Relatório do HW2 - CP

Diogo Domingues nº42950 e João Domingues nº41978

November 28, 2017

Abstract

Este trabalho foca-se em implementar, analisar e discutir os resultados de diferentes versões concurrentes de uma lista ligada. As diferentes versões vão fazer uso de diferentes estratégias e metodologias. O objetivo final é o melhoramento da performance (operation/second) sem perder a funcionalidade.

1 Introduction

Neste trabalho, foi-nos pedido para paralelizar a implementação de um programa em Java. Este programa em Java usa uma lista ligada para implementar um conjunto de inteiros. Foi-nos atribuido um codigo funcional e sequencial desta lista.
Em cada execução são apresentadas as estatísticas do algoritmo (operações por segundo, total de operações e total de operações por tipo de operação).
A versão que nos foi atribuida corre inicialmente um tempo de aquecimento antes de recolher as métricas.
O objetivo do trabalho foi desenvolver diferentes versões, concurrentes, da lista ligada e perceber sobre que condições é que cada versão funciona melhor.

2 Approach

Inicialmente começamos por correr o ficheiro sequencial "IntSetLinkedList" para ganharmos uma noção maior sobre como resolver o problema de implementar concurrentemente a mesma linkedlist de diversas maneiras.

Para a lista "IntSetLinkedListSynchronized", ou seja, a linked list concurrente implementada através do uso de synchronized statements, decidimos modificar os tipos dos metodos add(), remove() e contains() para que estes fossem metodos synchronized.

No que toca à lista "IntSetLinkedListGlobalLock", versão concurrente da lista ligada que faz uso de uma unica lock global, decidimos criar a lock global e fazer [lock()] quando os metodos add(), remove() ou contains() são chamados e desbloquear [unlock()] antes de o metodo retornar algo. Deste modo quando está a ser feita uma leitura ou uma escrita na lista, apenas uma thread está a correr.

Na implementação seguinte que diz respeito à versão da linked lista com uma global read and write lock, ou seja, "IntSetLinkedListGlobalRWLock", decidimos criar a lock do tipo read and write globalmente para depois usar (writelock().lock()) quando os metodos de escrita, add() e remove() são chamados para no final destes, desbloquear a lock com (writelock().unlock()). Para o metodo de leitura contains(), bloqueamos as thread com (readlock().lock()) e antes de returnar o resultado, desbloqueamos as mesma com (readlock().unlock()).

Para a implementação da lista "IntSetLinkedListPerNodeLock", versão concurrente da lista ligada que usa uma metodologia de hand-over-hand, foi implementado uma lock para cada nó. Esta implementação de uma lock no nó vai permitir várias threads executarem os diversos métodos de escrita e de leitura ao contrário das implementações anteriores. Aquando a execução dos métodos add(), remove() e contais(), existe sempre um ciclo para encontrar a posição do nó a adicionar, do nó a remover ou do nó a encontrar. Nesta implementação, é necessário fazer (lock()) sobre os nós que estão a ser percorridos e que apresentam a possibilidade de serem alterados (possibilidade de existir escrita). Quando, durante os ciclos, passamos pelo nó e sabemos que já não vai ser alterado, então desbloqueamos a lock do nó, através do (unlock()), para que outras threads sejam desbloqueadas e possam percorrer o

nó libertado.

Uma otimização da anterior implementação passa por percorrer primeiro a lista para descobrir onde é que o nó deverá ser adicionado, no caso do método add(), qual o nó que deve ser removido, nocaso do método remove(), e qual a posição onde o nó poderá existir, no caso do método contains(). Após esta verificação inicial da lista, os nós sobre os quais serão feitas as alterações (escrita) ou sobre os quais são pedidas informações (leitura) são bloqueados através das suas locks. Como durante o ciclo de verificação os nós não estão protegidos, é necessário agora voltar a validar se os nós continuam na lista e são vizinhos. Se a condição se manter, faz-se a escrita ou leitura, se a condição não se manter é necessário repetir a verificação para decidir a posição a colocar/remover/encontrar o nó. Antes de retornar algum valor dos métodos, os nós bloqueados são desbloqueados. Este tipo de implementação foi feito na classe "IntSetLinkedListOptimisticPerNodeLock".

A versão concurrente da linkedlist que apresenta uma implementação similar à anterior com a excepção que usa uma estratégia lazy é a "IntSetLinkedListLazyPerNodeLock". Para esta variação, implementámos uma variavel booleana em cada nó que indica se o nó está presente na lista ou não. Após a remoção de um elemento da lista, esta variavel é alterada. A existência desta variavel permite tornar o metodo contains() lock-free, uma vez que a variavel que marca a presença do nó na lista é alterada antes de existir alteração da lista. Neste sentido o método contains() apenas precisa de percorrer todos os elementos ligados pela lista e se encontrar o elemento verifica se este ainda está na lista através da variavel booleana.

