Abstract

O intuito deste projeto é implementar em Java e comparar a performance de sete algoritmos de sincronização num conjunto de inteiros implementado por uma LinkedList, nomeadamente, Synchronized, lock Global, lock Read-Write, Hand-Over-Hand locking, estratégia Optimistic, a Lazy e a Lock-Free. Para analisar a performance de cada um destes algoritmos realizou-se diversos testes, nos quais mede-se a quantidade de operações por segundo de cada algoritmo, variando a percentagem de leituras, o número de threads e o tamanho do conjunto de inteiros.

Concluiu-se que, num conjunto com 4000 elementos, os algoritmos Coarse-Grained superam o algoritmo Hand-Over-Hand mas actuam pior do que restantes (Lock-Free, Lazy e Optimista). Quando a lista contém, inicialmente, 256 elementos o desempenho do Optimistic Lock torna-se pior do que os Coarse-Grained. Finalmente, o algoritmo Lock-Free, devido à sua implementação em Java e não noutra linguagem mais baixo nível, revela resultados inferiores ao Lazy.

Introdução

No âmbito da cadeira de Concorrência e Paralelismo foi proposto que se implementasse múltiplas técnicas de sincronização entre acessos concorrentes a um conjunto de inteiros implementado recorrendo a uma LinkedList com nós sentinela. O que torna este problema interessante e desafiante é encontrar as vantagens e desvantagens de cada algoritmo num determinado contexto.

Recorrendo ao programa de *benchmark* é necessário avaliar e comparar a performance, com diferentes parâmetros, das seguintes técnicas: sequencial, utilizar a declaração *Synchronized*, uma só *lock* Global, *lock Read-Write*, *Hand-Over-Hand locking*, estratégia Optimistic, estratégia *Lazy* e finalmente a estratégia *Lock-Free*.

Abordagem

É importante descrever as várias técnicas de sincronização, incluindo as particularidades das implementações de cada uma:

Lock Global: Esta estratégia inclui-se numa sincronização Coarse-Grained, isto é, dada uma implementação sequencial da classe adiciona-se um campo lock e assegura-se que cada chamada a um método adquire o lock e após executar todas as operações liberta esse mesmo lock, impedindo qualquer outra thread de realizar operações nesse objeto. No caso dos níveis de concorrência serem baixos esta solução é uma excelente forma de implementar a lista, o problema surge quando há muitos acessos concorrentes.

Synchronized: Tal como no Lock Global, o tipo de sincronização é Coarse-Grained. Este é uma construção do Java que previne interferência de threads e inconsistências na memória, isto é, se um objeto é visível a mais que uma thread, todas as leituras e escritas às variáveis serão feitos através de métodos ***synchronized.*** [3]

Lock Read-Write: Este algoritmo, tal como os dois anteriores tem uma sincronização Coarse-Grained. Apesar de utilizar a mesma estratégia tem traços distintivos. Neste caso, o Lock ReadWrite mantém um par de locks, um para operações de leitura e outro para escritas, tal como o nome sugere. Este último permite que várias threads acedam ao recurso, enquanto não existirem escritas. Por outro lado, o lock das escritas é exclusivo garantindo assim que outras threads acedem sempre à versão atualizada. A implementação proposta deste algoritmo utiliza um lock da classe ReentrantReadWriteLock[2].

Hand-Over-Hand Locking*:* Este é o primeiro algoritmo apresentado que tem uma sincronização Fine-Grained. Isto permite melhorar a concorrência com locks ao nível do nó e não na lista inteira. Desta forma, à medida que se atravessa a lista bloqueia-se o acesso ao nó predecessor e ao corrente para evitar situações como: sendo **a** e **b** dois nós e **a** aponta para **b**, não é seguro remover o lock de **a** antes de fazer lock de **b** pois outra thread poderá ter removido **b** no intervalo entre o unlock do **a** e o lock do **b**.

Estratégia Optimista: O propósito desta estratégia é reduzir a sequência potencialmente longa de libertações e obtenções de locks. Uma forma de reduzir os custos de sincronização é ser "optimista" como a definição indica, isto é, pesquisar sem locks, bloquear os nós encontrados, e de seguida confirmar que os nós

André Lopes nº45617
Nelson Coquenim nº45694
Grupo 23

ainda são válidos. Caso sejam válidos realiza-se a operação que se pretendia. Caso não sejam válidos de momento, liberta os locks e reinicia o algoritmo.

