1 Tabela de dispersão concorrente

1.1 Avaliação de desempenho

Tabela 1: Média de resultados de SetBenchmark para LHashSet0 (5 execuções) em Mops/s

N \ Threads	1	2	4	8
256	**14.22**	6.42	6.56	11.92
5 000	1.00	0.71	0.77	0.86
50 000	0.05	0.05	0.08	0.05

Tabela 2: Média de resultados de SetBenchmark para LHashSet1 (5 execuções) em Mops/s

N \ Threads	1	2	4	8
256	13.94	3.76	5.39	5.65
5 000	1.00	0.53	0.53	0.54
50 000	0.05	0.05	0.07	0.05

Tabela 3: Média de resultados de SetBenchmark para LHashSet2 (5 execuções) em Mops/s

N \ Threads	1	2	4	8
256	13.81	4.55	3.30	3.26
5 000	0.99	0.65	0.61	0.93
50 000	0.05	0.11	0.12	0.08

Tabela 4: Média de resultados de SetBenchmark para LHashSet3 (5 execuções) em Mops/s

N \ Threads	1	2	4	8
256	13.71	7.34	6.95	10.34
5 000	0.99	1.60	1.97	**1.71**
50 000	0.05	0.08	**0.14**	0.12

Tabela 5: Média de resultados de SetBenchmark para STMHashSet (5 execuções) em Mops/s

N \ Threads	1	2	4	8
256	3.85	3.76	5.04	5.68
5 000	0.44	0.68	0.70	0.74
50 000	0.04	0.06	0.06	0.06

Condições do teste

- CPU: Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz
- OS: Linux 5.10.0-7-amd64, Debian 5.10.40-1 x86_64
- Executado num tty de texto.

1.1.1 Discussão

Análise de resultados

O STMHashSet apresenta um maior overhead em todos os casos.

Com 1 thread:

- Observa-se um maior overhead em LHashSet0 com um N de 256 do que em LHashSet1;
- Observa-se um maior overhead em LHashSet1 do que em LHashSet2;
- Observa-se um maior overhead em LHashSet3 com um N de 256 do que em LHashSet2.

Com 2 threads:

- Observa-se um maior overhead em LHashSet1 do que em LHashSet0 para N 256 e 5 000;
- Observa-se uma melhoria de LHashSet1 para LHashSet2;
- Observa-se uma melhoria de LHashSet2 para LHashSet3.

Com 4 threads:

- Observa-se um maior overhead em LHashSet1 do que em LHashSet0;
- Observa-se um maior overhead em LHashSet2 do que em LHashSet1 (excepto $N = 50000$);
- Observa-se uma melhoria de LHashSet1 e LHashSet2 para LHashSet3.

Para $N = 50000$, LHashSet 2 foi o melhor resultado, seguido de LHashSet0.

Com 8 threads:

- Observa-se um maior overhead em LHashSet1 do que em LHashSet0;
- Observa-se um maior overhead em LHashSet2 do que em LHashSet1 para $N = 256$, e uma melhoria de LHashSet1 para LHashSet2 com os N superiores;
- Observa-se uma melhoria de LHashSet1 e LHashSet2 para LHashSet3.

Conclusões

Foram assinalados a negrito nas tabelas os melhores Mops obtidos para cada N.

Para $N = 256$ o LHashSet0 com 1 thread revelou-se a solução mais eficiente. Isto pode ser explicado por o custo/overhead de context switching com um N tão pequeno nunca ser superado pelo potencial benefício de utilizar vários threads.

Para $N = 5000$ o LHashSet3 com 8 threads revelou-se a solução mais eficiente. Isto sugere que foi um bom compromisso entre elevado número de threads e overhead implicitos e uma quantidade de elementos razoável.

Para $N = 50000$ o LHashSet3 com 4 threads revelou-se a solução mais eficiente. Isto sugere que com esta elevada quantidade de elementos o overhead de manter 8 threads já não era mais sustentável. Sendo assim necessário reduzir os threads para elevar a eficiência.

Os resultados parecem-nos consistentes com as nossas expectativas. A STM não se mostrou como uma abordagem razoável na implementação deste tipo de estrutura de dados com a finalidade de tratar os N tratados. Todavia, revela uma grande escalabilidade pois consoante subimos os threads consistentemente aumentou ou manteve a eficiência.

2 Crawler concorrente

2.1 Avaliação de desempenho

Tabela 6: Média de resultados de *crawler_bench.sh* para ConcurrentCrawler (5 execuções) com Javadoc do Cooperari (354 ficheiros)

Threads	2	4	8	16
Tempo (ms)	2513.8	1955.8	1748.8	1618.2
Transferências/s	140.82	181	202.42	218.76

Tabela 7: Média de resultados de *crawler_bench.sh* para ConcurrentCrawler (5 execuções) com Javadoc do JDK8 (1789 ficheiros)

Threads	2	4	8	16
Tempo (ms)	37090.2	33108.8	29431.2	30476.2
Transferências/s	48.23	54.03	60.79	58.70

Condições do teste

- CPU: Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz
- OS: Linux 5.10.0-7-amd64, Debian 5.10.40-1 x86_64
- Executado num tty de texto.

2.1.1 Análise de resultados e Conclusões

Os resultados parecem-nos consistentes com as nossas expectativas. Em todos os testes o aumento de threads resultou consistentemente numa redução de tempo de processamento. Havendo a excepção de 16 threads com a Javadoc do JDK8, onde se tornou claro que o custo/overhead de context switching se determinava contra-producente face os ganhos, conforme discutido na análise da tabela de dispersão concorrente.