Universidade do Minho

Mestrado em Engenharia Informática

Engenharia dos Sistemas de Computação - 2014/2015

Portfolio

22 de Junho de 2015

Fábio Gomes pg27752

Índice

O ambiente de execução em clustering e Utilitários de Monitorização - Introdução

O problema que nos foi apresentado consiste em analisar a performance de um sistema de forma a poder saber as capacidades e limites operacionais para as aplicações que irão ser desenvolvidas para aquela infraestrutura.

Foi proposto que os testes incidissem em diferentes classes com diferentes compiladores, como o *icc* e *gcc* (versão 4.4.6 e 4.9.0), número de processos (*MPI*) e threads (*OpenMP*) e ainda para as versões SERIAL, OpenMP e MPI para cada Benchmark.

Os dados que o NPB devolve não são suficientes e para tal foi necessário recorrer a ferramentas externas para obter informações extras ao longo do tempo de execução. Especificamente o derivado *dstat* para a alocação de memória, acessos ao disco e dados enviados/recebidos na rede; o *mpstat* para a utilização dos diferentes processadores.

Com estes dados recolhidos foi possível gerar gráficos para comparar as diferentes combinações de compiladores e número de processos e threads.

NAS Parallel Benchmarks (NPB)

São um pequeno conjunto de programas destinados a ajudar a avaliar o desempenho dos supercomputadores paralelos. Os Benchmarks são derivadas de aplicações da dinâmica de fluido computacional (CFD) e consistem em cinco kernels e três pseudo-*aplicações* (NPB 1). O pacote de Benchmark foi estendido para incluir novos pontos de referência para malha adaptativa não-estruturada, I/O, aplicações multi-zona, e grelhas computacionais paralelas. Os tamanhos dos problemas no NPB são predefinidos e indicados com diferentes classes (A,B,C,D,E,F ou S,W). Implementações de referência de NPB estão disponíveis em modelos de programação mais usadas como MPI e OpenMP (NPB 2 e NPB 3), sendo estas as versões que mais nos interessam.

Há diferentes especificações de Benchmark:

- 5 kernels
  - `IS - Integer Sort, random memory access`
  - `EP - Embarrassingly Parallel`
  - `CG - Conjugate Gradient, irregular memory access and communication`
  - `MG - Multi-Grid on a sequence of meshes, long- and short-distance communication, memory intensive`
  - `FT - discrete 3D fast Fourier Transform, all-to-all communication`
- 3 pseudo-aplicações
  - `BT - Block Tri-diagonal solver`
  - `SP - Scalar Penta-diagonal solver`
  - `LU - Lower-Upper Gauss-Seidel solver`

E classes de tamanho:

- S: small for quick test purposes
- W: workstation size (a 90's workstation; now likely too small)
- A, B, C: standard test problems; ~4X size increase going from one class to the next
- D, E, F: large test problems; ~16X size increase from each of the previous classes

Optei pela *EP*, *FT*, *IS* e *SP* como foco para os testes com classes A e B em cada.

Caracterização do Sistema

Para utilização deste projeto utilizei o nó 431-6 do SeARCH, recorrendo a jobs específicos para ele, com a seguinte especificação:

	Cluster Node
Manufacturer	intel
Model	X5650
Clock speed	2.67 GHz
Architecture	x86-64
#Cores	12
#Threads per Core	2
Total Threads	24

Compilação, Versões e Parâmetros

Para este trabalho tive que utilizar diferentes versões de compiladores e ferramentas, todas com -O.

gcc / mpicc / gfortran

4.4.6 e 4.9.0

Icc / ifort

13.0.1

Para cada kernel de Benchmark foi necessário usar estas 3 versões:

Serial

Versão base em que tipicamente é utilizado um processador.

OpenMP

Utiliza um número especificado de threads, neste caso 2, 8, 16 e 48.

MPI

Com o número providenciado de processos, 2, 4, 8 e 16 (no caso do SP 4, 9 e 16).

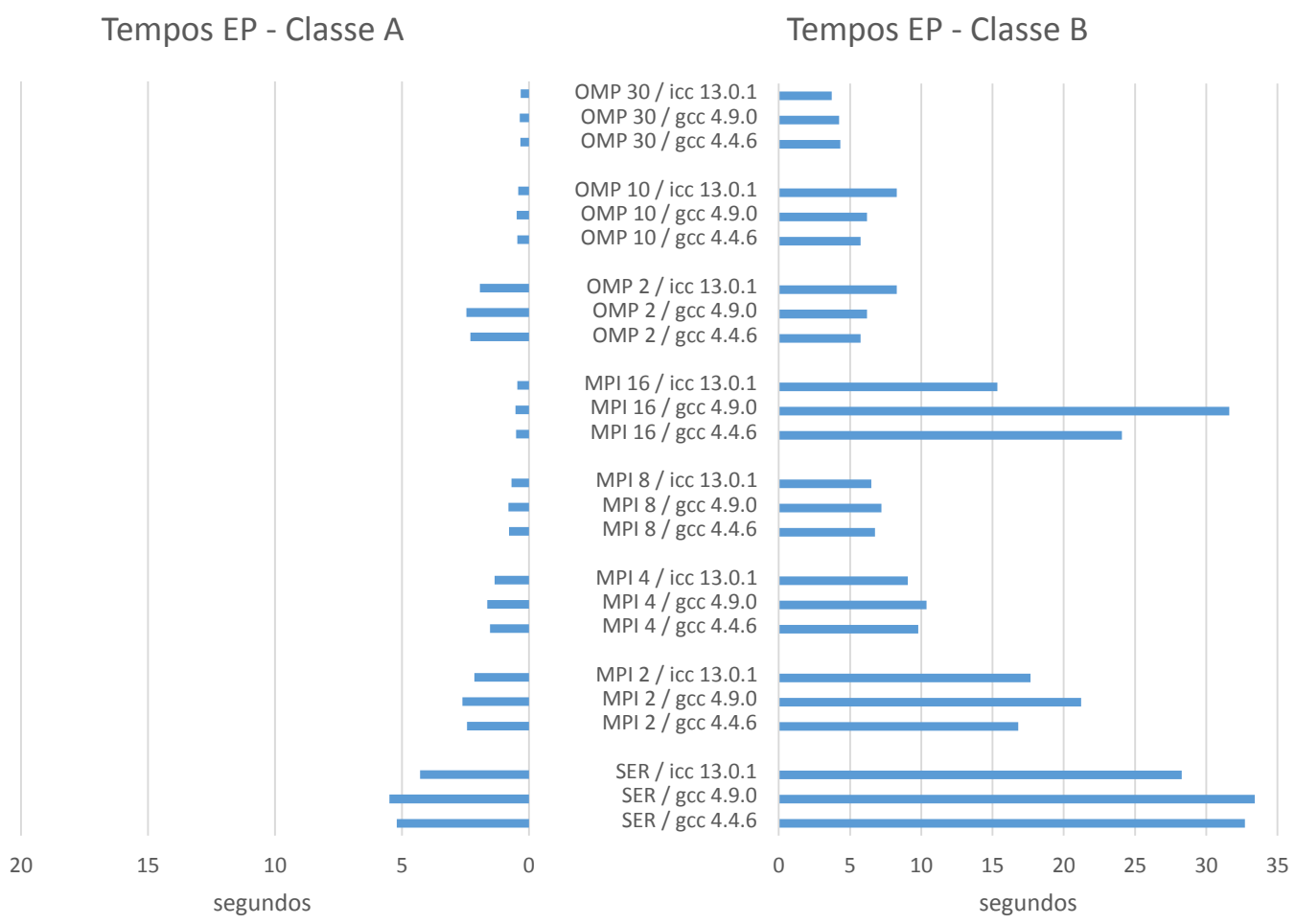
Benchmark EP

Há 2 classes, A e B, que estão representadas. A verde são marcados os tempos melhores entre o mesmo teste comparando com as 3 versões de compiladores e a vermelho o pior.

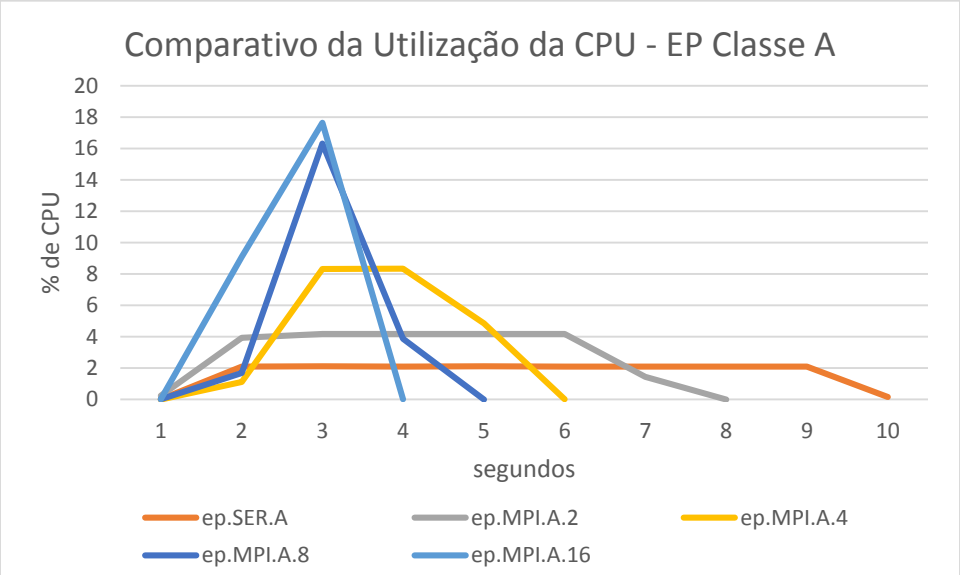
Análise da Duração

A seguinte tabela inclui os tempos de execução e os *Mop/s* (Milhões de Operações por segundo) para cada combinação de Teste.

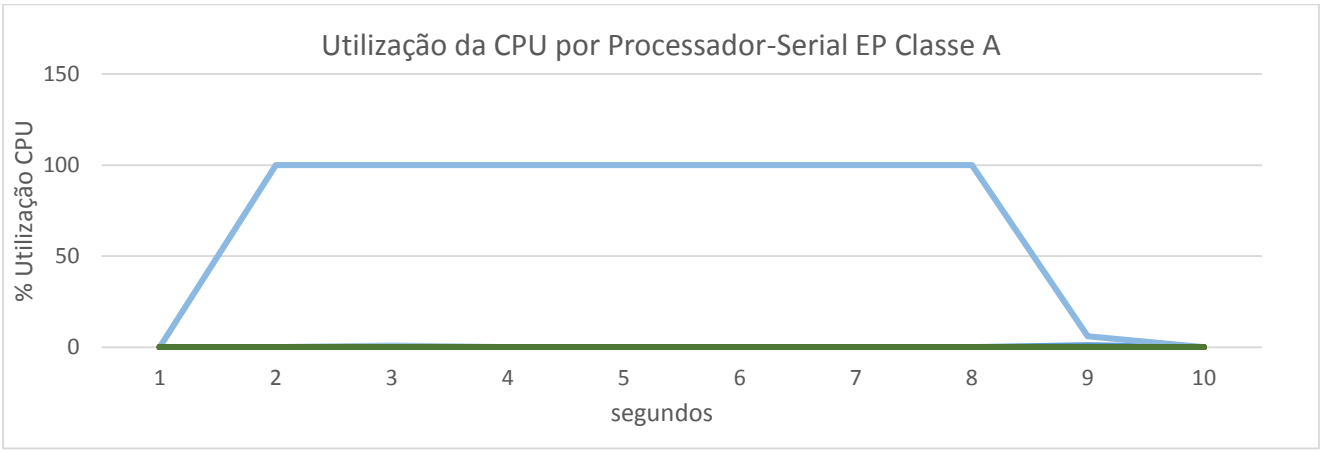
		EP															
		SER		MPI								OpenMP					
				2 Procs		4 Procs		8 Procs		16 Procs		2 Threads		10 Threads		30 Threads	
		s	Mop/s	s	Mop/s	s	Mop/s	s	Mop/s	s	Mop/s	s	Mop/s	s	Mop/s	s	Mop/s
gcc 4.4.6	**A**	19,4	27,7	8,59	62,48	5,36	100,2	2,27	236,6	1,15	467	10	53,45	1,82	194	0,78	690
gcc 4.9.0		20,3	26,5	8,88	60,4	4,47	120,1	2,44	220,25	1,21	442	9,84	54,57	2,01	266	0,81	659
icc 13.0.1		9,36	57,4	4,14	129,8	2,56	209,6	1,19	452,81	0,64	835	4,33	124,1	0,78	687	0,37	1461
gcc 4.4.6	**B**	77,2	27,8	34,3	62,55	17,3	124,5	9,29	231,24	4,59	468	31,8	67,55	7,04	305	2,95	728
gcc 4.9.0		79,3	27,1	34,9	61,46	22,5	95,26	9,8	219,19	5,01	428	35	61,39	6,87	312	3,55	604
icc 13.0.1		37,4	57,4	15,9	135,4	10,2	209,6	5,23	410,62	2,2	975	15,9	135,4	3,05	703	1,41	1520



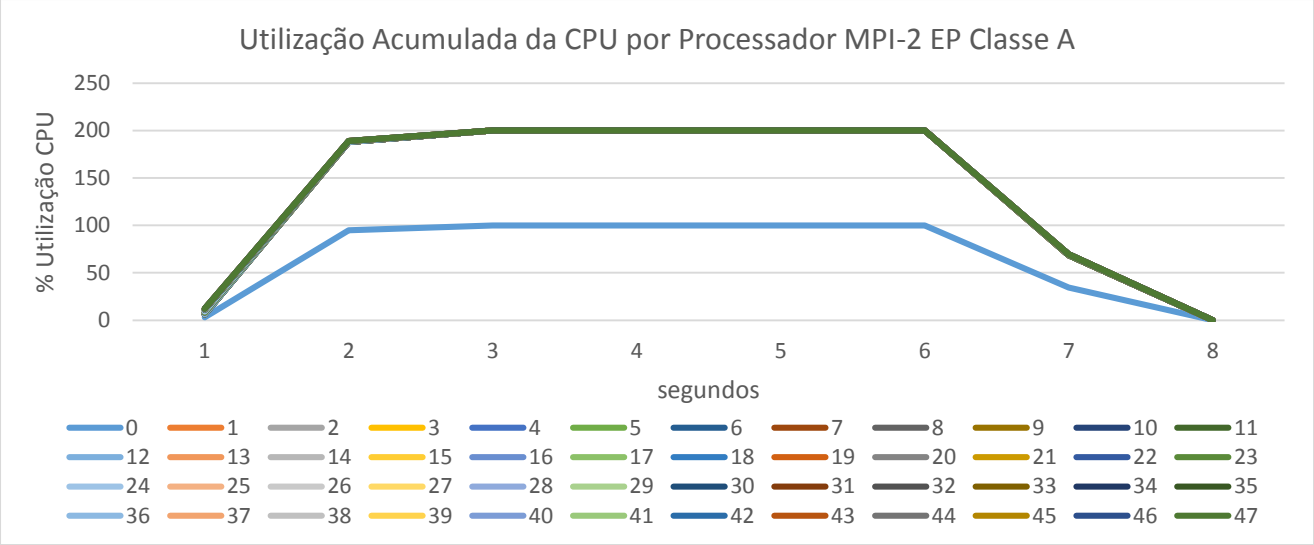
É possível distinguir que o `icc` foi o melhor compilador porque obteve 100 % dos melhores tempos. Sendo assim foi utilizado o `icc` como base dos testes seguintes.