Estratégia *Lazy:*Pode-se afirmar que a estratégia "optimista" já fornece um algoritmo bastante poderoso. Tendo uma desvantagem, que é o facto das leituras (i.e contains) serem bloqueantes, isto prejudica a performance se as leituras forem mais que as escritas. Portanto, o objectivo desta estratégia será eliminar os locks do método contains(). Para este efeito, adiciona-se em cada nó uma *flag* que indica se o nó foi removido. Com isto, o algoritmo tem de manter o invariante de que todos os nós não marcados estão acessíveis a partir da cabeça, logo, se uma thread não encontra o nó a partir da cabeça ou o nó está com a *flag* a true, significa que nó não está na lista. Em relação às inserções, encontra-se os nós entre os quais será inserido o novo nó, bloqueia-se os mesmos e insere-se o nó. Quanto às remoções agora serão feitas em dois passos: remoção lógica e física. Primeiro procede-se à remoção lógica, que corresponde em marcar o nó como removido, e de seguida a remoção física, que consiste em redirecionar o campo next do predecessor. Por outro lado, nas leituras (contains) não é necessário bloquear nenhum nó, limitando a sua execução a percorrer a lista e caso encontre o valor verificar se esse nó está marcado.

Estratégia *Lock-Free*: Por fim, um algoritmo com uma filosofia diferente dos referidos anteriormente, este contrariamente aos anteriores não recorre a locks para garantir exclusão mútua nos acessos a uma determinada secção crítica. De facto, a forma como é garantida a exclusão é juntar a marcação de nós como logicamente removidos, a atomicidade de leituras seguidas de escritas. Esta propriedade é obtida através de primitivas fornecidas pelo hardware, neste caso o *compare and swap* (CAS(A,B,C)) que verifica se o registo A tem o valor B, caso tiver A passa a ter o valor C e retorna true, caso contrário retorna falso. Esta construção será utilizada para mudar os campos next verificando se o valor atual não foi mudado entretanto. Certamente esta solução sem uma pequena alteração não funcionará, pois é necessário que as variáveis dum nó não possam ser atualizadas quando o mesmo já foi lógica ou fisicamente removido da lista. Percebe-se aqui que é necessário tratar a variável *next* e a *flag* de logicamente removido como uma unidade atómica. Adicionalmente, a remoção física de nós logicamente marcados será feita aquando da pesquisa pelos nós necessários para as inserções ou remoções.

No que toca a implementação, a forma de tornar as atualizações da *flag* conjuntamente com o do campo next atómicas foi recorrer à classe Java **AtomicMarkableReference**<T> [5].

Hipóteses a validar:

I. É expectável que tanto a técnica **Synchronized** e ***Global Lock*** apresentem resultados semelhantes visto que ambas só permitem uma thread na zona crítica.

II. Com uma performance superior será a ***Lock Read-Write*** visto que esta já permite leituras em paralelo na zona crítica. A diferença no throughput entre esta e as últimas duas anteriores manifestará-se mais com o aumento das leituras.

III. O ***Hand-Over-Hand Locking*** devido ao elevado número de locks que efectua é previsto que o seu desempenho deteriore-se com o aumento no número de threads, e possivelmente será pior que as alternativas anteriores.

IV. A estratégia ***Optimistic*** como não realiza locks no percorrer da lista e mesmo apesar de efectuar o percurso, no mínimo, duas vezes ir-se-á, possivelmente, observar uma melhoria no desempenho em relação às técnicas referidas até este ponto. Mas apenas se a complexidade temporal de percorrer duas vezes a lista sem locks for menor do que percorrer uma vez a lista com locks.

V. A estratégia ***Lazy*** em comparação com a alternativa anterior já só percorre no mínimo uma vez. Sendo que no método contains, uma vez que não contém locks e é *wait-free* (i.e. retorna sempre num número limitado de passos) faz com esta estratégia seja preferível à *Optimistic*.

VI. Por último, a estratégia **Lock-Free** não sendo bloqueante prevê-se que esta detenha um *throughput* superior às anteriores. Em contrapartida, acrescenta a necessidade de suportar a modificação atómica de uma referência conjuntamente com uma marca booleana. Adicionalmente, obriga que percursos

pela lista realizem a limpeza física de nós, que potencialmente pode não ter sucesso resultando em diversas tentativas de limpeza de nós mesmo para métodos cujo as mudanças sejam noutros nós distantes. Desde modo a diferença do throughput entre esta estratégia e a solução anterior, teoricamente, decresce com o aumento da concorrência.