Finalmente a versão final da linked list é a "IntSetLinkedListLockFree" que implementa uma lista ligada concurrente sem locks. Para esta implementação foi necessário recorrer a objetos do tipo Atomic Markable Reference para fazerem as operações de adição, remoção e alteração da variável que marca a presença do elemento na lista, atomicamente atravésd o metodo CompareAndSet(). Os nós passaram a apontar para um objeto "Atomic Markable Reference" que contêm a referência para o proximo nó da lista. Quando os métodos add(), remove() e contains() são executados, é necessário descobrir entre que nós é que o nó que vai ser lido ou escrito se encontra. Para tal fizemos uso de um método auxiliar find() que nos devolve um objeto do tipo window com dois nós vizinhos (o previous e o current). O metodo find() irá percorrer todos os nós da lista até encontrar os nós vizinhos que estão entre o valor pretendido e durante essa procura elimina da lista todos os nós marcados.

3 Validation

Para encontrarmos os erros que apareciam à medida da implementação, utilizámos o debugger do eclipse que nos permitiu resolver numa primeira instância os erros.
A seguinte validação foi feita com a ajuda de um método auxiliar validate() que após a execução dos vários testes é capaz de verificar se a lista está ordenada, se apresenta ou não duplicados. Tal não deverá acontecer se a implementação do código estiver bem feita.
Depois de feita a validação e de executar diversas vezes localmente cada implementação com diversos parâmetros passámos a testar a execução das implementações na máquina de 16 cores.
Para cada tipo de implementação da lista, escolhemos executar 5 vezes a implementação com o mesmo parâmetro de % de escrita. Para parâmetro de tempo escolhemos 10s fixos e para parâmetro de % de escrita escolhemos 0,25,50,75,100.

4 Evaluation

Os gráficos encontram-se depois das conclusões.

5 Conclusions

Em suma foi possível confirmar os resultados esperados. Esperáva-mos que as implementações iniciais com synchronized statements e global lock fossem capazes de gerir a lista ligada concurrentemente, e que a sua performance fosse igual ao pior à versão sequencial da lista. Com o uso de uma read and write lock global, esperáva-mos que a performance e comportamento da lista melhora-se significativamente relativamente Às implementações anteriores em casos de escrita menor que 50%. Com a utilização de implementações que bloqueam os nós, a nossa expectativa seria de que a performance deveria melhorar no geral, independentemente da % de escrita. Para uma escrita de 25% podemos verificar pelos gráficos que tanto o lazy como o lock-free são as melhores versões a executar. Para uma escrita de 50% e 75% também verificámos

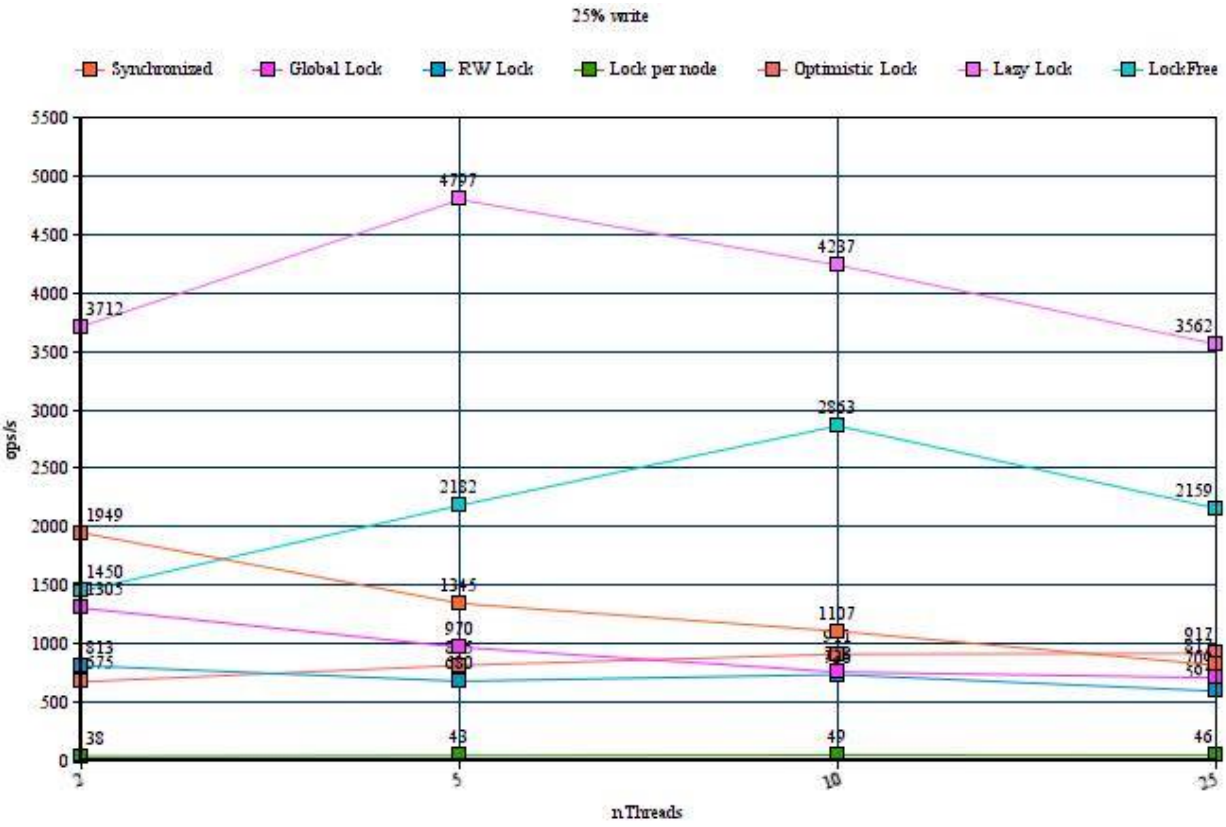

Figure 1: Esta imagem compara diferentes versões com 25% de write.

