022. ISSN: 2169-3536. doi: 10.1109/ACCESS.2022.3205723.

[21] BUI, P., RAJAN, D., ABDUL-WAHID, B., et al. "Work Queue + Python: A Framework For Scalable Scientific Ensemble Applications". In: *ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (Supercomputing)*, USA, 2011. Disponível em: <http://ccl.cse.nd.edu/research/papers/wq-python-pyhpc2011.pdf>. Acesso em: 28/03/2023.

[22] YU, L., MORETTI, C., THRASHER, A., et al. "Harnessing Parallelism in Multicore Clusters with the All-Pairs, Wavefront, and Makeflow Abstractions", *Cluster Computing*, v. 13, n. 3, pp. 243–256, set. 2010. ISSN: 1386-7857, 1573-7543. doi: 10.1007/s10586-010-0134-7.

[23] "Atlas - Bull 4029GP-TVRT, Xeon Gold 6240 18C 2.6GHz, NVIDIA Tesla V100, Infiniband EDR | TOP500", 2020. Disponível em: <https://www.top500.org/system/179854/>. Acesso em: 12/08/2023.

[24] "Petrobras' Atlas Supercomputer Joins List of World's Largest", jul. 2020. Disponível em: <https://jpt.spe.org/

petrobras-atlas-supercomputer-joins-list-worlds-largest>. Acesso em: 12/08/2023.

[25] YOO, A. B., JETTE, M. A., GRONDONA, M. "SLURM: Simple Linux Utility for Resource Management". In: Goos, G., Hartmanis, J., Van Leeuwen, J., et al. (Eds.), *Job Scheduling Strategies for Parallel Processing*, v. 2862, *Lecture Notes in Computer Science*, pp. 44–60, Berlin, Heidelberg, 2003. Springer. ISBN: 978-3-540-39727-4. doi: 10.1007/10968987_3.

[26] KOUTOUPIS, P. "The Lustre Distributed Filesystem", *Linux J.*, v. 2011, n. 210, pp. 3:3, 2011. ISSN: 1075-3583.

[27] GRAY, S. H., ETGEN, J., DELLINGER, J., et al. "Seismic Migration Problems and Solutions", *GEOPHYSICS*, v. 66, n. 5, pp. 1622–1640, set. 2001. ISSN: 0016-8033. doi: 10.1190/1.1487107.

[28] RODRIGUES, J., DUARTE, F., DOMINGOS, D., et al. "Least-Squares Reverse Time Migration: Lessons Learned from Recent Applications in Challenging Areas of the Santos Basin, Brazil". In: *Second International Meeting for Applied Geoscience & Energy*, SEG Technical Program Expanded Abstracts, Society of Exploration Geophysicists and American Association of Petroleum Geologists, pp. 2699–2703, Houston, TX, USA, ago. 2022. doi: 10.1190/image2022-3745685.1.

[29] ZAHARIA, M., CHOWDHURY, M., DAS, T., et al. "Resilient Distributed Datasets: A {Fault-Tolerant} Abstraction for {In-Memory} Cluster Computing". In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pp. 15–28, 2012. Disponível em: <https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia>. Acesso em: 16/11/2023.

[30] WANG, D., JOSHI, G., WORNELL, G. "Efficient Task Replication for Fast Response Times in Parallel Computation". In: *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, pp. 599–600, New York, NY, USA, jun. 2014. Association for Computing Machinery. ISBN: 978-1-4503-2789-3. doi: 10.1145/2591971.2592042.

[31] LEBLANC, T., ANAND, R., GABRIEL, E., et al. "VolpexMPI: An MPI Library for Execution of Parallel Applications on Volatile Nodes". In: Ropo, M., Westerholm, J., Dongarra, J. (Eds.), *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Lecture Notes in Computer

Science, pp. 124–133, Berlin, Heidelberg, 2009. Springer. ISBN: 978-3-642-03770-2. doi: 10.1007/978-3-642-03770-2_19.

[32] GEORGE, C., VADHIYAR, S. "Fault Tolerance on Large Scale Systems Using Adaptive Process Replication", *IEEE Transactions on Computers*, v. 64, n. 8, pp. 2213–2225, ago. 2015. ISSN: 0018-9340. doi: 10.1109/TC.2014.2360536.

[33] SAMFASS, P., WEINZIERL, T., HAZELWOOD, B., et al. "TeaMPI—Replication-Based Resilience Without the (Performance) Pain". In: Sadayappan, P., Chamberlain, B. L., Juckeland, G., et al. (Eds.), *High Performance Computing*, Lecture Notes in Computer Science, pp. 455–473, Cham, 2020. Springer International Publishing. ISBN: 978-3-030-50743-5. doi: 10.1007/978-3-030-50743-5_23.

[34] JOSHI, S., VADHIYAR, S. "PartRePer-MPI: Combining Fault Tolerance and Performance for MPI Applications", out. 2023. doi: 10.48550/arXiv.2310.16370.

[35] SUBASI, O., YALCIN, G., ZYULKYAROV, F., et al. "Designing and Modelling Selective Replication for Fault-Tolerant HPC Applications". In: *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 452–457, maio 2017. doi: 10.1109/CCGRID.2017.40.

[36] PAUL, S. R., HAYASHI, A., SLATTENGREN, N., et al. "Enabling Resilience in Asynchronous Many-Task Programming Models". In: Yahyapour, R. (Ed.), *Euro-Par 2019: Parallel Processing*, v. 11725, *Lecture Notes in Computer Science*, pp. 346–360, Cham, 2019. Springer International Publishing. ISBN: 978-3-030-29400-7. doi: 10.1007/978-3-030-29400-7_25.

[37] GUPTA, N., MAYO, J. R., LEMOINE, A. S., et al. *Implementing Software Resiliency in HPX for Extreme Scale Computing*. Relatório Técnico SAND-2020-3975R, Sandia National Lab. (SNL-CA), Livermore, CA (United States), abr. 2020.

[38] SUBASI, O., YALCIN, G., ZYULKYAROV, F., et al. "A Runtime Heuristic to Selectively Replicate Tasks for Application-Specific Reliability Targets". In: *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 498–505, set. 2016. doi: 10.1109/CLUSTER.2016.54.

Apêndice A

Código Fonte

O código-fonte do Work Queue modificado utilizado neste trabalho está disponível em https://github.com/Lab-COMPASSO/cctoolsSpeculative/archive/refs/heads/speculative_exec.zip.

De forma semelhante, o Wavefront modificado está disponível em https://github.com/rcoacci/work-queue-examples/archive/refs/heads/speculative_exec.zip.

Caso se deseje cópias de trabalho que utilizem controle de versão, devem ser utilizados os comandos:

```
git clone -b speculative_exec https://github.com/Lab-COMPASSO/cctoolsSpeculative.git
git clone -b speculative_exec https://github.com/rcoacci/work-queue-examples.git
```

Maiores detalhes sobre o uso do Git para controle de versão estão fora do escopo deste trabalho, mas podem ser encontrados em https://git-scm.com/book/en/v2.