Validação

De modo a avaliar o desempenho de cada técnica de sincronização realizou-se diversos testes para estas, sendo que a métrica de desempenho utilizada será o número de operações por segundo (*throughput*). Os testes realizaram-se numa máquina de 16 cores físicos e com 32GB de RAM e apenas com um 1GB de RAM atribuído à JVM. De modo a reduzir o *noise* durante a execução dos testes, todas as medidas são médias de cinco execuções de 1000 milissegundos cada. Serão realizados testes com uma lista de 256 e outra de 4000 elementos. Suplementarmente, realiza-se em todos os teste uma fase de *warm up* de 2000 milissegundos onde a lista é populada e onde não é guardada nenhuma estatística. Estes testes permitiram avaliar os algoritmos com diferentes pesos do lock no tempo de execução, com uma lista maior cada operação durará mais tempo o que torna o tempo de execução do lock menos significativo e vice-versa. Para os diferentes tamanhos da lista pretende-se variar a percentagem de escritas, testando-se com uma percentagem baixa, média e alta de escritas, respectivamente, 10%, 50% e 90%. Adicionalmente, varia-se o número de threads, com múltiplos de dois, entre 1 e 16. Em último, e para cada tamanho da lista, realiza-se um teste que irá medir o throughput das várias técnicas para diferentes percentagens de leituras, com múltiplos de 10, entre 5% e 90% utilizando sempre 16 *threads*.

Recorrendo aos gráficos gerados dos vários testes, através da biblioteca matplotlib[4] do python será possível validar ou renunciar as hipóteses enunciadas anteriormente na abordagem visto que estes demonstram o *throughput* de cada técnica de sincronização.

Discussão dos Resultados

Importa agora cruzar as hipóteses formuladas com os resultados obtidos. Um dos aspectos que não se previu, nos algoritmos que usam locks, foi o facto da performance piorar em certos casos quando é aumentada de 1 para 2 threads. Este dado pode ser causado pelo facto do lock, no caso dos Coarse-Grained, ou locks dos nós no caso dos Fine-Grained ser(em) partilhado(s) em memória, portanto, só com uma thread, nas operações de lock e unlock são hits na cache do processador pois não há necessidade de sincronizar as caches entre vários processadores, com duas surge a exigência de sincronização o que poderá acrescentar um *overhead*. O que não foi provado pois não se encontrou um profiler de Java que permitisse validar esta hipótese, analisando os *hit and misses* das caches do(s) processador(es).

Validação das hipóteses tendo em conta uma lista inicial de 4000 elementos:

I. No geral, o Synchronized apresenta uma melhor performance que o GlobalLock, não sendo esta diferença significativa.

II. Através da Fig.8, pode-se observar que, na técnica Read-Write Lock, um aumento das leituras leva a um aumento do throughput. Revelando-se este melhor que outros algoritmos Coarse-Grained quando a percentagem de escritas é menor que 10%. Adicionalmente, observa-se que a performance deste tende a aproximar-se ao GlobalLock com o aumento da percentagem de escritas, isto porque reduz-se o número de leituras em paralelo.

III. Nos figuras 2, 5, 6 e 8 verifica-se que o Hand-Over-Hand devido ao elevado número de locks a sua performance mantém-se constantemente baixa, independentemente do número de escritas e threads (Fig.8). Estando a razão para este facto na formulação desta hipótese.

IV. Analisando o OptimisticLock com 10% de leituras (Fig.2) está relativamente próximo do LazyLock, sendo melhor do que o Lock-Free, os algoritmos Coarse-Grained e o Hand-Over-Hand. A performance desta técnica “optimista” afasta-se da Lazy à medida que se aumenta a percentagem de leituras, isto é devido ao contains não ser wait-free. Também é observável que, aumentando o número de threads com uma percentagem de leituras superior a 50% (Fig. 4 e 6), a partir dum certo

número de threads o Lock-Free torna-se melhor que o Optimistic pois, tal como o LazyLock, o seu contains é wait-free.