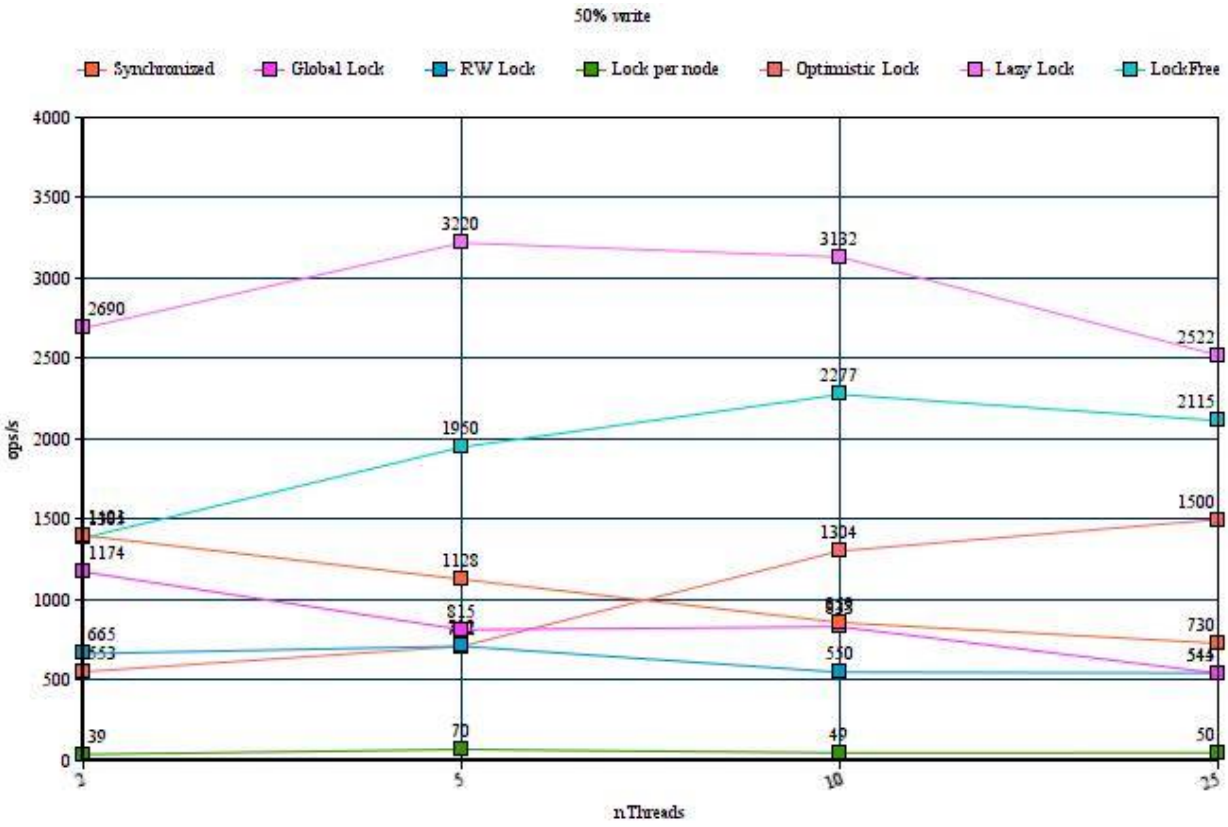

Figure 2: Esta imagem compara diferentes versões com 50% de write.