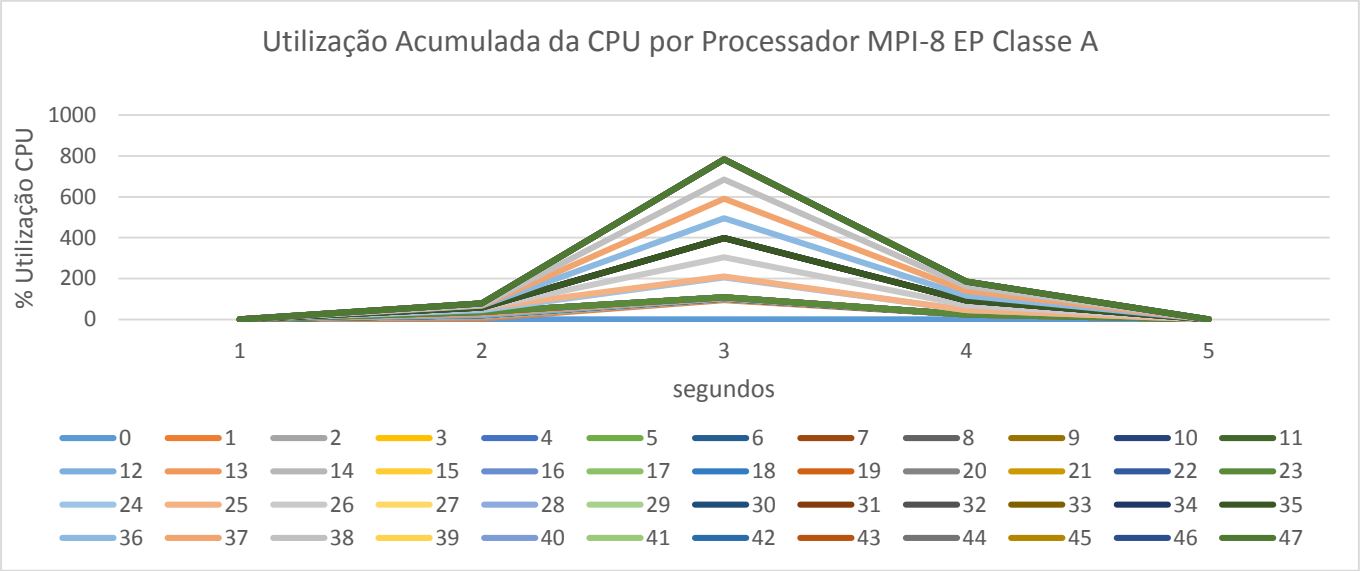
Este gráfico comparativo em que temos os 5 testes para o EP com a Classe A, não é muito interessante pois fica apenas claro quais são as combinações mais rápidas. Assim utilizando a soma acumulada, ou apenas individual, das percentagens de utilização por processador dá uma melhor perceção dos resultados. Visto o `mpstat` ter monitorizado durante os jobs 48 processadores o máximo de performance será `48*100% = 4800%`. Os gráficos seguintes são alguns exemplos.



Como a versão Serial só utiliza um processador a utilização individual será de 100.



Com 2 processos, a utilização acumulada é de 200 % como previsto.

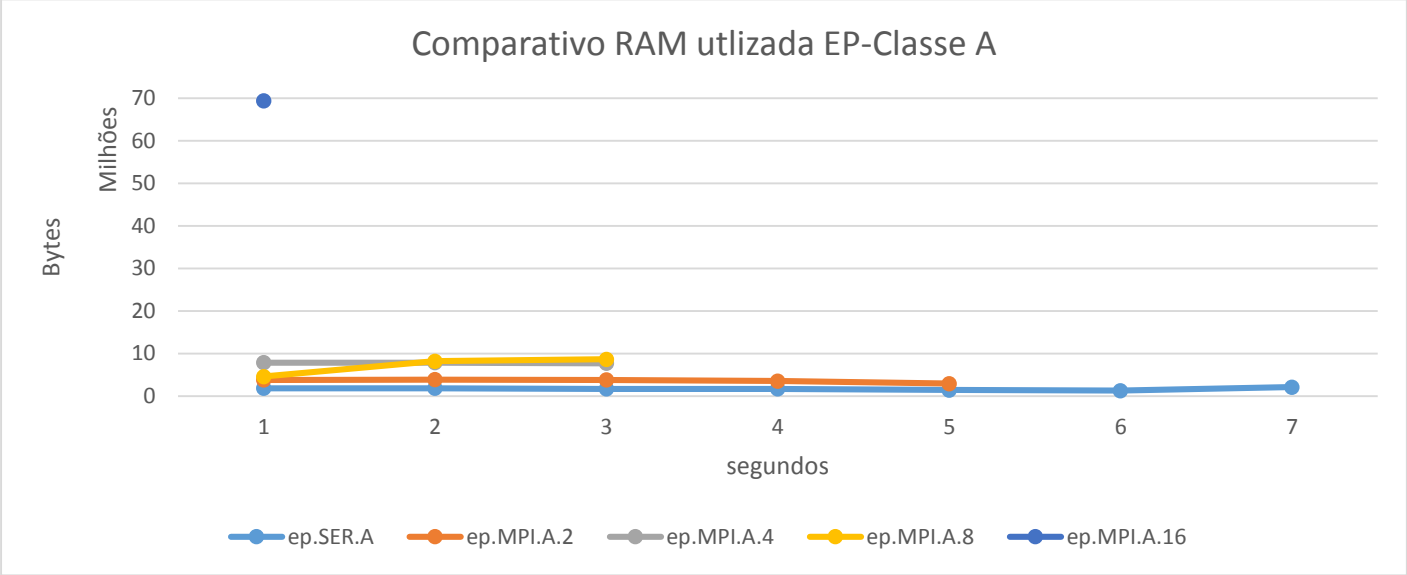


Com 8 processos a utilização bate nos 800 %.

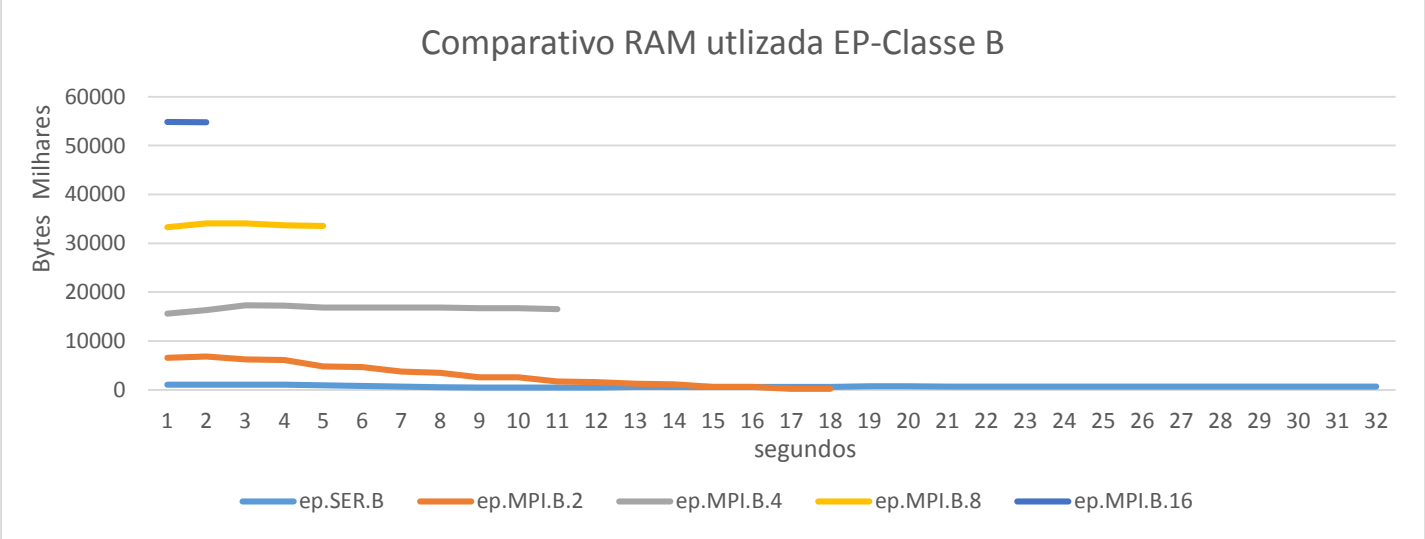
Análise da Memória

Os gráficos seguintes demonstram a alocação de Memória ao longo do tempo de execução.

Classe A:



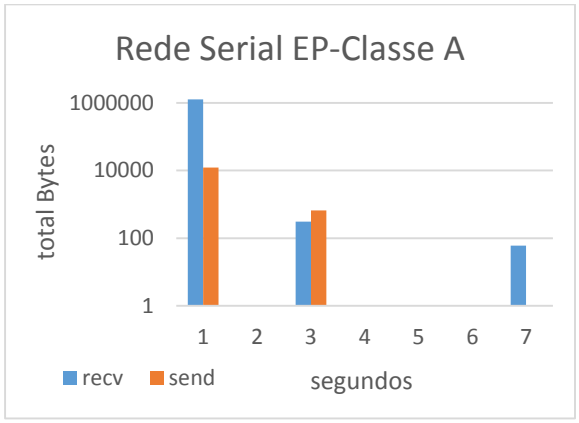
Classe B:

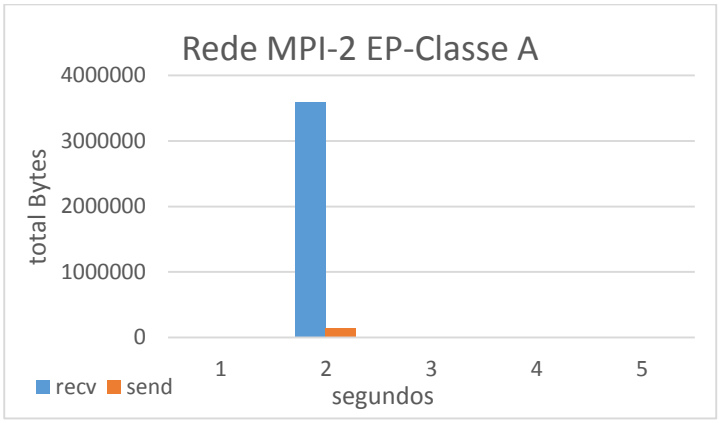


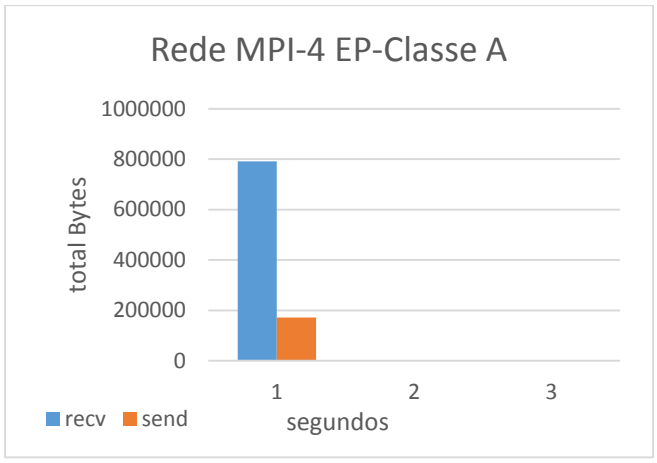
Como expectável a memória necessária para o funcionamento dos programas aumenta com o número de paralelismo. Cabendo a nós, programadores, decidir qual o melhor balanço entre duração e memória disponível.

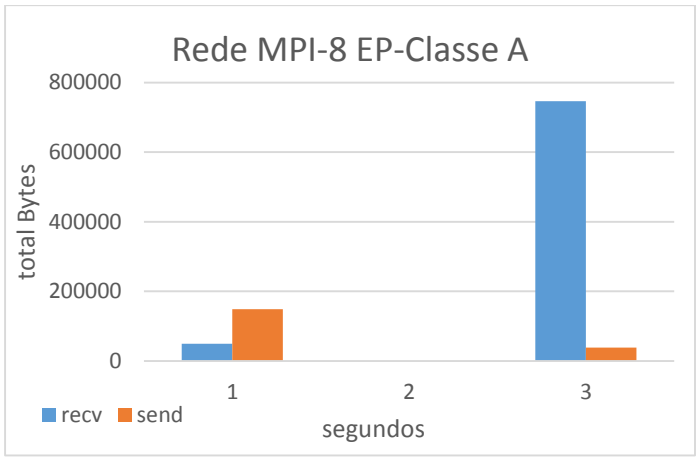
Análise da Rede

Os gráficos são muito diferentes e não é fácil fazer uma comparação entre eles pois também os valores ora são na ordem das centenas de milhar como passam para as centenas como estão a zero. De notar que o gráfico para o Serial da Classe A está com eixo dos Bytes com escala logarítmica de base 10.









Análise de Acessos ao Disco

Os testes que utilizei não são focados na interação com o Disco por isso não faz sentido realizar esta medição para qualquer dos testes seguintes.

Conclusão

O `icc` neste kernel foi o que obteve melhores resultados.