V. Em relação ao LazyLock, este comporta-se bastante melhor que o OptimistLock. De facto, com uma percentagem alta de leituras (Fig.6) esta diferença ainda é mais evidente, sendo com 16 threads 4x melhor. O motivo para tal é o facto do contains do algoritmo Lazy ser wait-free. Por outro lado, há uma excepção (Fig.2), com 10% de leituras e 16 threads o OptimistcLock tem melhor performance podendo-se especular que se trata de noise. Adicionalmentem este algoritmo exibe um aumento do throughput relacionado com o aumenta das threads.(Fig.2, 4, 6)

VI. Contrariamente ao previsto, a performance do Lock-free foi inferior à do LazyLock, o motivo será o uso da classe AtomicMarkableReference para efetuar operação compare and set. Esta acarreta um overhead em relação a uma implementação em C ou C++ que disponibiliza operações baixo nível para este efeito. Esta diferença acentua-se numa lista inicial de tamanho 4000 onde as diferenças entre o LazyLock e o Lock-Free são notórias. Ainda assim verifica-se um incremento do throughput com o aumento do número de threads

Em todos os algoritmos de sincronização Coarse-Grained, nomeadamente Synchronized, Global Lock e no Global Read Write (apenas quando este tem pouca leituras), pode-se depreender que o impacto da variação do número de threads na performance de cada é mínimo isto porque estes algoritmos fazem com que a lista inteira seja locked, eliminando assim a concorrência nos métodos da mesma.

Analisando os resultados com a lista de 256 elementos, pode-se observar que alguns algoritmos mantiveram uma performance semelhante ao resultados da lista de dimensão maior, especificamente, as implementações do Global, Synchronized, Hand-Over-Hand e Read-Write obtiveram desempenhos próximos, sendo que o último, como esperado comporta-se melhor para situações com mais leituras (Fig. 5). O *throughput* dos algoritmo Lazy piorou e aproxima-se mais do Lock-Free, possivelmente, porque o tamanho da lista é menor o que aumenta a probabilidade dos conflitos, que por sua vez leva uma performance inferior (Fig. 7,8). A diferença mais evidente é a performance do algoritmo optimista, que com uma lista maior, apresenta resultados entre o Lock Free e os algoritmos Coarse-Grained mas com a lista a 256 elementos fica abaixo dos Coarse Grained sendo apenas melhor que o Hand-Over-Hand este facto pode-se dever ao facto deste percorrer duas vezes a lista, ou seja, o complexidade de percorrer a lista **uma vez com locks foi inferior à de percorrer a lista duas vezes sem locks**[1].

Verifica-se praticamente em todos os algoritmos de sincronização uma melhoria na *performance* quando comparados com a versão sequencial, excepto a técnica Hand-Over-Hand que, no máximo e quando executada por mais do que uma thread, apenas executa 47 000 operações/segundo (lista de 256 elementos) e 4000 operações (lista de 4000 elementos) ficando abaixo dos resultados da versão sequencial, 132 000 ops/s, no extremo de 10% de leituras e 145 000 ops/s com 90% de leituras.

Conclusão

A escolha do algoritmo de sincronização mais indicado para um dado problema irá sempre depender do contexto do mesmo. É necessário ter em conta diversos factores, tais como a quantidade de escritas e leituras, o overhead de implementar uma *atomically markable reference* e o tamanho do problema. Pelos resultados obtidos neste trabalho, conclui-se que os algoritmos de coarse-grained funcionam bem para situações com pouca concorrência, mas se muitos processos necessitarem de aceder ao objecto, surgirão *bottlenecks*, pois obriga-se a que os processos esperem pela sua vez. Apesar disto, estes algoritmos têm uma performance melhor do que o algoritmo Hand-Over-Hand, independentemente, deste último permitir múltiplas threads acederem ao objecto. A performance do algoritmo Hand-Over-Hand é inferior, uma vez que se efectuam locks ao percorrer a lista. O algoritmo optimista mostrou assim ser mais eficiente para listas de tamanhos maiores (4000 elementos). Por último, a estratégia Lazy demonstrou ser a mais eficiente em maior parte dos cenários, mesmo quando comparada com a Lock Free.

André Lopes nº45617
Nelson Coquenim nº45694
Grupo 23

Gráficos **(A partir de 16 threads o throughput estabiliza)**

LinkedList	10% reads	50% reads	90% reads
Size List - 256	132 000	132 000	145 000
Size List - 4000	134 000	135 000	145 000