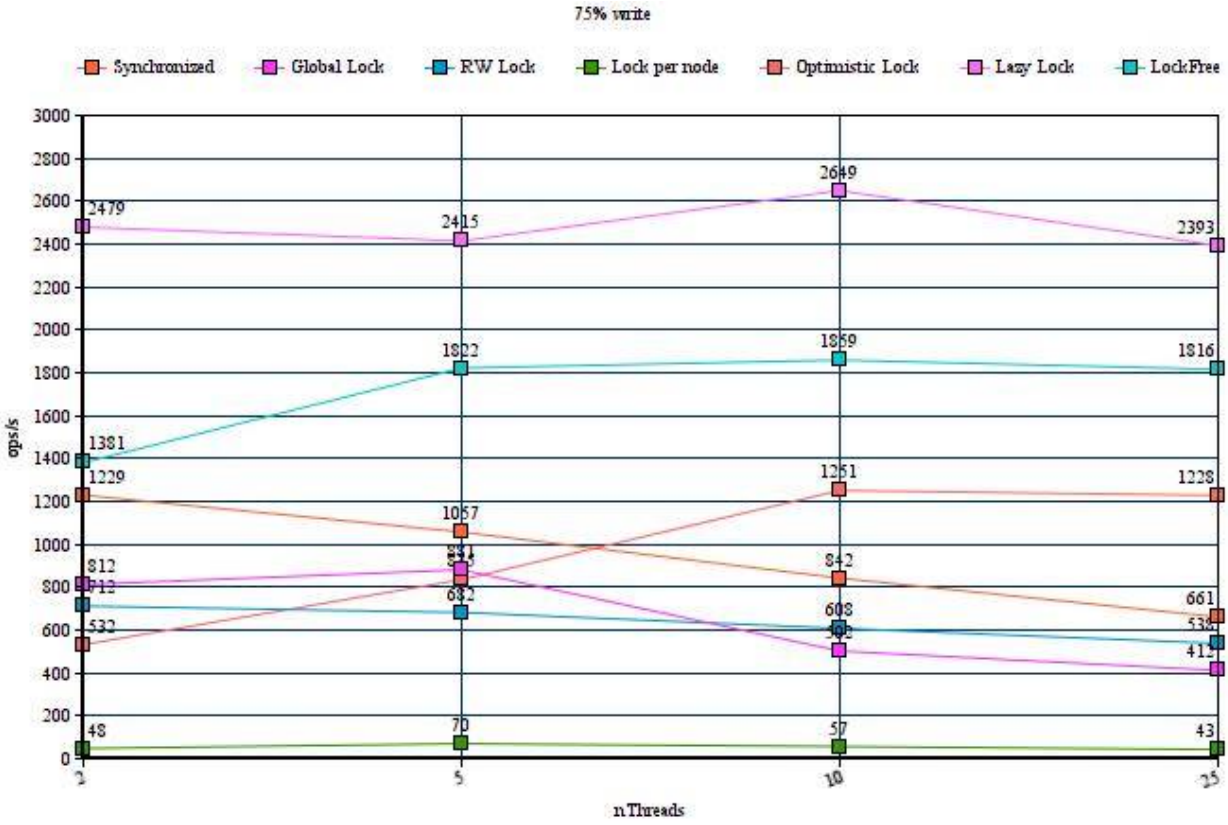

Figure 3: Esta imagem compara diferentes versões com 75% de write.

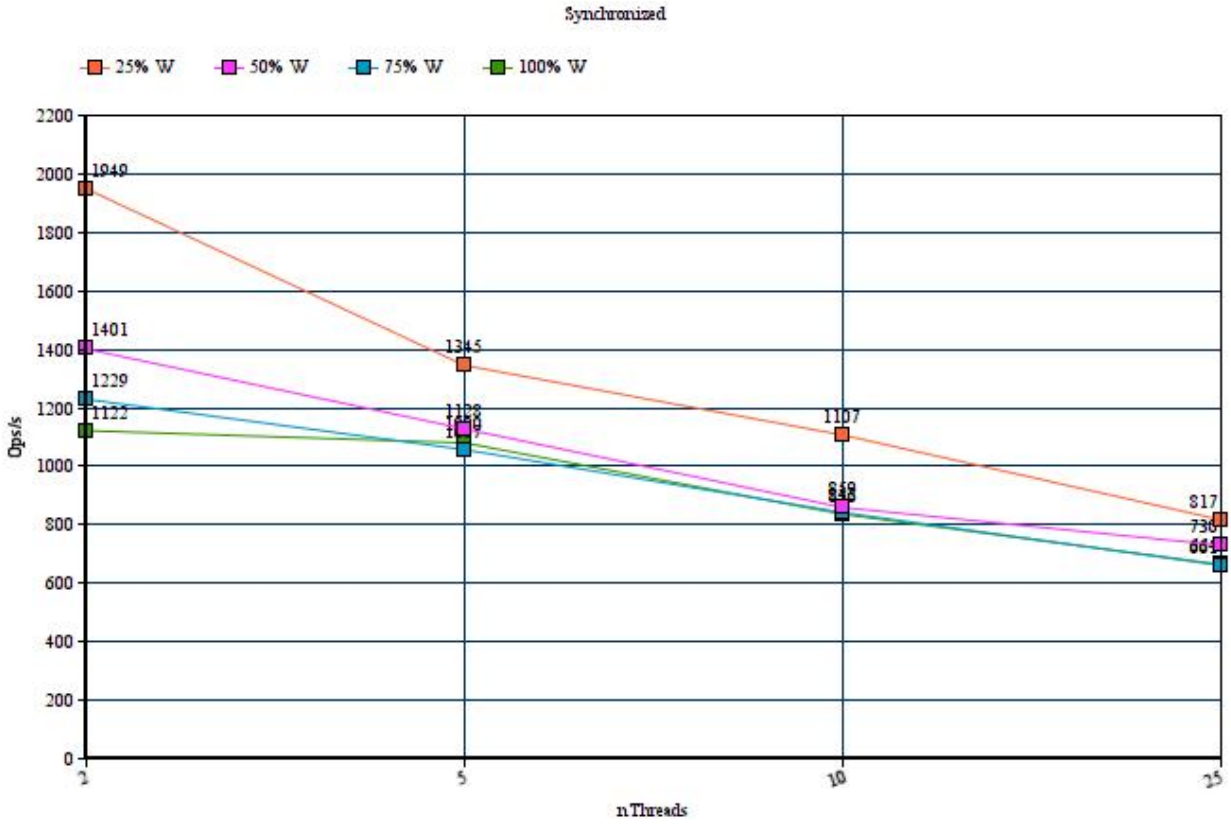

Figure 4: Esta imagem compara a performance da mesma versão com diferentes % de write.