Benchmark FT

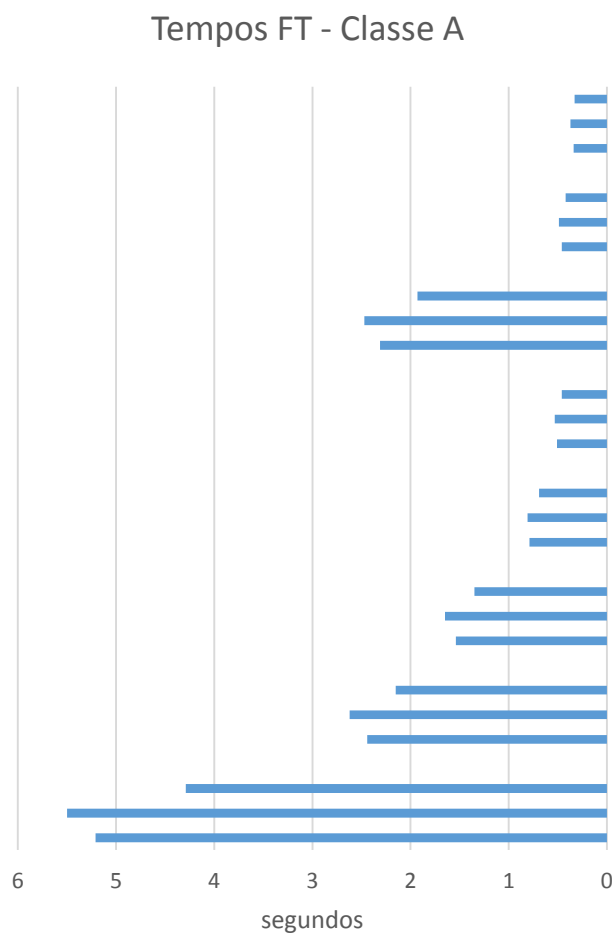
Há 2 classes, A e B, que estão representadas. A verde são marcados os tempos melhores entre o mesmo teste comparando com as 3 versões de compiladores e a vermelho o pior.

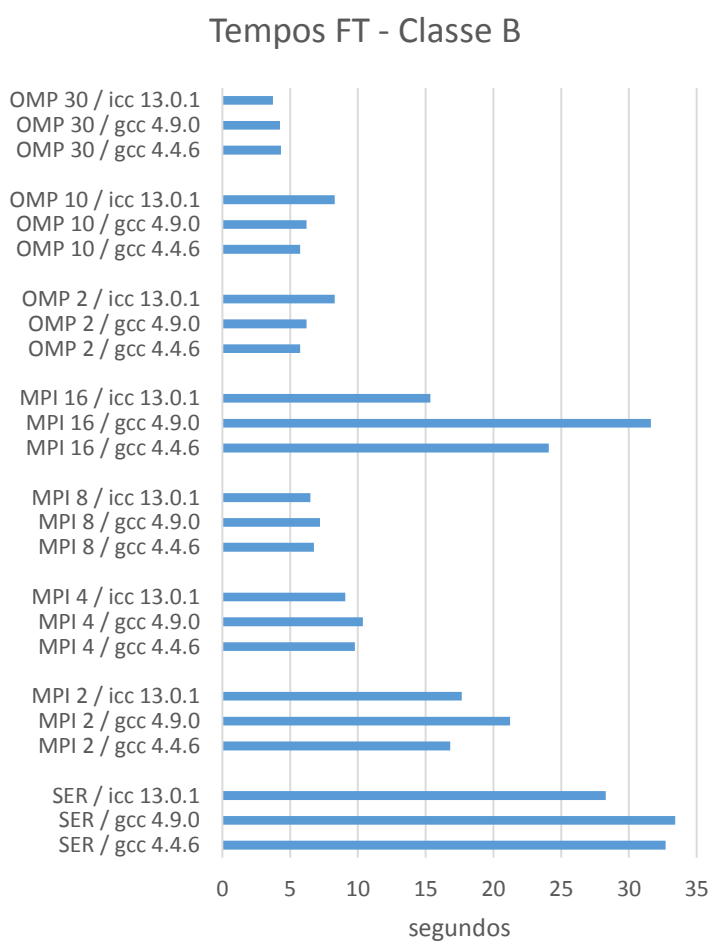
Análise da Duração

A seguinte tabela inclui os tempos de execução e os *Mop/s* (Milhões de Operações por segundo) para cada combinação de Teste.

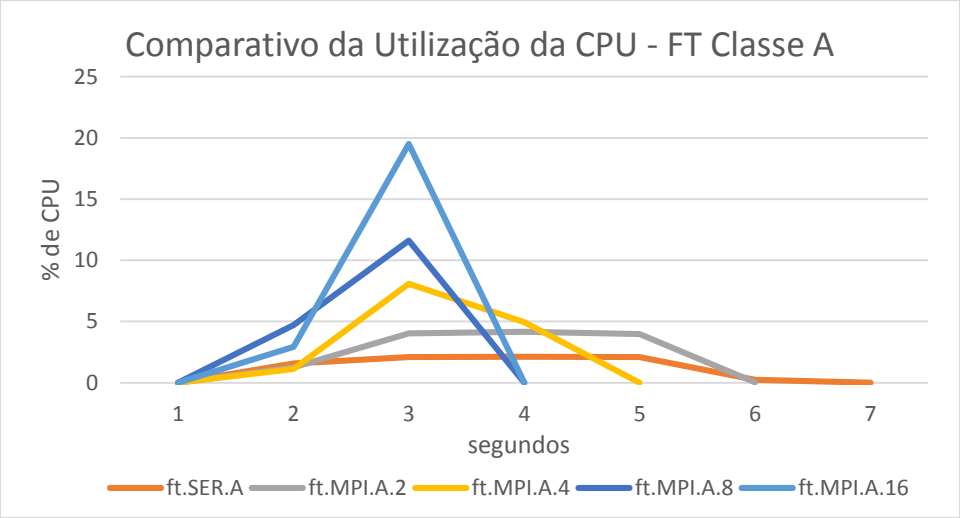
.

		FT															
		SER		MPI								OpenMP					
				2 Procs		4 Procs		8 Procs		16 Procs		2 Threads		10 Threads		30 Threads	
		s	Mop/s	s	Mop/s	s	Mop/s	s	Mop/s	s	Mop/s	s	Mop/s	s	Mop/s	s	Mop/s
gcc 4.4.6	**A**	5,21	1369	2,44	2920	1,54	4619	0,79	9019	0,51	14009	2,31	3086	0,46	15597	0,34	20806
gcc 4.9.0		5,5	1297	2,62	2724	1,65	4320	0,81	8819	0,53	13550	2,47	2888	0,49	14462	0,37	19245
icc 13.0.1		4,29	1663	2,15	3315	1,35	5301	0,69	10403	0,46	15378	1,93	3700	0,42	17144	0,33	21332
gcc 4.4.6	**B**	69,5	1324	32,7	2814	16,8	5476	9,78	9414	6,76	13609	24,1	3821	5,74	16042	4,33	21247
gcc 4.9.0		65,3	1410	33,4	2756	21,2	4337	10,4	8876	7,21	12769	31,6	2911	6,2	14851	4,24	21709
icc 13.0.1		60,3	1528	28,3	3254	17,7	5209	9,06	10161	6,5	14167	15,3	139	8,29	11109	3,72	24746

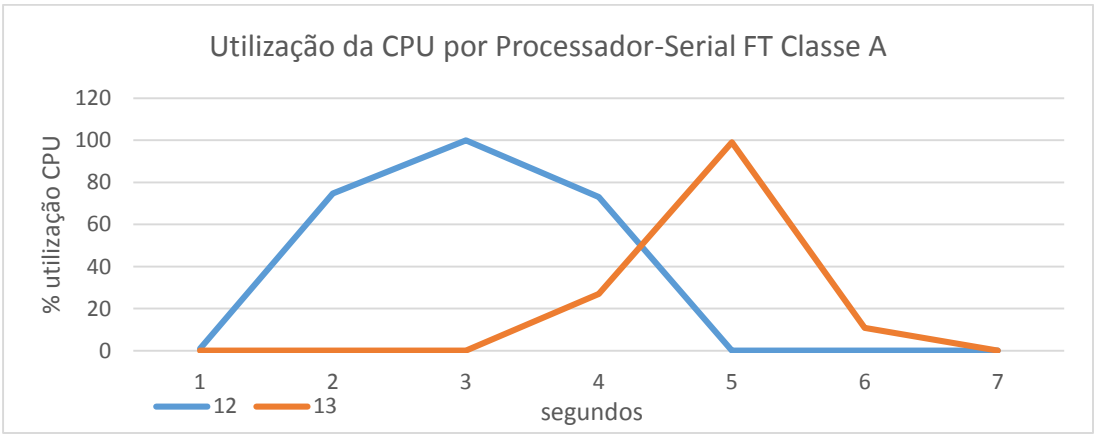




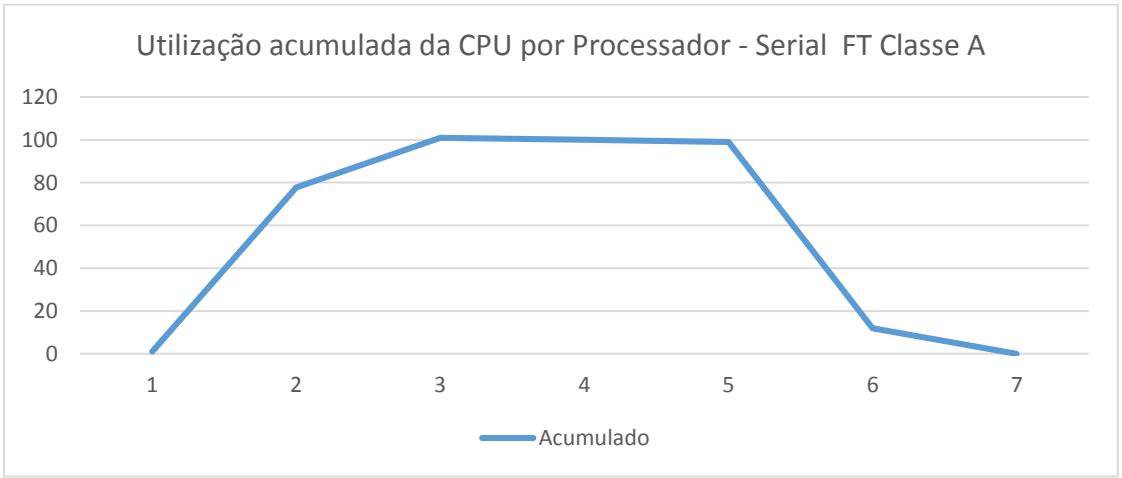
O compilador `icc` obteve uma maioria de melhores tempos, não sendo o melhor em apenas 2 ocasiões.



Este gráfico comparativo anterior apresenta os 5 testes para o FT com a Classe A, sendo possível reparar quais são as combinações mais rápidas. De seguida são apresentados os resultados em gráfico da soma acumulada, ou apenas individual, das percentagens de utilização por processador.



Aconteceu uma situação interessante, o processo trocou de processador durante a execução.

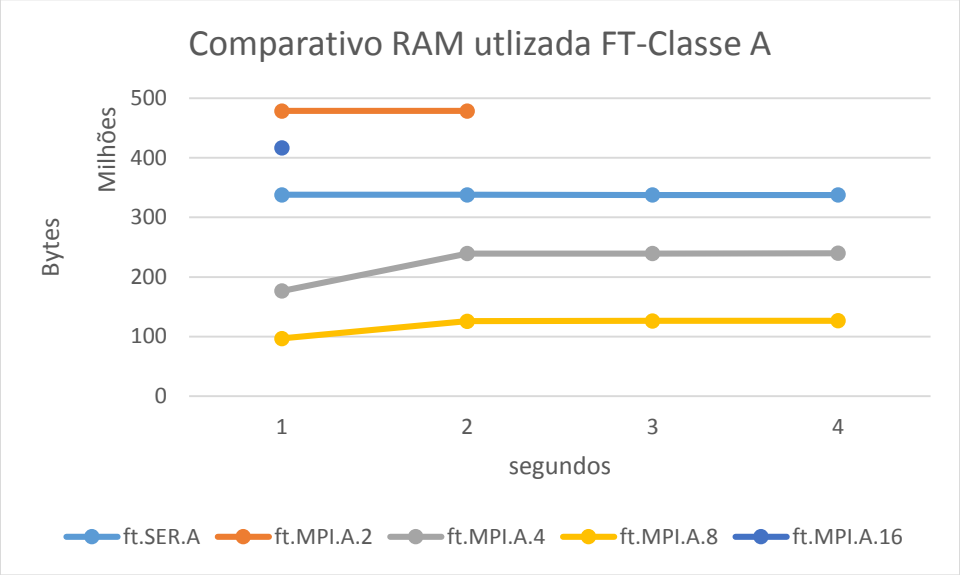


Mas o acumulado permaneceu no máximo dos 100 %.

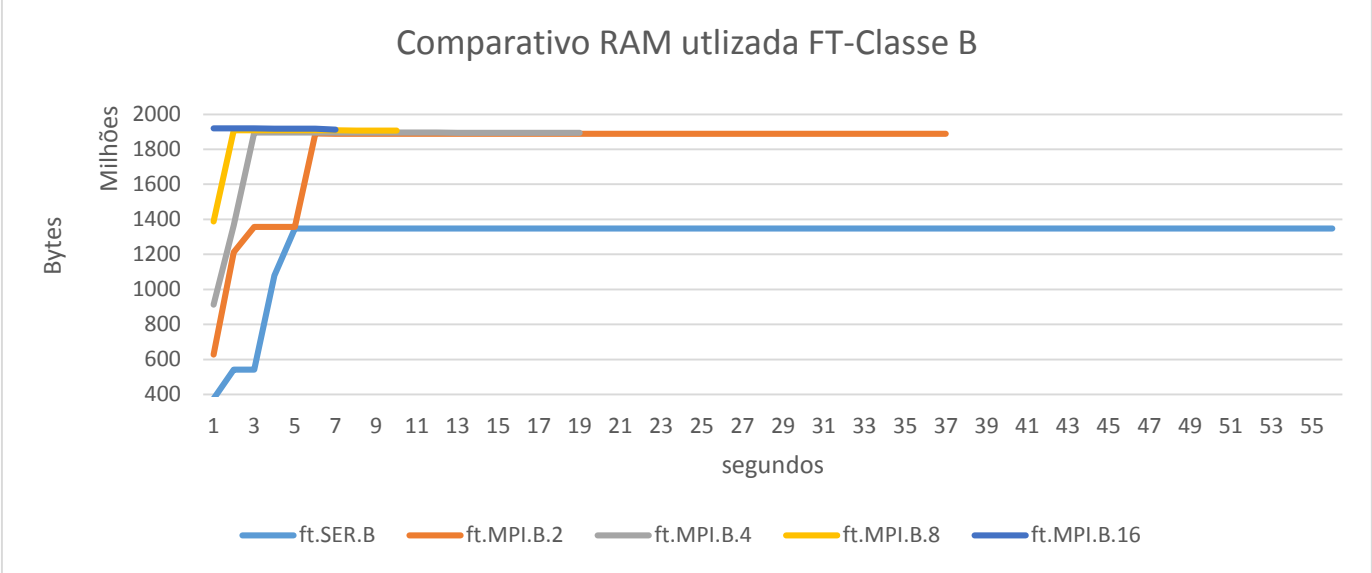
Análise da Memória

Os gráficos seguintes demonstram a alocação de Memória ao longo do tempo de execução.