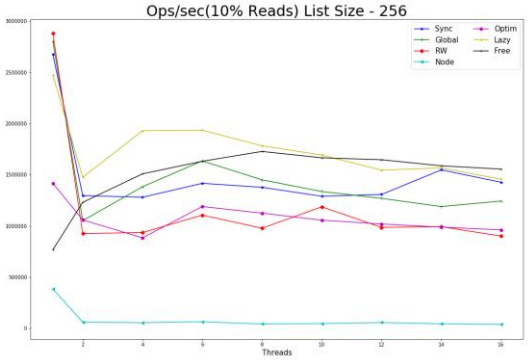

Fig.1

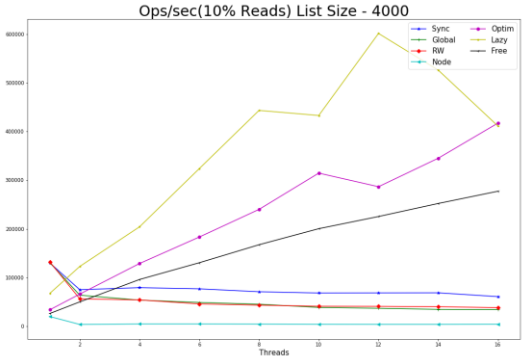

Fig.2

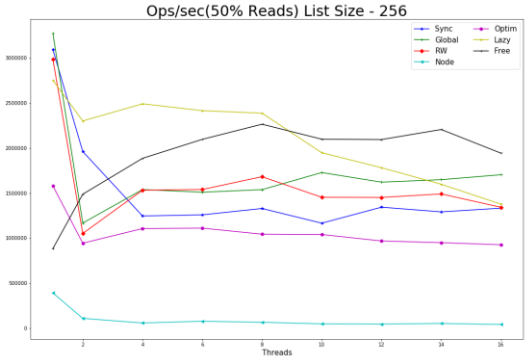

Fig.3

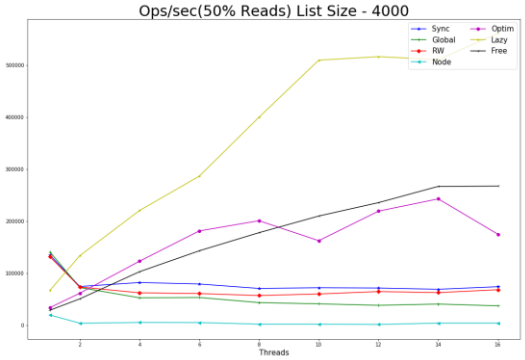

Fig.4

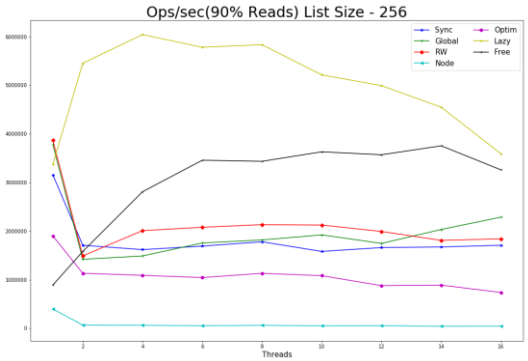

Fig.5

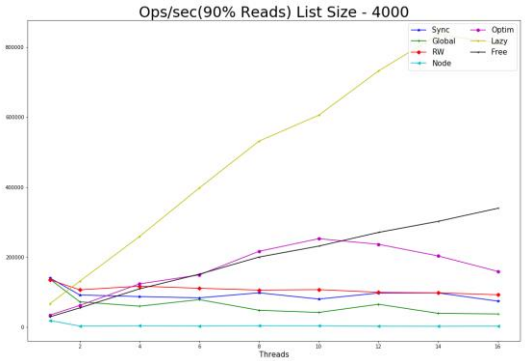

Fig.6

André Lopes nº45617
Nelson Coquenim nº45694
Grupo 23

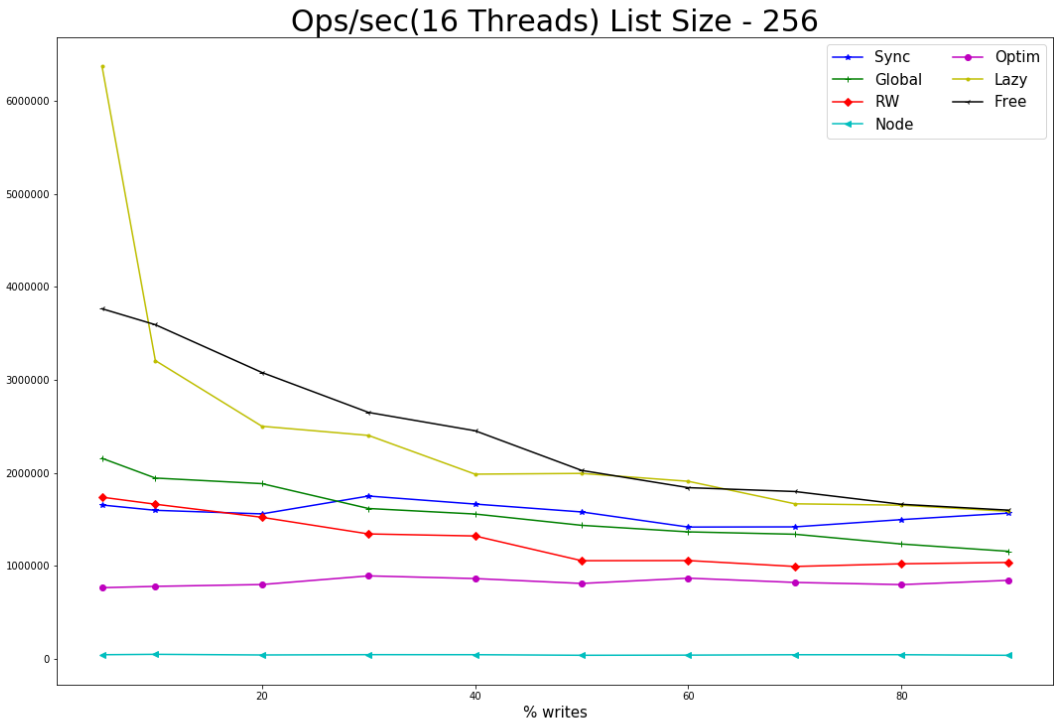

Fig.7

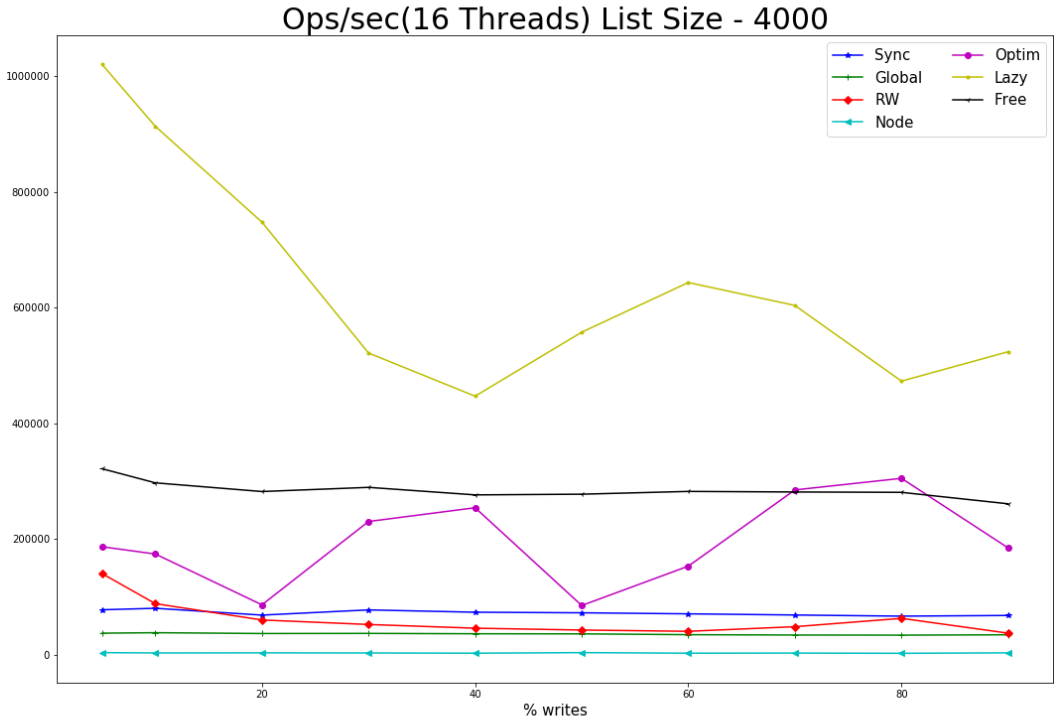

Fig.8

André Lopes nº45617
Nelson Coquenim nº45694
Grupo 23

Agradecimentos

Ambos os elementos do grupo agradecem a disponibilidade dos professores João Lourenço e Bernardo Ferreira em esclarecer dúvidas mas também os colegas que participaram na discussão na plataforma do Piazza.

Bibliografia

[1]Herlihy, M. and Shavit, N. (2012). The art of multiprocessor programming. Burlington, MA: Morgan Kaufmann.

[2]docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html. Oracle
[3]docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html. Oracle
[4]https://matplotlib.org/. John Hunter, 2007.
[5]https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicMarkableReference.html.Oracle