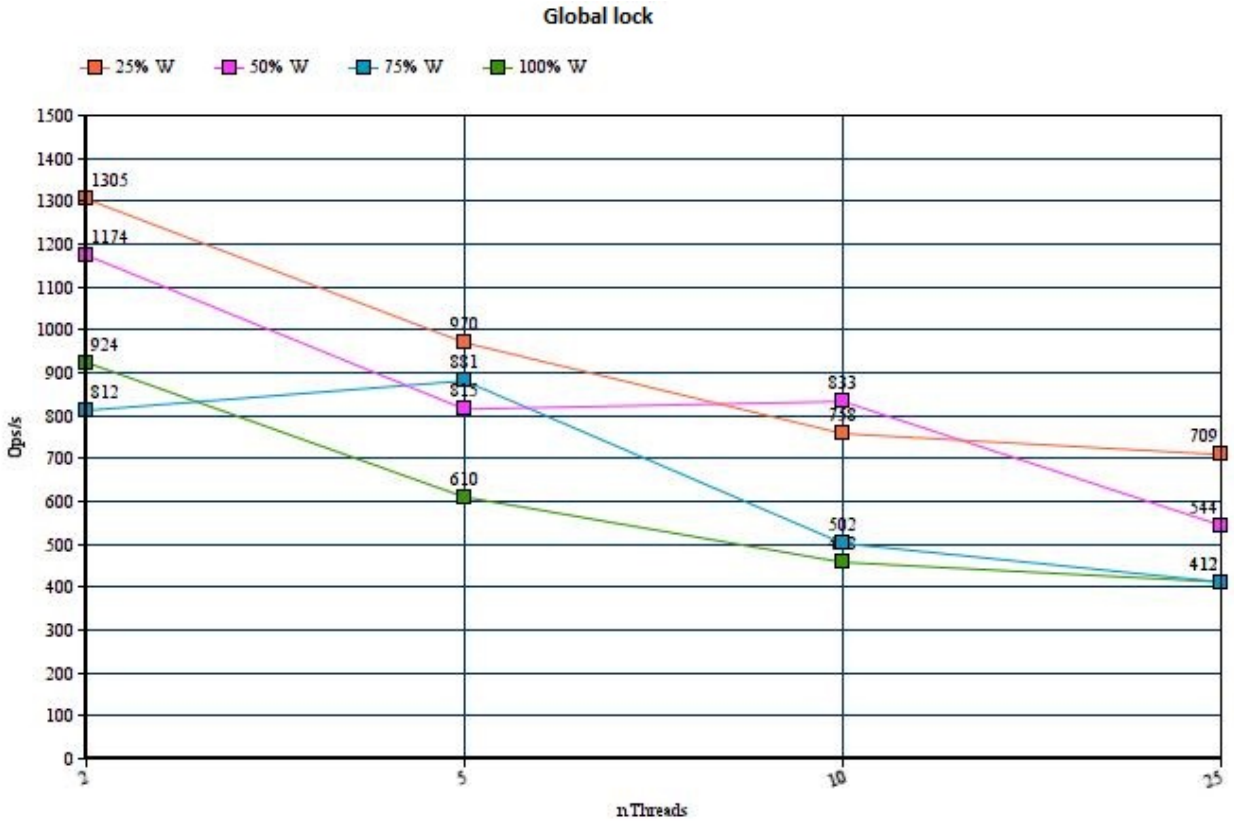

Figure 5: Esta imagem compara a performance da mesma versão com diferentes % de write.