Classe A:



Classe B:



A memória usando MPI com a Classe B ficou estável para os testes na ordem dos 1800 MBytes.

Conclusão

O `icc` neste kernel foi, de novo, o que obteve melhores resultados.

Benchmark IS

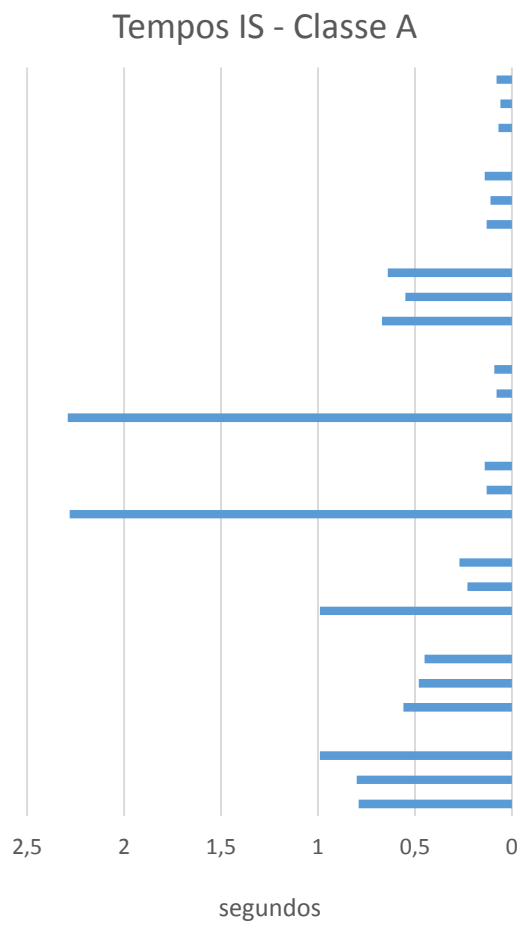
Há 2 classes, A e B, que estão representadas. A verde são marcados os tempos melhores entre o mesmo teste comparando com as 3 versões de compiladores e a vermelho o pior. Este é o teste mais curto.

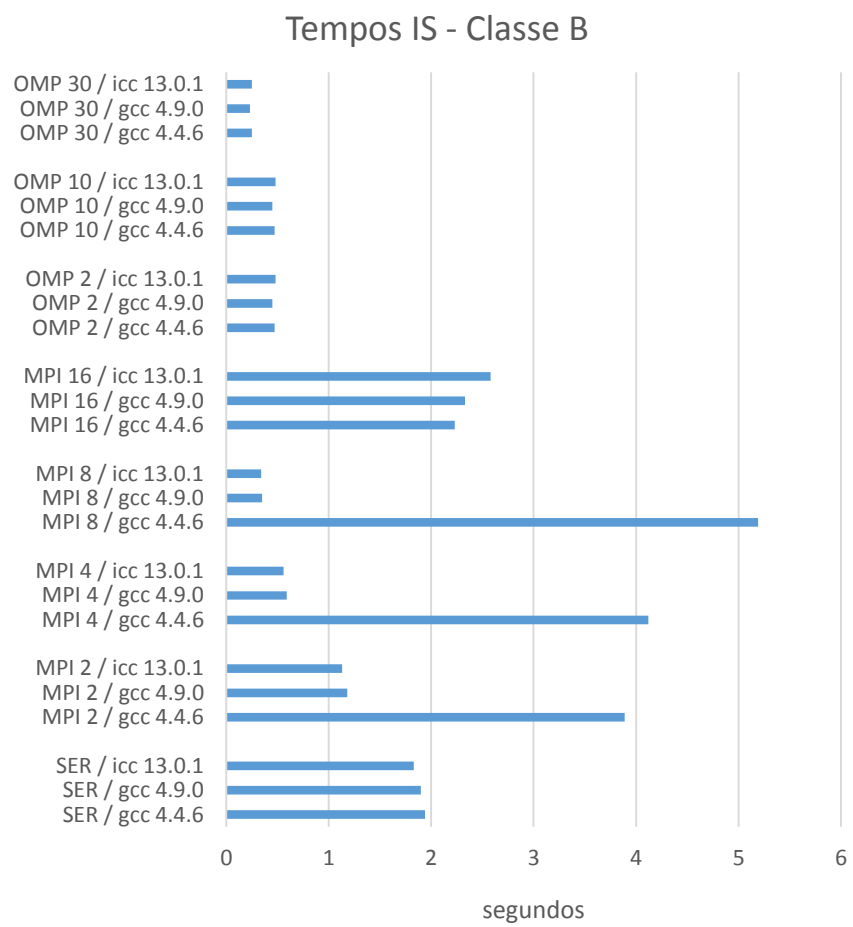
Análise da Duração

A seguinte tabela inclui os tempos de execução e os *Mop/s* (Milhões de Operações por segundo) para cada combinação de Teste.

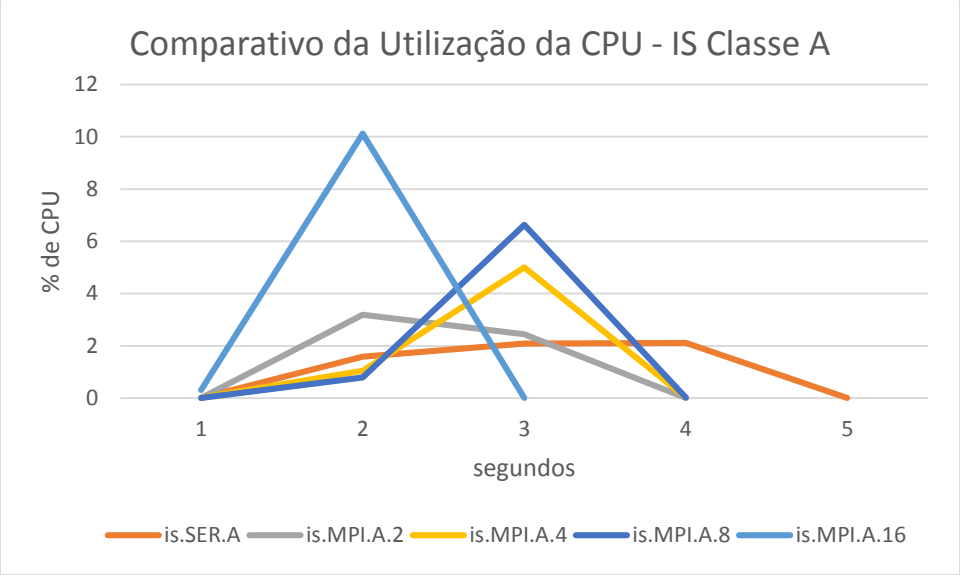
.

		IS															
		SER		MPI								OpenMP					
				2 Procs		4 Procs		8 Procs		16 Procs		2 Threads		10 Threads		30 Threads	
		s	Mop/s	s	Mop/s	s	Mop/s	s	Mop/s	s	Mop/s	s	Mop/s	s	Mop/s	s	Mop/s
gcc 4.4.6	**A**	0,79	107	0,56	148,8	0,99	84,32	2,28	37	2,29	37	0,67	125,9	0,13	660	0,07	1179
gcc 4.9.0		0,8	104	0,48	174,3	0,23	358,9	0,13	648	0,08	1078	0,55	151	0,11	770	0,06	1361
icc 13.0.1		0,99	84,3	0,45	185	0,27	315,5	0,14	590	0,09	998	0,64	131,7	0,14	616	0,08	1111
gcc 4.4.6	**B**	3,3	102	1,94	172,8	3,89	86,21	4,12	81	5,19	65	2,23	150,4	0,47	707	0,25	1341
gcc 4.9.0		3,29	102	1,9	177	1,18	284,1	0,59	570	0,35	955	2,33	143	0,45	743	0,23	1437
icc 13.0.1		4,23	79,4	1,83	183,6	1,13	296	0,56	595	0,34	995	2,58	130	0,48	705	0,25	1329

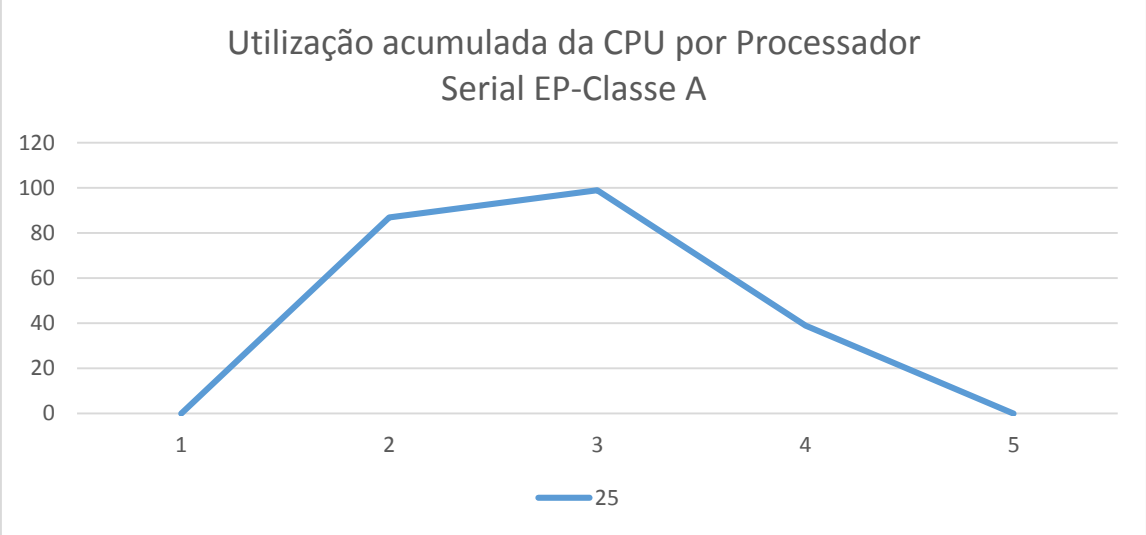




Neste teste o `gcc 4.9.0` foi o que obteve no geral os melhores resultados, batendo o `icc`.



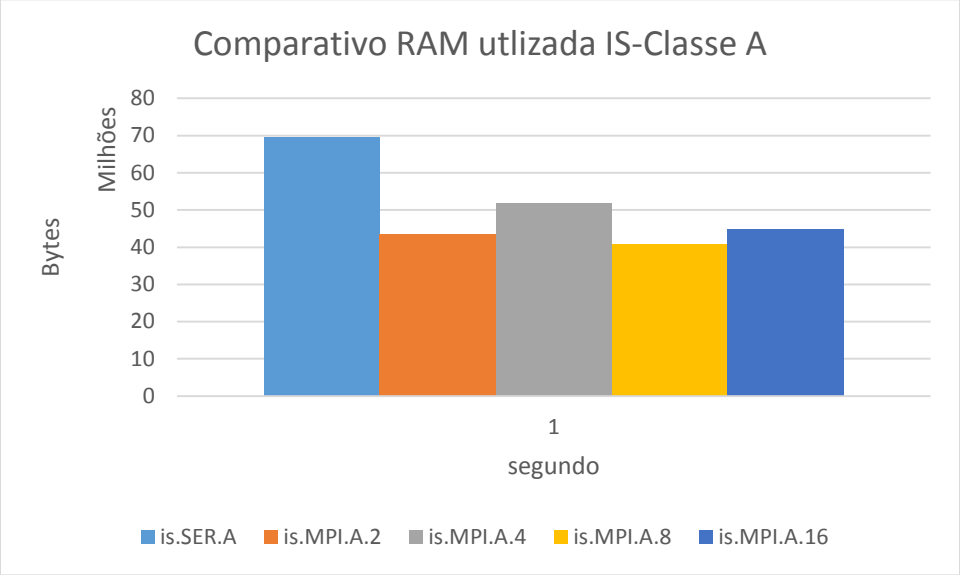
Este gráfico comparativo anterior apresenta os 5 testes para o IS com a Classe A, sendo possível reparar quais são as combinações mais rápidas.



Análise da Memória

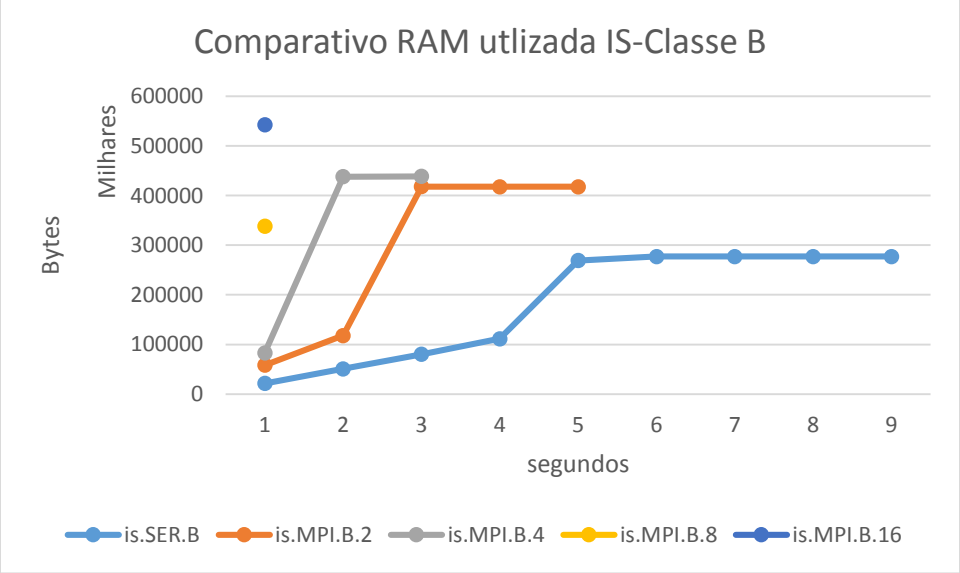
Os gráficos seguintes demonstram a alocação de Memória ao longo do tempo de execução.