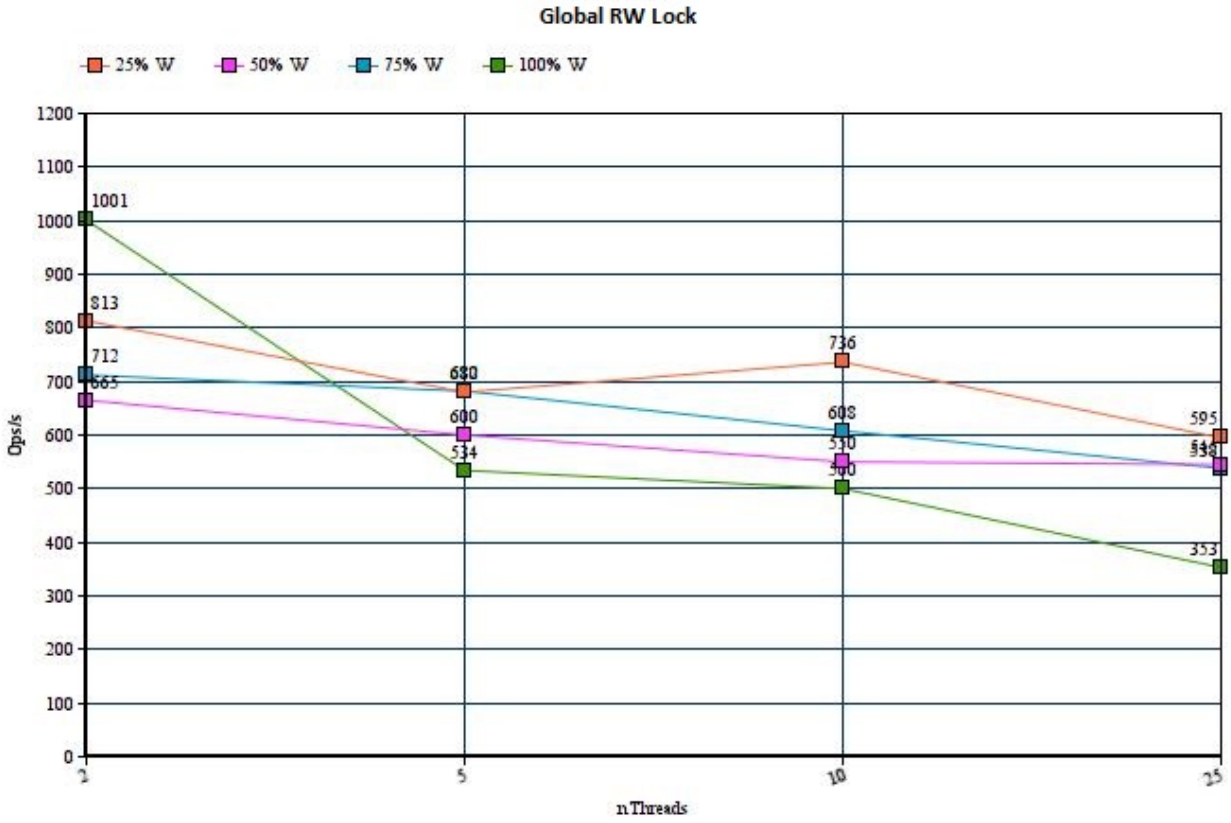

Figure 6: Esta imagem compara a performance da mesma versão com diferentes % de write.

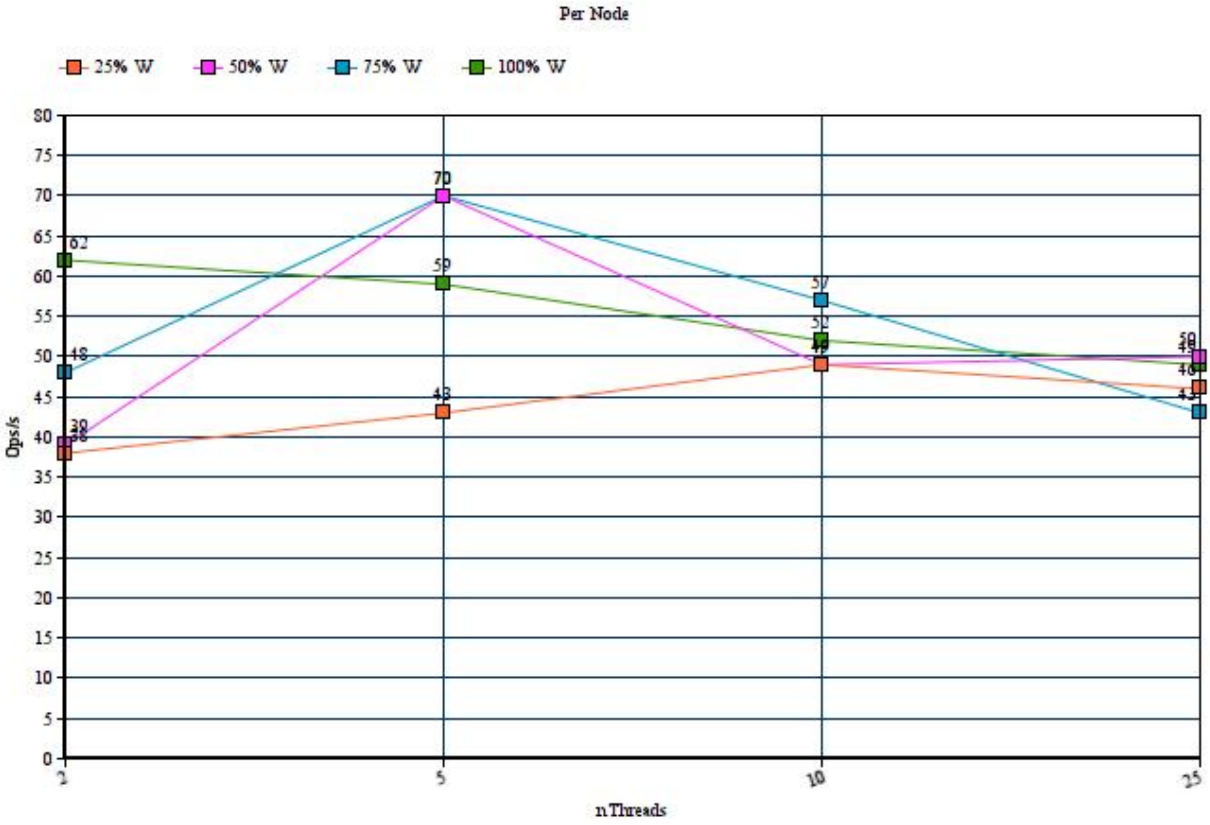

Figure 7: Esta imagem compara a performance da mesma versão com diferentes % de write.