Classe A:



Como estes testes normalmente nem 1 segundo duram, realizei vários seguidos e depois uma média.

Classe B:



Esta Classe já proporciona tempos maiores podendo assim gerar um gráfico temporal. Curiosamente com o aumento dos processos a memória utilizada diminuiu.

Conclusão

O `gcc 4.9.0` neste kernel foi o que obteve melhores resultados.

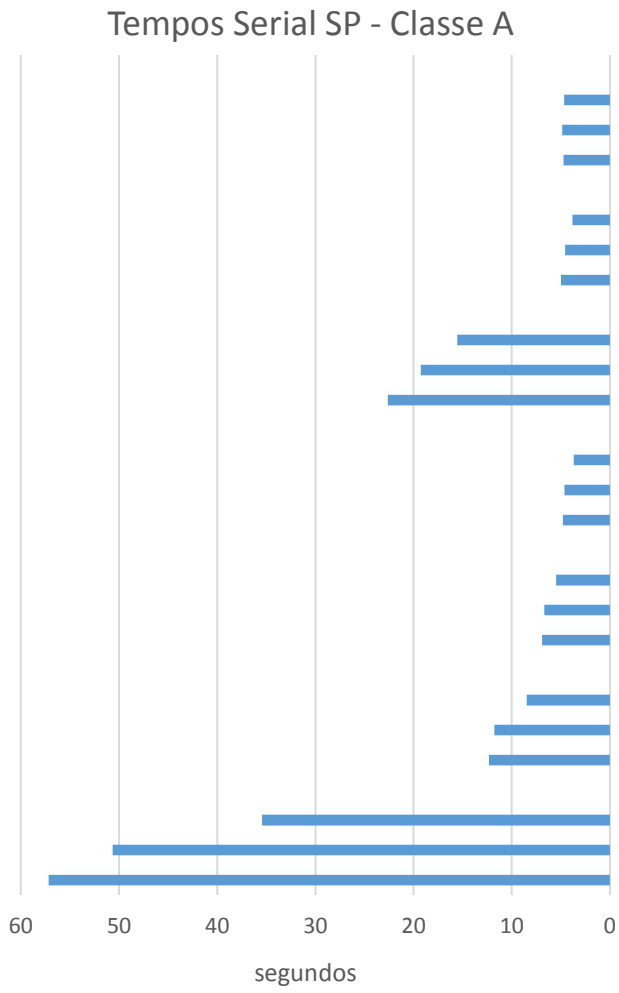
Benchmark SP

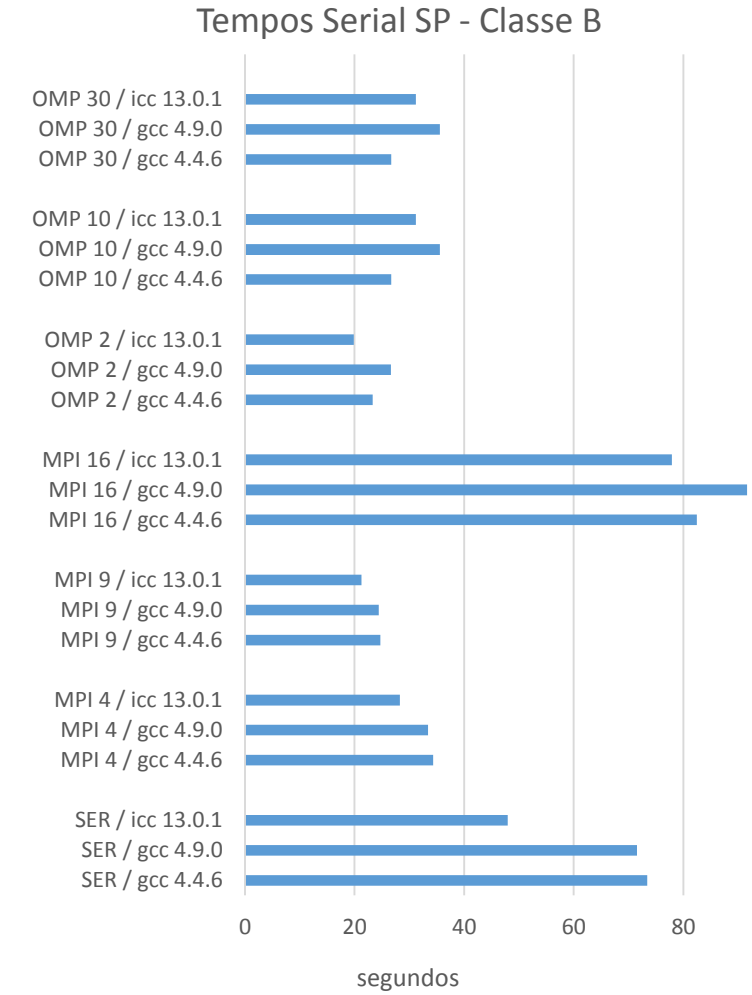
Há 2 classes, A e B, que estão representadas. A verde são marcados os tempos melhores entre o mesmo teste comparando com as 3 versões de compiladores e a vermelho o pior.

Análise da Duração

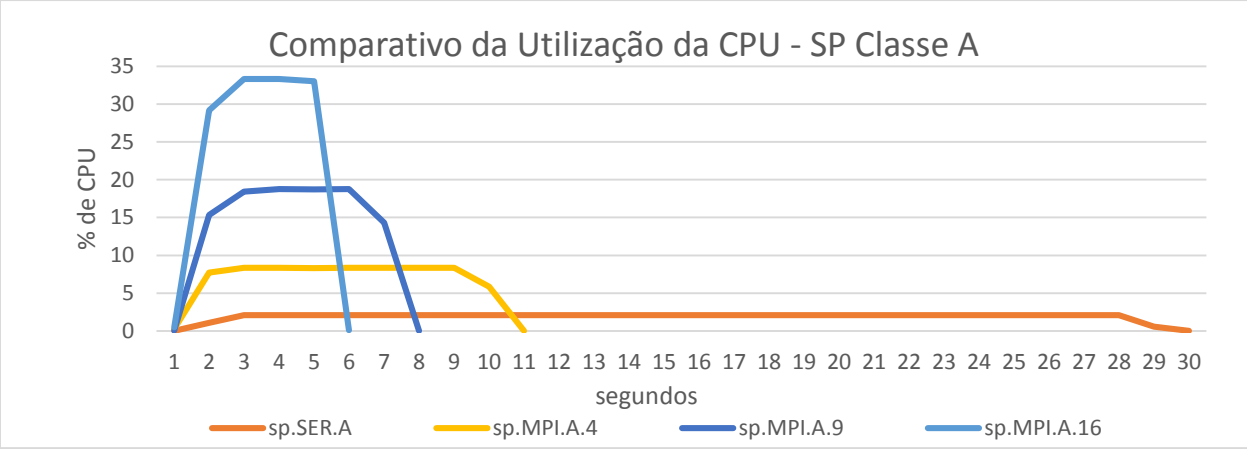
A seguinte tabela inclui os tempos de execução e os *Mop/s* (Milhões de Operações por segundo) para cada combinação de Teste.

		sp													
		SER		MPI						OpenMP					
				4 Procs		9 Procs		16 Procs		2 Threads		10 Threads		30 Threads	
		s	Mop/s	s	Mop/s	s	Mop/s	s	Mop/s	s	Mop/s	s	Mop/s	s	Mop/s
gcc 4.4.6	A	57,2	1486	12,3	6908	6,9	12313	4,78	17800	22,6	3756	4,99	17024	4,74	17918
gcc 4.9.0		50,7	1678	11,8	7223	6,69	12699	4,64	18333	19,3	4415	4,55	18678	4,85	17543
icc 13.0.1		35,5	2398	8,48	10022	5,48	1500	3,67	23179	15,5	5472	3,81	22289	4,66	18230
gcc 4.4.6	B	232	1532	73,4	4836	34,4	10330	24,7	14363	82,5	4304	23,3	15229	26,7	13299
gcc 4.9.0		232	1529	71,5	4963	33,4	10633	24,4	1456	91,7	3872	26,6	13325	35,6	9986
icc 13.0.1		162	2195	47,9	7408	28,3	12566	21,3	16703	78	4554	19,9	17862	31,2	11373

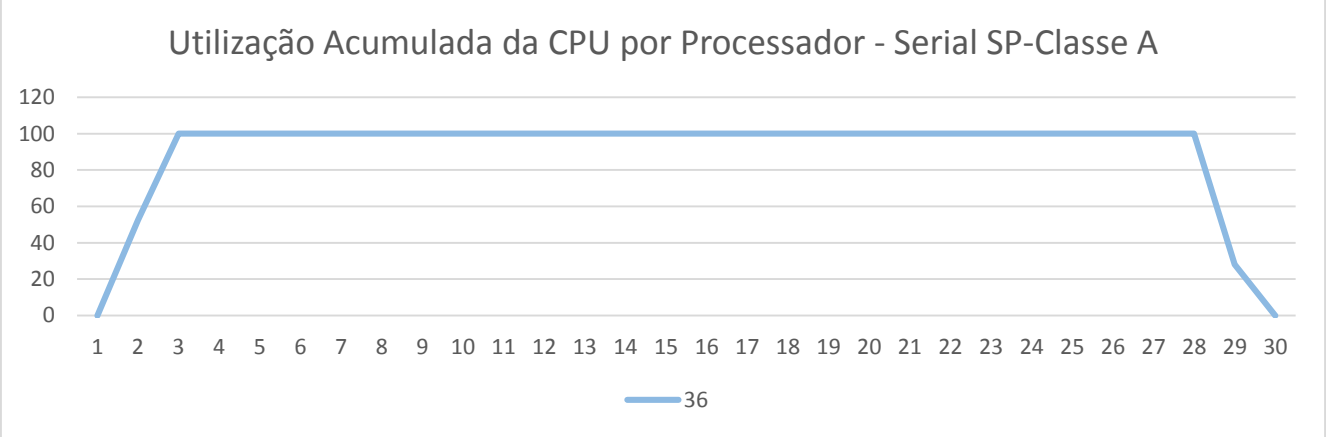




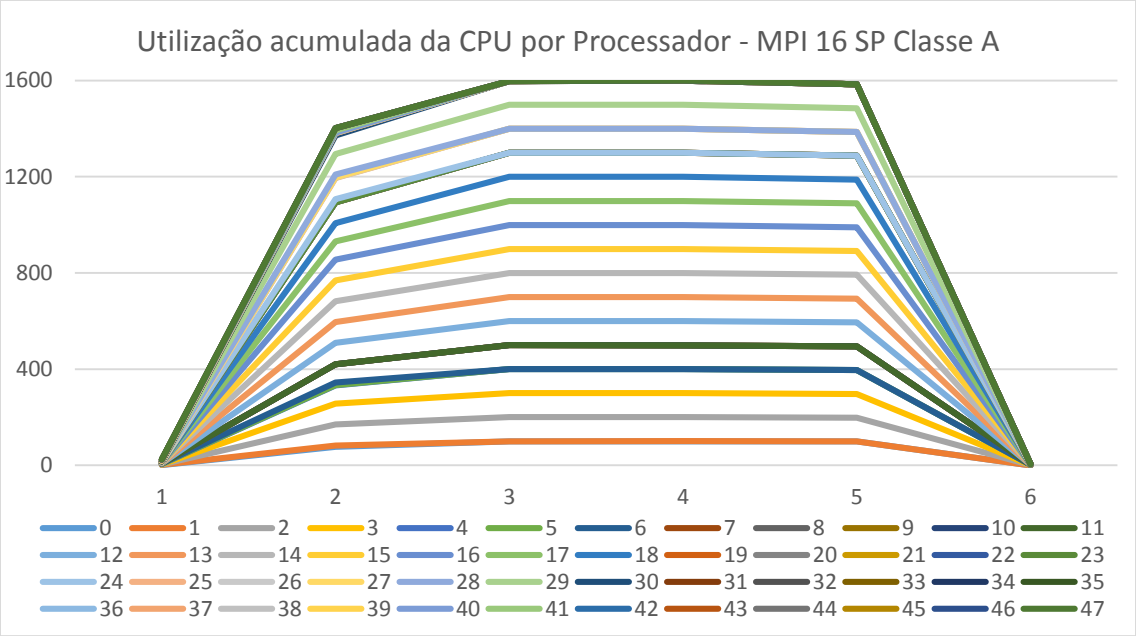
O `icc` para esta sequência de testes foi o compilador que obteve melhores resultados, falhando apenas um.



Com o aumento dos processos a utilização geral da CPU aumenta provocando tempos melhores.



O Processador 36 tomou conta do processo e a sua utilização foi de 100 %.

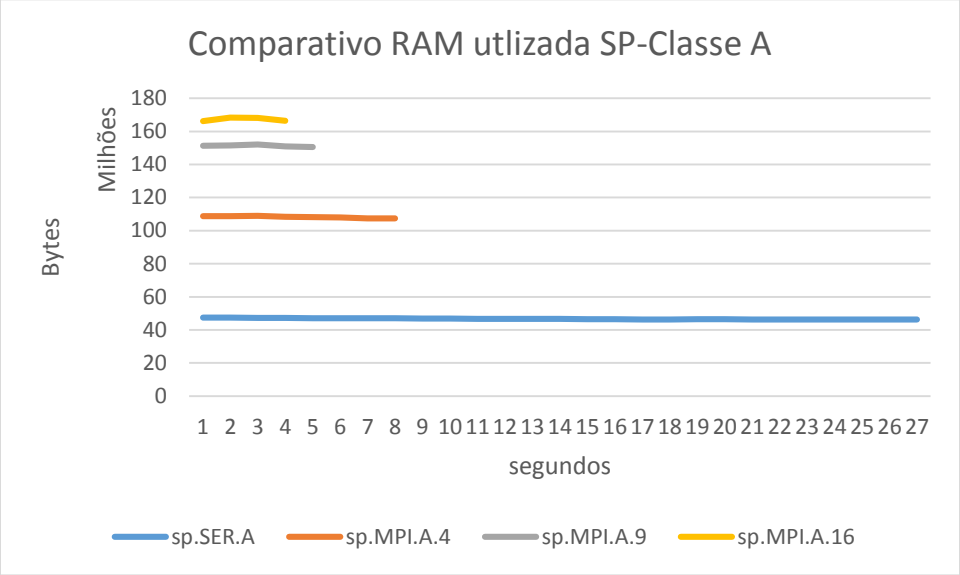


Com 16 processos para o MPI a soma de todas as contribuições chega ao previsto de 1600 %.

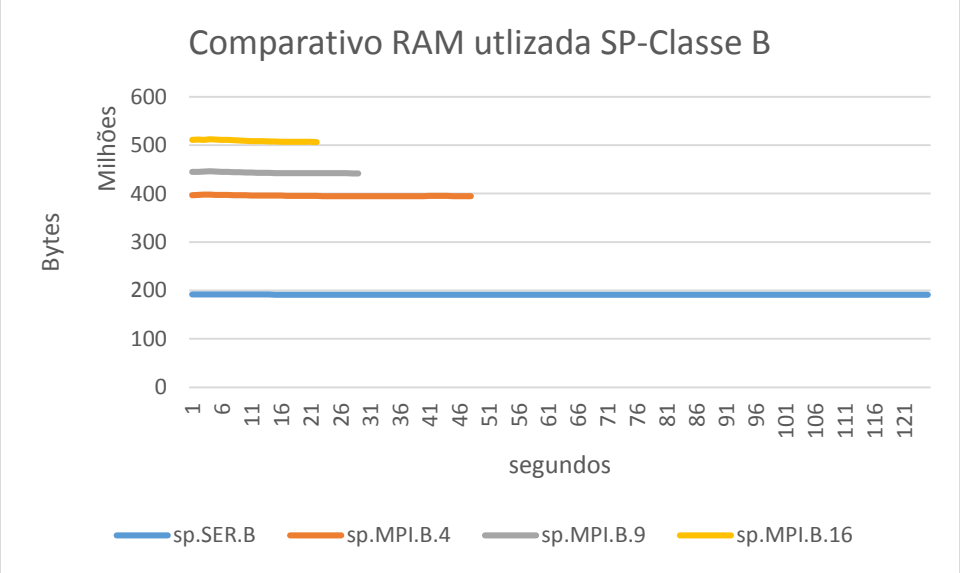
Análise da Memória

Os gráficos seguintes demonstram a alocação de Memória ao longo do tempo de execução.

Classe A:



Classe B:



Com o aumento de Processos o consumo de memória inevitavelmente também aumentou.

Conclusão

O `icc` neste kernel obteve os melhores resultados.

Scripts, Diretorias e Gráficos

Estes testes tiveram que ser realizados sucessivamente e sem a ajuda de scripts o trabalho seria muito penoso. Criei a seguinte estrutura de pastas para facilitar a organização:

```
.
|-- gcc
|   |-- 4.4.6
|   |   |-- MPI
|   |   |   |-- A
|   |   |   `-- B
|   |   |-- OMP
|   |   |   |-- A
|   |   |   `-- B
|   |   `-- SER
|   |       |-- A
|   |       `-- B
|   `-- 4.9.0
|       |-- MPI
|       |   |-- A
|       |   `-- B
|       |-- OMP
|       |   |-- A
|       |   `-- B
|       `-- SER
|           |-- A
|           `-- B
`-- icc
    |-- MPI
    |   |-- A
    |   `-- B
    |-- OMP
    |   |-- A
    |   `-- B
    `-- SER
        |-- A
        `-- B
```

Nas pastas A e B estão os executáveis respetivos.

De seguida fica um excerto do script que utilizei para fazer make aos programas e depois copiar para as pastas devidas, mostradas acima. Neste caso para MPI com `gcc` com a variável `v` a ser a versão (4.4.6 ou 4.9.0). Há o `make bundle` do NPB mas assim tenho a certeza e maior controlo do que acontece.

```
for th in 2 4 8 16
do
make is CLASS=A "NPROCS=$th"
make ep CLASS=A "NPROCS=$th"
make ft CLASS=A "NPROCS=$th"

make is CLASS=B "NPROCS=$th"
make ep CLASS=B "NPROCS=$th"
make ft CLASS=B "NPROCS=$th"
done

cp bin/*.A.* "../tests/gcc/$v/MPI/A/"
cp bin/*.B.* "../tests/gcc/$v/MPI/B/"
```

Para a medição de estatísticas foi necessário utilizar o `dstat` em background e o `mpstat` depois a colecionar os dados de cpu.

```
dstat -cdmrn --integer --output OMP_dstat.txt > /dev/null &
mpstat -P ALL 1 >> mps_OMP.txt
```

Com as medições iniciadas já se pode começar a chamar os programas. Incluí `sleep` para depois ser mais fácil analisar as medições pois estão espaçadas e em *idle*.

```
exec > "t_OMP.txt"
for th in 2 10 20 30
do
export OMP_NUM_THREADS=$th
echo  $th " ./ep.A.x"
./ep.A.x
sleep 3
echo  $th " ./ft.A.x"
./ft.A.x
sleep 3
echo  $th " ./is.A.x"
./is.A.x
sleep 3
echo  $th " ./sp.A.x"
./sp.A.x
sleep 3
done
sleep 6
```

Análise Final de Resultados e Conclusão

O trabalho necessário para realizar este projeto foi imenso, sendo talvez o que mais tempo precisou incluindo os de PCP do semestre passado. Tivemos que aplicar conhecimentos de Linux para mais rapidamente o realizarmos. Não consegui realizar tudo o que me propus, como realizar o teste em várias máquinas para fazer o comparar e dar sentido ao Benchmark e utilizar diferentes otimizações.

Os dados que deram origem a todo o trabalho estão divididos em 5 ficheiros *xlsx*: *grafico.xlsx* contém o resumo dos tempos obtidos, *A_cpu.xlsx* com a utilização por processador e *A_io.xlsx* com os dados de memória, disco e rede; e os respetivos para B, *B_cpu.xlsx* e *B_io.xlsx*. Devem ser consultados pois contêm dados individuais para cada teste. Apenas alguns deles foram colocados no Relatório.

Sempre utilizei o `top` para uma rápida análise da utilização dos processos no sistema mas com este projeto descobri novas ferramentas que serão muito úteis no futuro e que dão mais informações.

Com tudo concluído, é tempo de fazer as comparações das diferentes métricas e analisar os resultados. Nos 4 testes realizados o `icc` venceu três e o `gcc 4.9.0` outro. Analisando ainda os dados obtidos o `icc` foi o compilador que no geral melhores resultados conseguiu.

Armazenamento - Introdução

O problema que nos foi apresentado consiste em analisar a performance dos discos do cluster. A ferramenta utilizada é o IOzone é um Benchmark para sistemas de ficheiros. Gera e mede uma variedade de operações sobre ficheiros, utilizando inclusive *mmap* e POSIX threads. Vencendo até o *Infoworld Bossie Awards* em 2007 como a melhor ferramenta de E/S para ficheiros.

Com estes dados recolhidos foi possível gerar gráficos comparativos entre todos os discos testados e fazer conclusões.

Caracterização do Sistema

Cada nó tem um disco associado, na tabela seguinte faço a sua associação. Para obter os Discos de cada nó recorri ao programa `tentakel`, para dispersão, e `udevadm`, para listagem, com a seguinte parametrização:

```
tentakel -g compute_linux "/sbin/udevadm info -a -p /sys/class/block/sda/sda5 -q env | grep MODEL"
```

Para referências futuras apenas será mencionado o nome do Disco como identificação.

Nó do Cluster	**Disco**
431-3	MB0500EBNCR
431-5	SAMSUNG_HD502HI
431-6	SAMSUNG_HD502HJ
432-1	MM0500EBKAE
541-1	WDC_WD10EZRX-00A8LB0
641-8	INTEL_SSDSC2BW240A3F
641-19	INTEL_SSDSC2BW240A4
652-1	WDC_WD20NPVT-00Z2TT0
662-6	INTEL_SSDSC2BW120A4

IOZONE e Parâmetros de teste

Para este trabalho utilizei o `IOzone` versão `3.397`, escolhi output para ficheiro binário *excel* e nome específico do ficheiro temporário para não coincidir com outros testes concorrentes. Cada ficheiro output e temporário tem o nome do nó associado, neste caso o *431-3*.

```
/opt/iozone/bin/iozone -Ra -b 431_3.xls -f 431_3.tmp
```

Testes utilizados

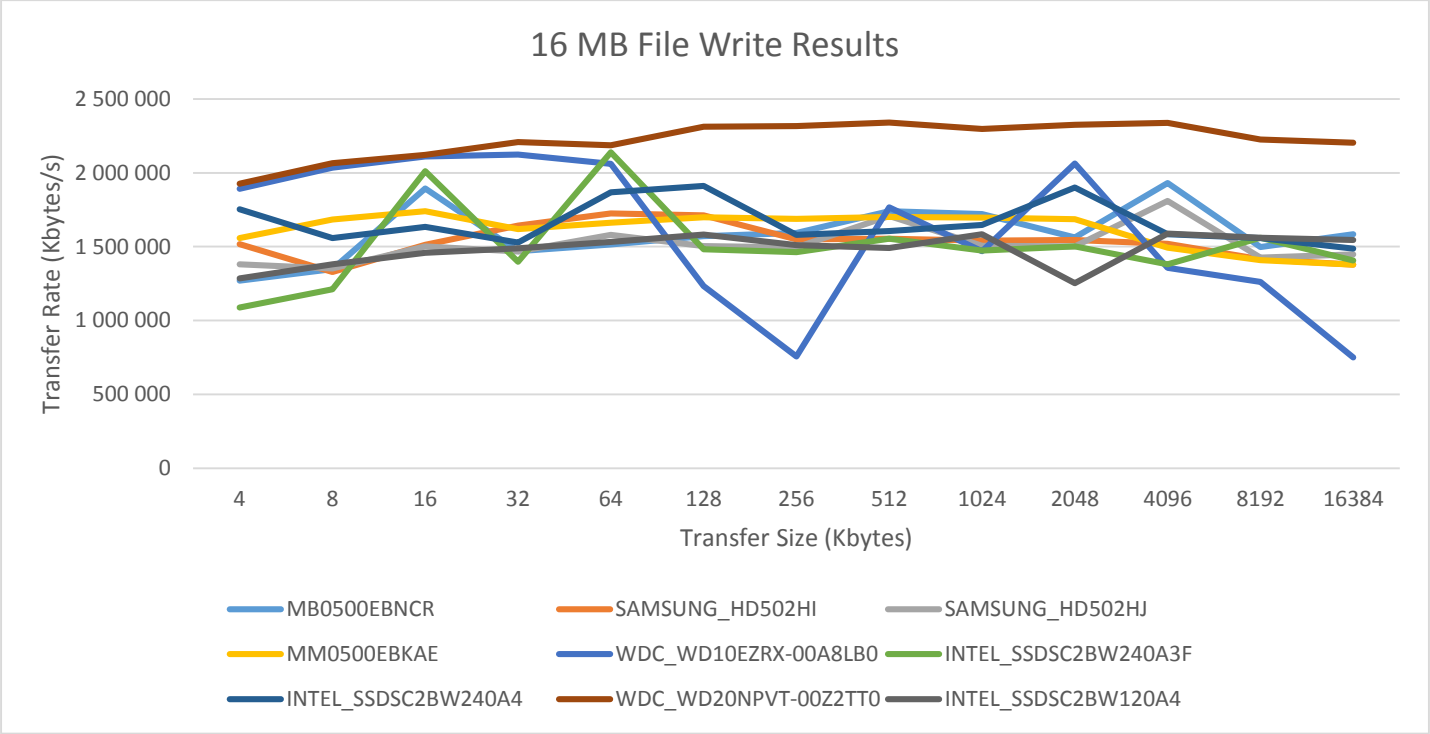
Realizei 13 testes para 512 MB de tamanho máximo de ficheiro. Sendo os testes os seguintes:

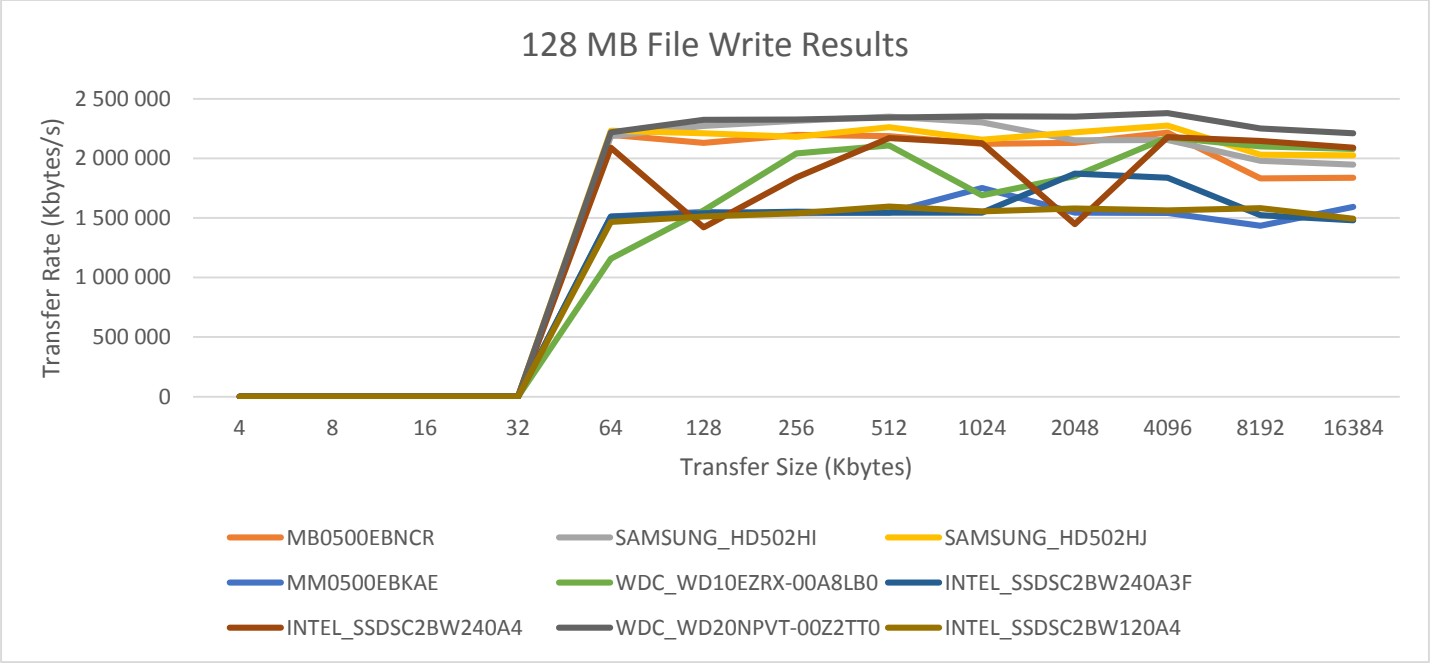
1. Write
2. Rewrite
3. Read
4. Reread
5. Random read
6. Random write
7. Backward read
8. Record rewrite
9. Stride read
10. Fwrite
11. Frewrite
12. Fread
13. Freread.