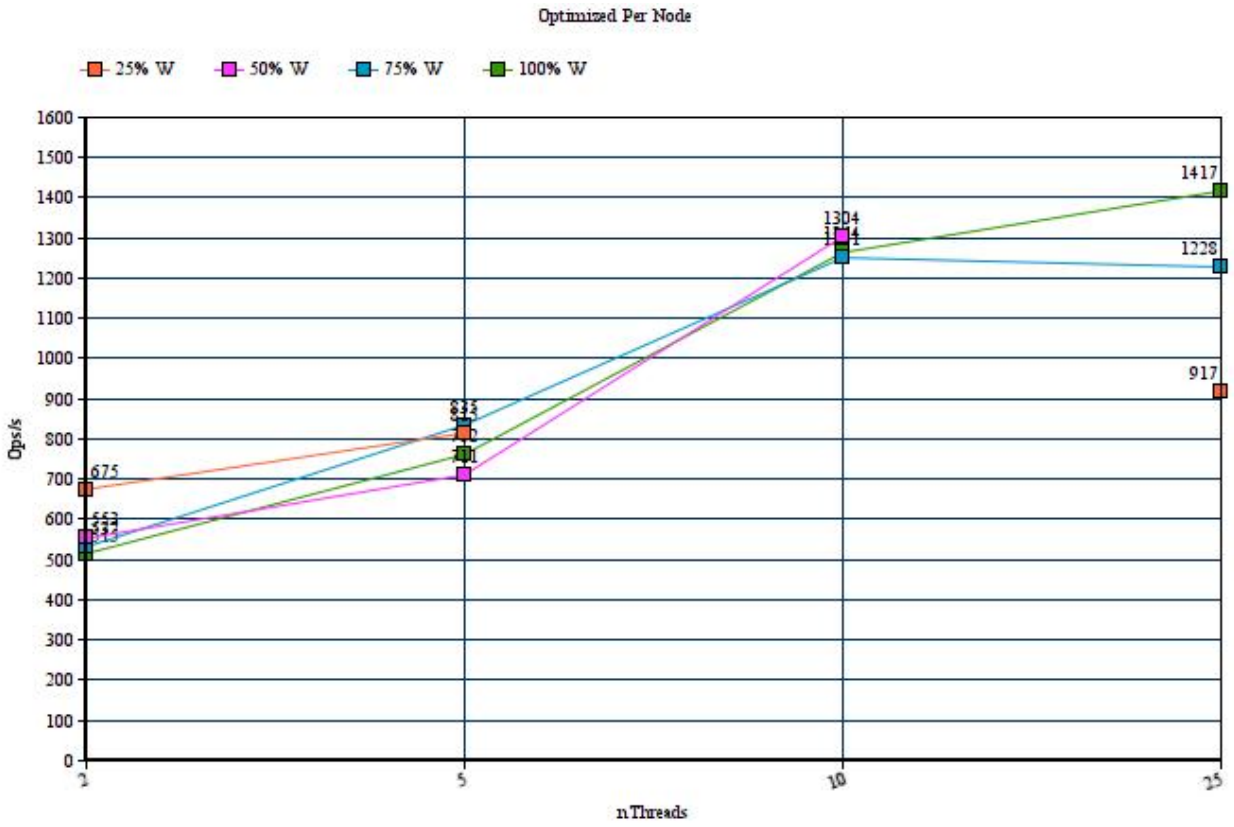

Figure 8: Esta imagem compara a performance da mesma versão com diferentes % de write.