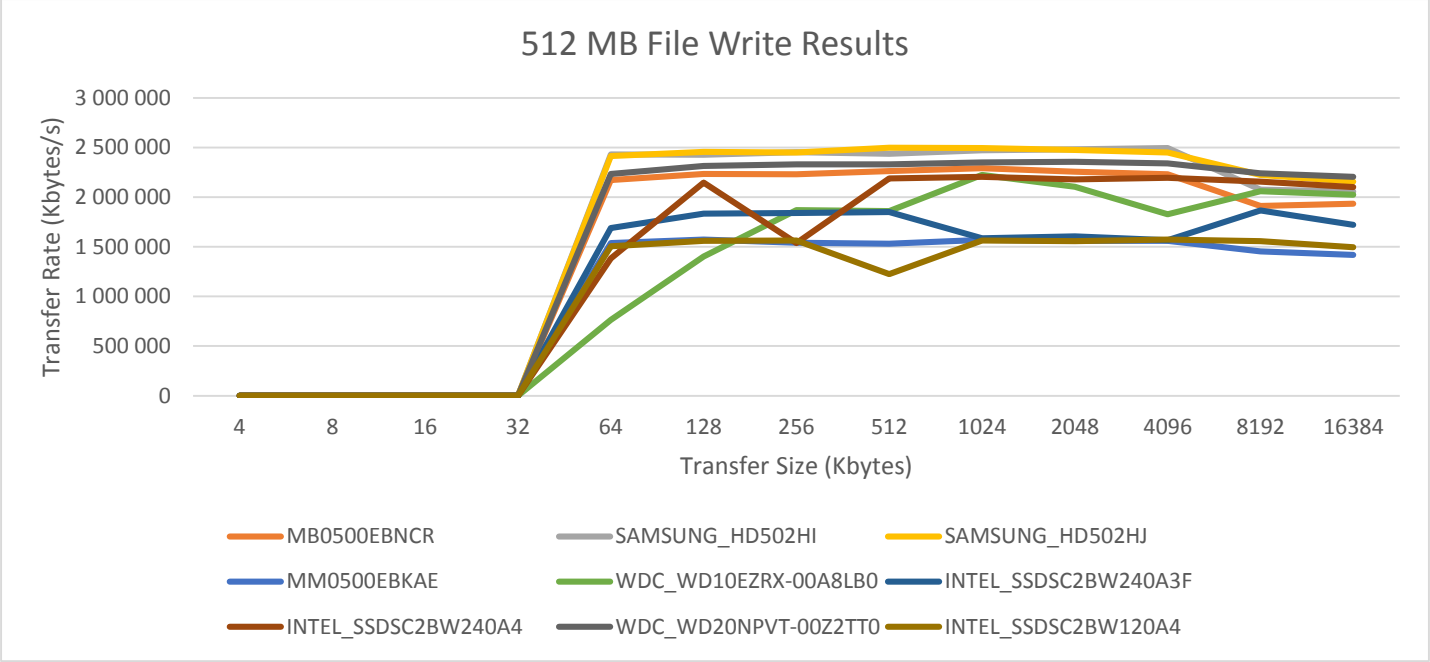
Write

Este teste mede a performance de escrita num ficheiro. Quando um novo ficheiro é escrito não são só os dados guardados mas também a informação sobre onde eles são guardados no disco, chamados de *metadados*. Consiste na informação de diretoria, espaço alocado e outras informações associadas ao ficheiro que não fazem parte dos dados do ficheiro em si. É normal que no início deste teste a performance seja mais baixa devido a esta informação extra.

Com os valores obtidos retirei 3 gráficos para 16 MB, 128 MB e 512 MB de ficheiro de teste.







Conclusão

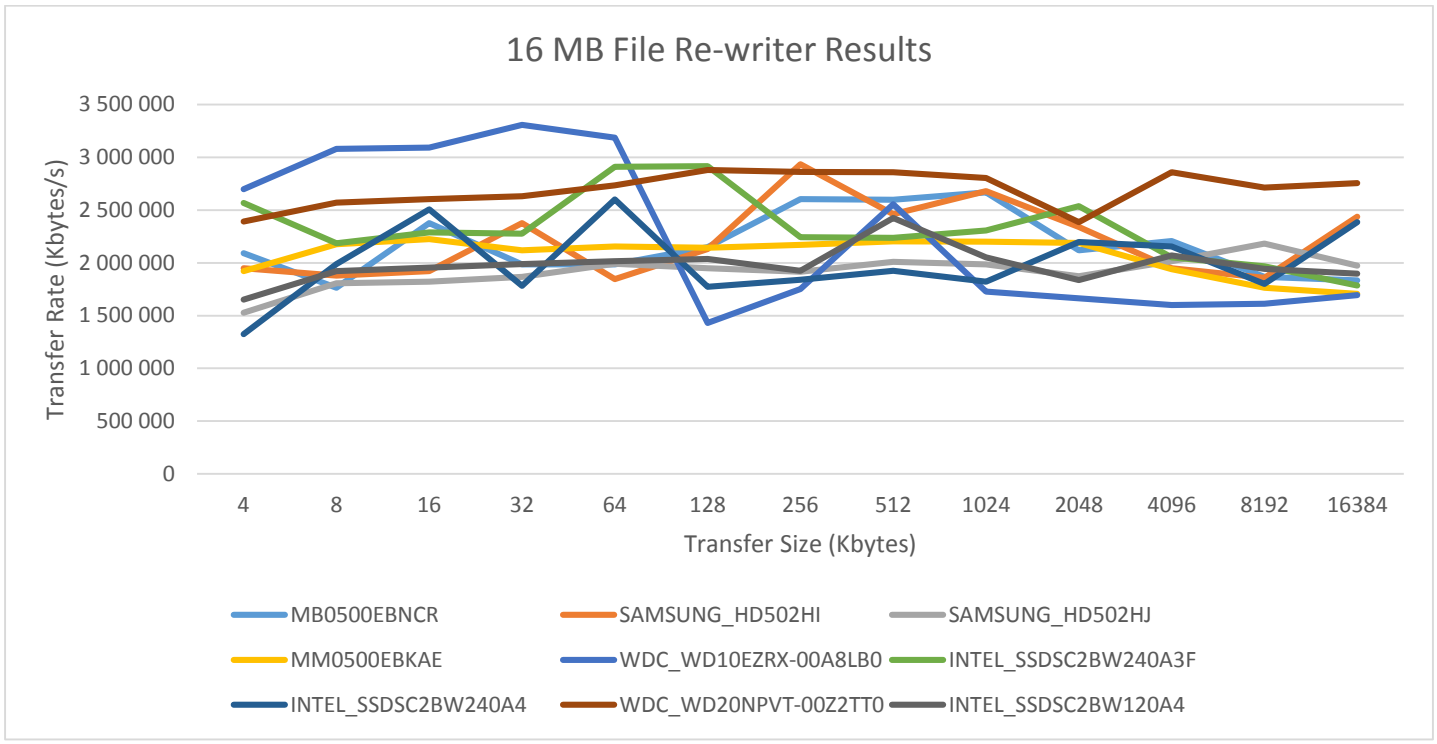
Para 16 MB e 128 MB o **WDC_WD20NPVT-00Z2TT0** foi o melhor, ficando em segundo lugar no teste de 512 MB quando perde para o **SAMSUNG_HD502HJ**.

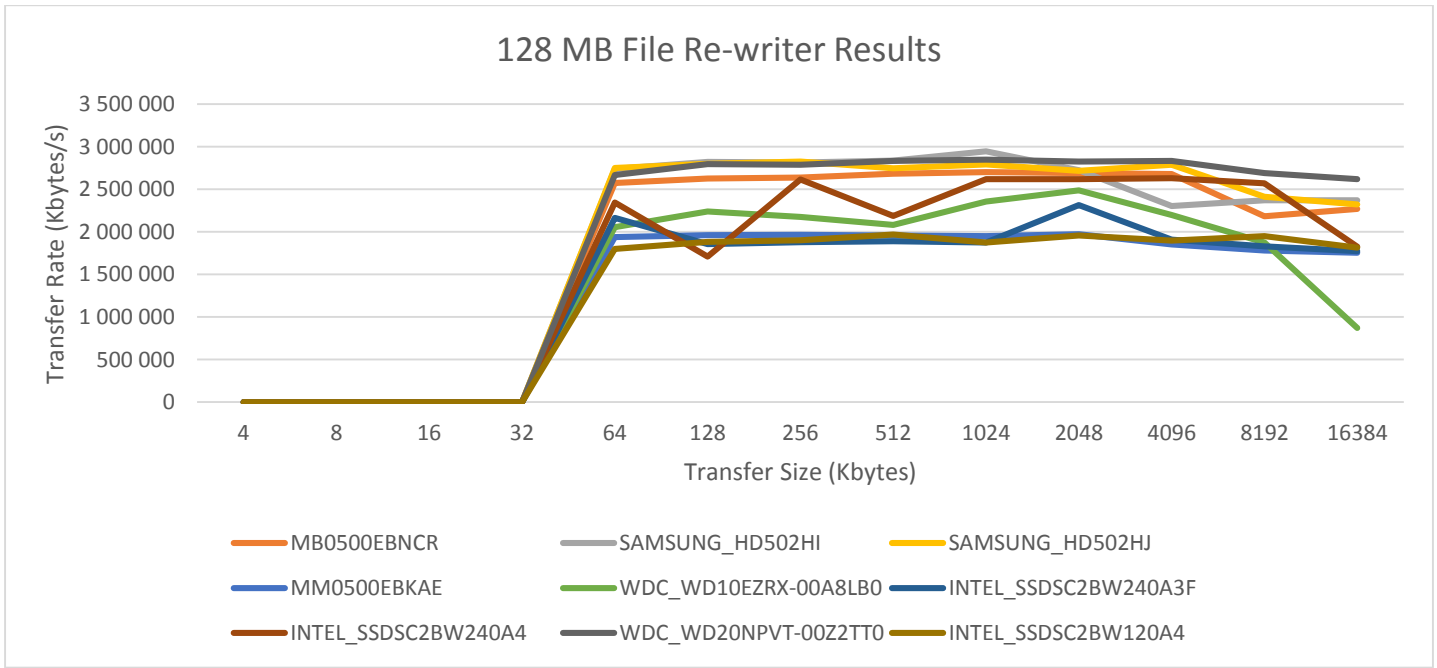
No geral o **WDC_WD20NPVT-00Z2TT0** obteve os melhores resultados.

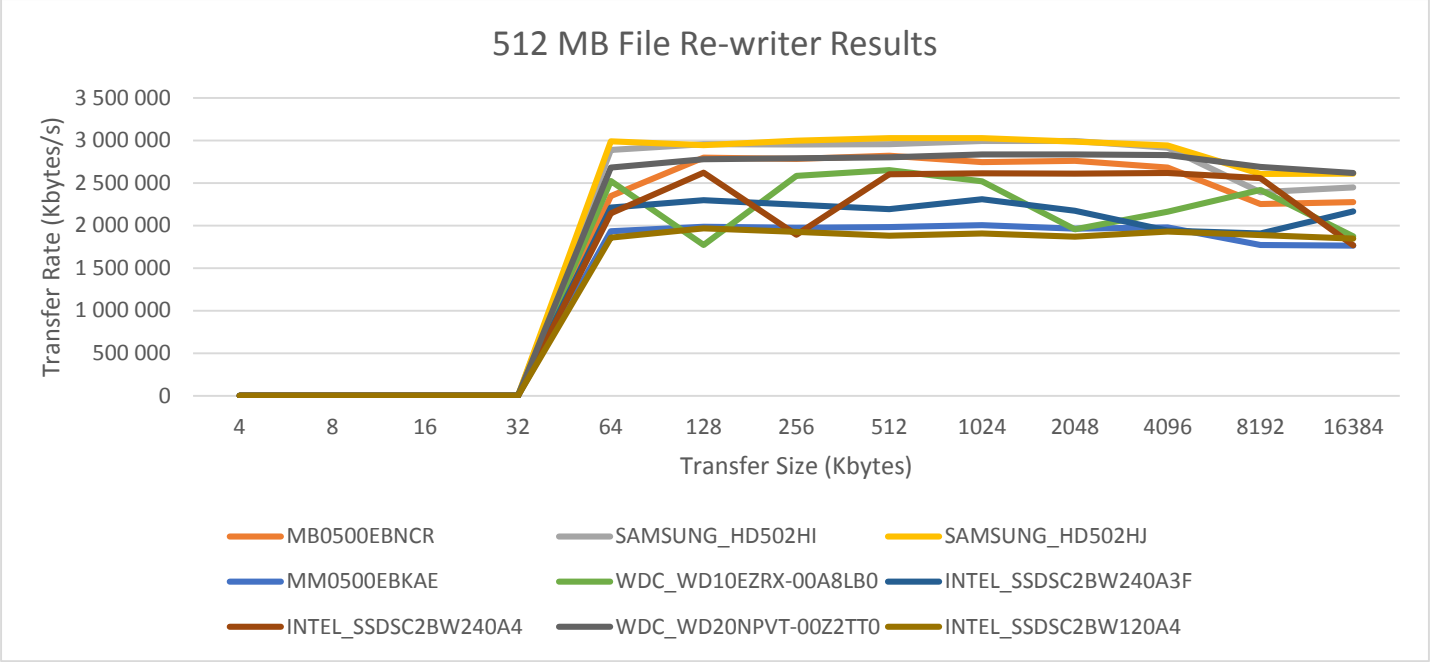
Re-Write

Este teste mede a performance de escrita de um ficheiro que já existe. Sendo assim é preciso menos trabalho pois já existem metadados guardados. É expectável que a performance deste teste seja superior ao anterior, pelas razões apresentadas.