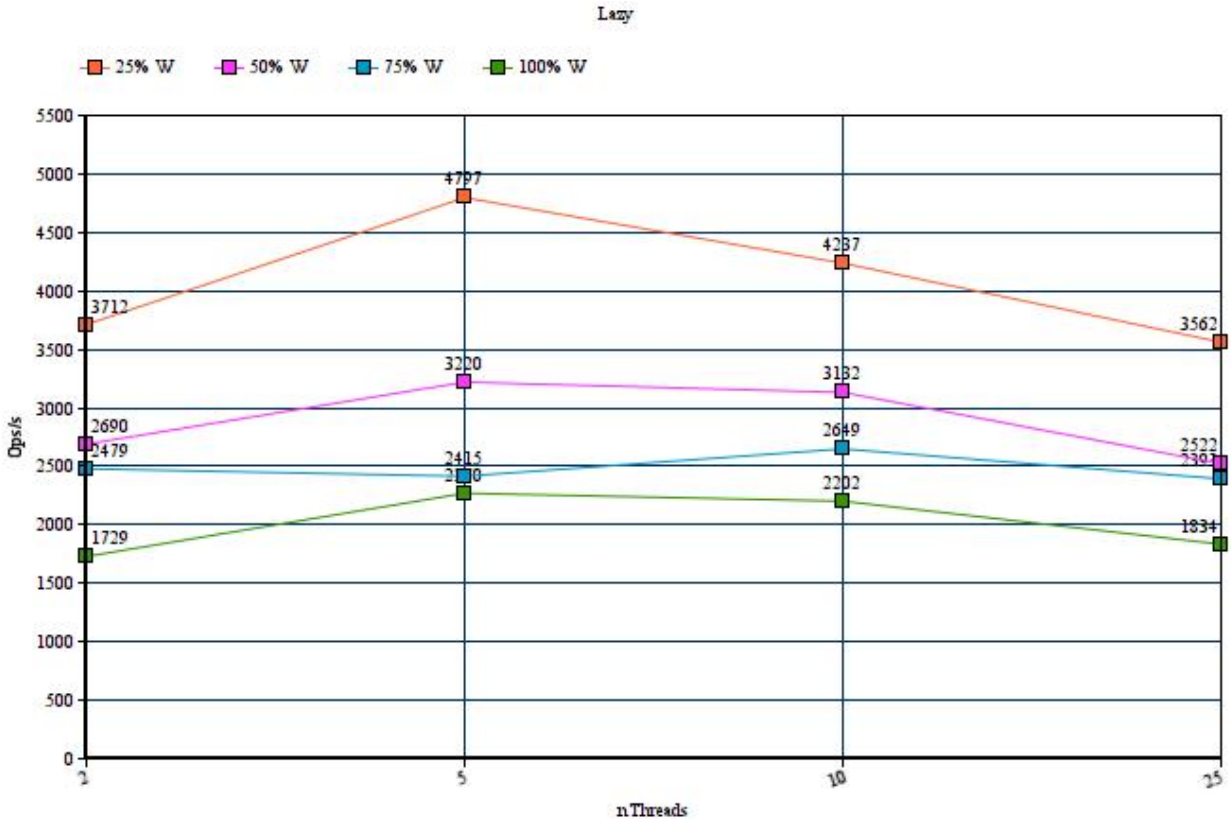

Figure 9: Esta imagem compara a performance da mesma versão com diferentes % de write.

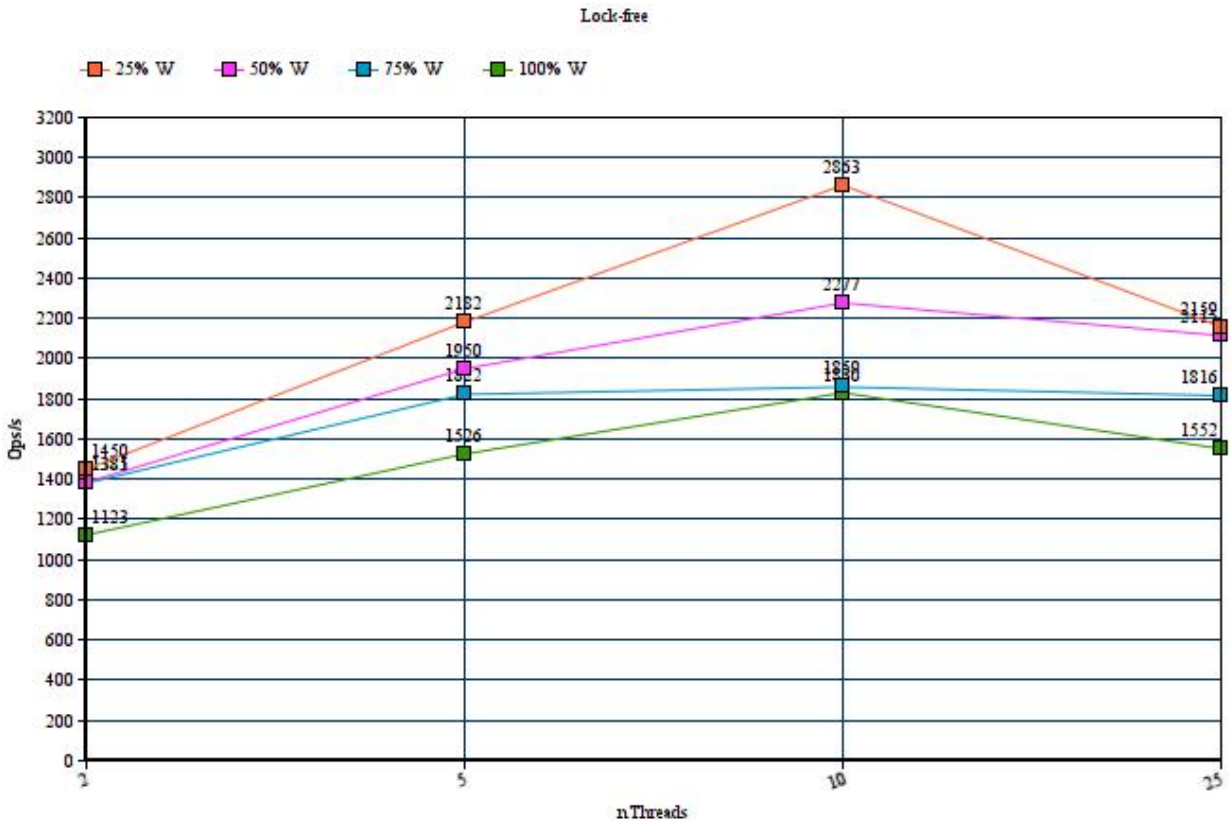

Figure 10: Esta imagem compara a performance da mesma versão com diferentes % de write.

a mesma situação. No entanto não esperávamos o resultado obtido para a versão do lock per node, uma vez que as operações por segundo para esta versão foram muito baixas. NO que toca à performance individual dos algoritmos, concluimos que: a versão synchronized funciona melhor numa situação de menos escritas, a versão global lock também apresenta uma melhor performance quanto menor for a escrita. No caso da versão optimized lock per node podemos verificar que com o numero de threads utilizadas não observámos grandes disparidades. A versão lazy apresenta uma melhor performance quanto menor for o numero de operações de escrita e finalmente a versão lock-free também teve o mesmo comportamento.

6 Acknowledgments

Gostariamos de agradecer ao Gonçalo Marcelino por nos ter ajudado a ultrapassar um problema que obtivemos na implementação da lista lock-free. O problema consistia no acesso a um objeto AtomicMarkableReference que estava a ser mal inicializado.

7 Bibliography

[CP-Book] Chapter 9 Linked Lists: The role of locking.