Com os valores obtidos retirei 3 gráficos para 16 MB, 128 MB e 512 MB.







Conclusão

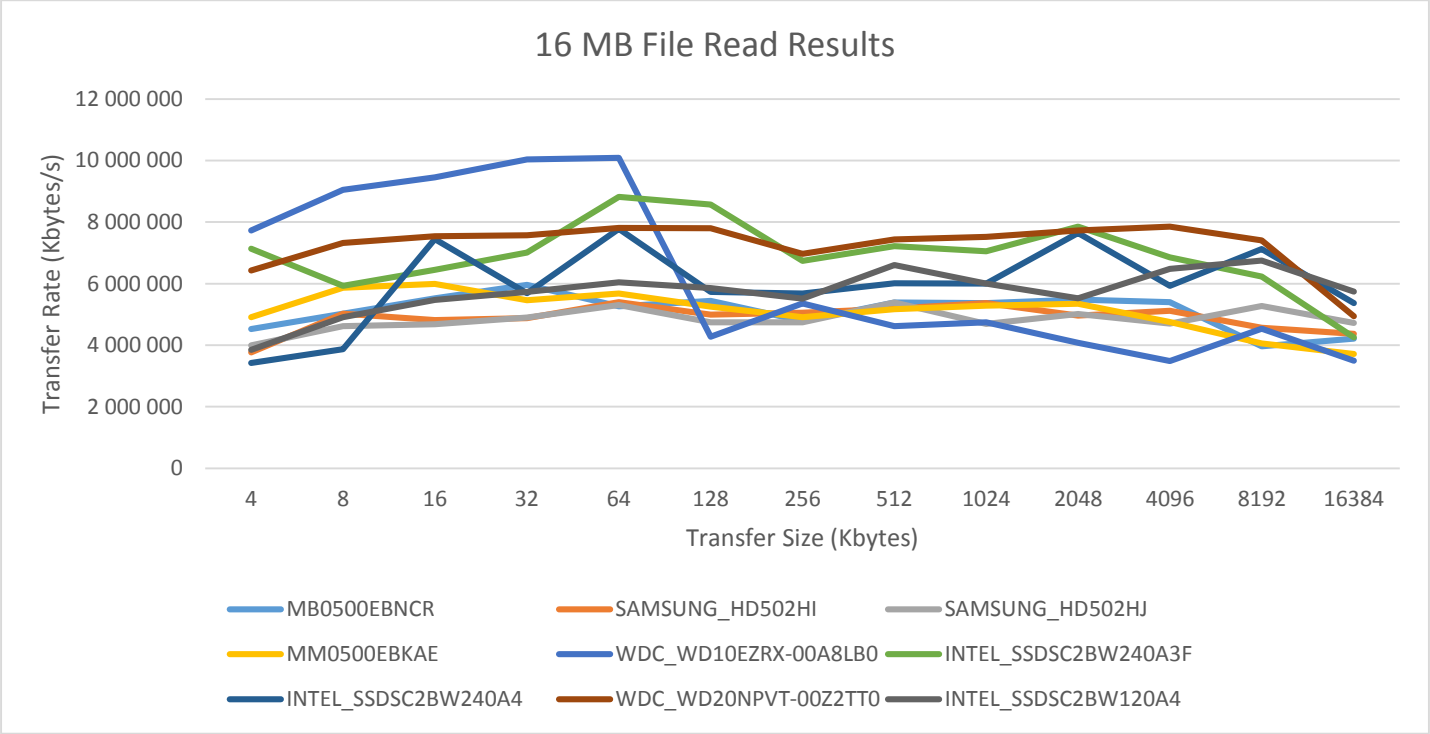
Como previsto todos os testes foram melhores que os de Write.

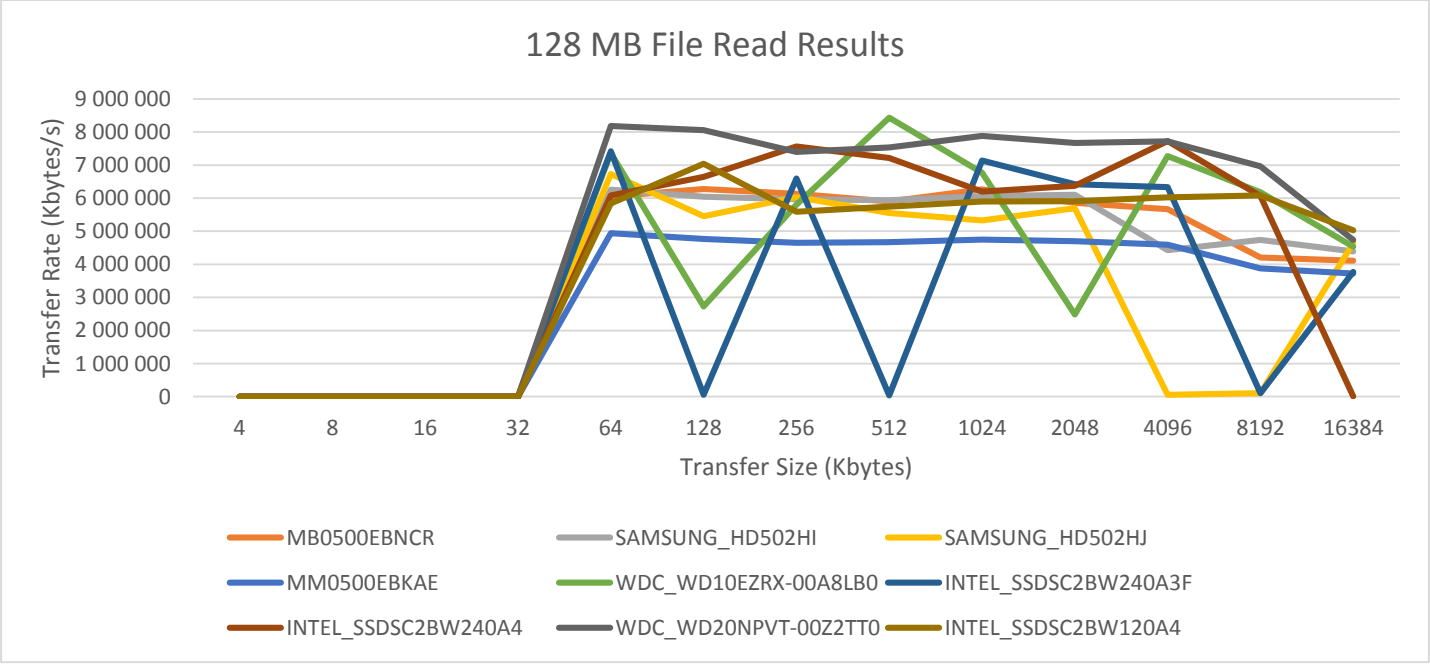
Para 16 MB o **WDC_WD20NPVT-00Z2TT0** foi o melhor, depois para 128 MB e 512 MB há de novo pouca diferença entre o **WDC_WD20NPVT-00Z2TT0** e o **SAMSUNG_HD502HJ**.

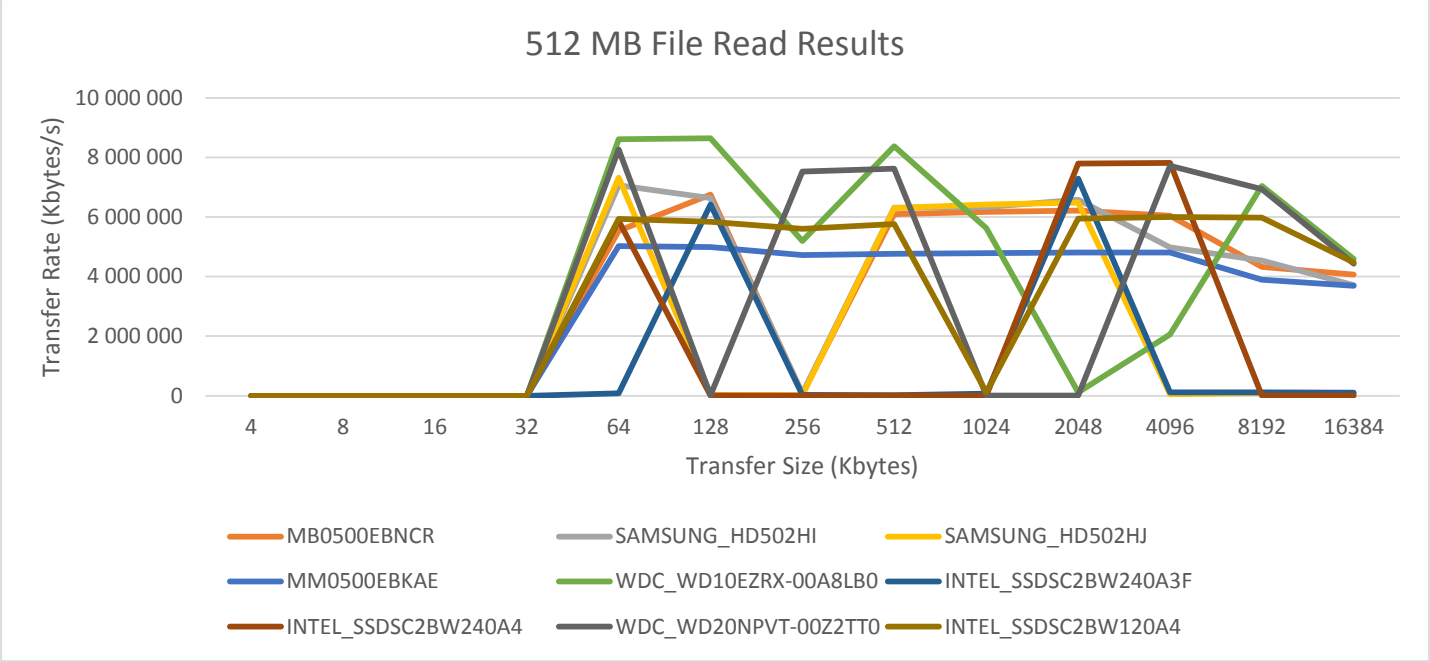
No geral o **WDC_WD20NPVT-00Z2TT0** obteve os melhores resultados.

Read

Este teste mede a performance de leitura de um ficheiro existente.
Com os valores obtidos retirei 3 gráficos para 16 MB, 128 MB e 512 MB.







Conclusão

Há medida que iam aumentando os valores do ficheiro de teste as oscilações foram aumentando.

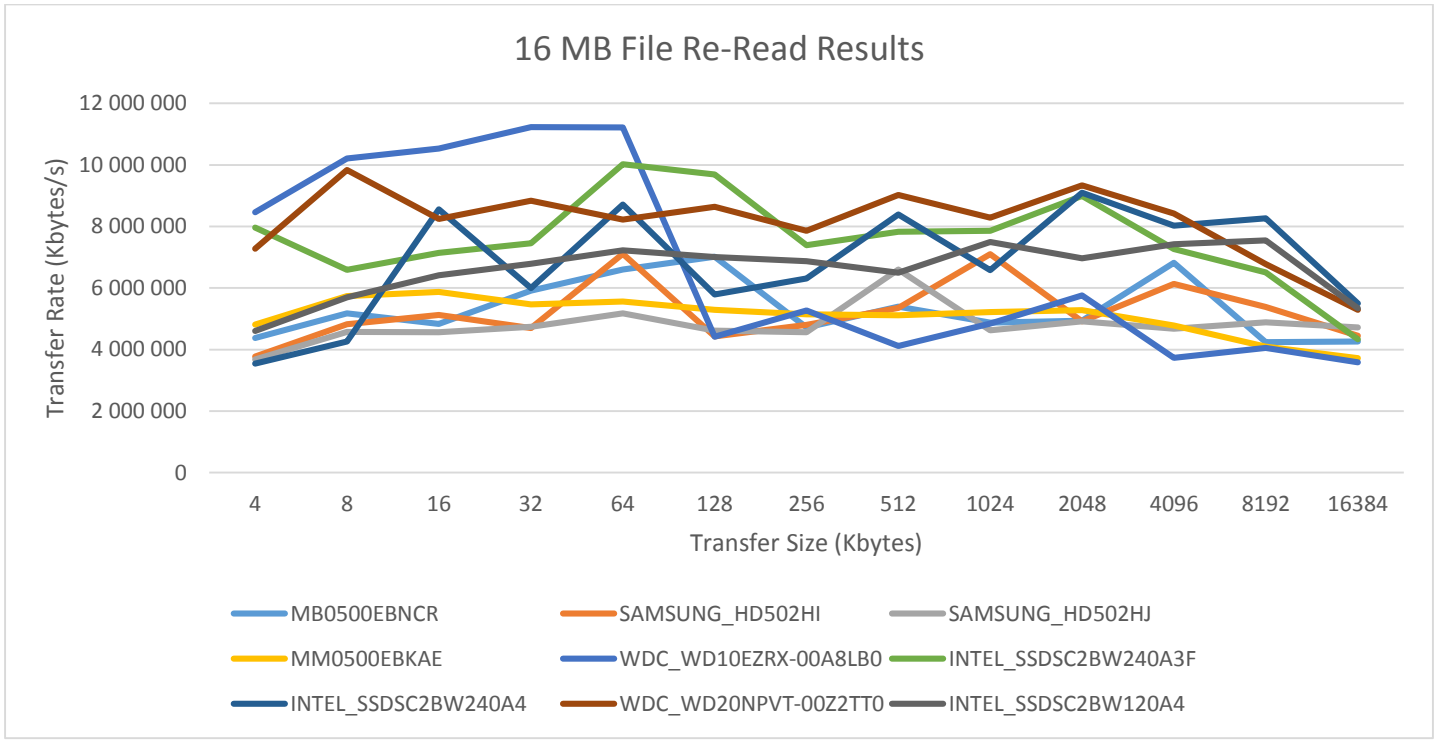
Para 16 MB e 128 MB o **WDC_WD20NPVT-00Z2TT0** foi o melhor e mais consistente. Para 512 MB apenas há um disco que se revelou constante, o **MM0500EBKAE**, que também no de 126 MB também se manteve.

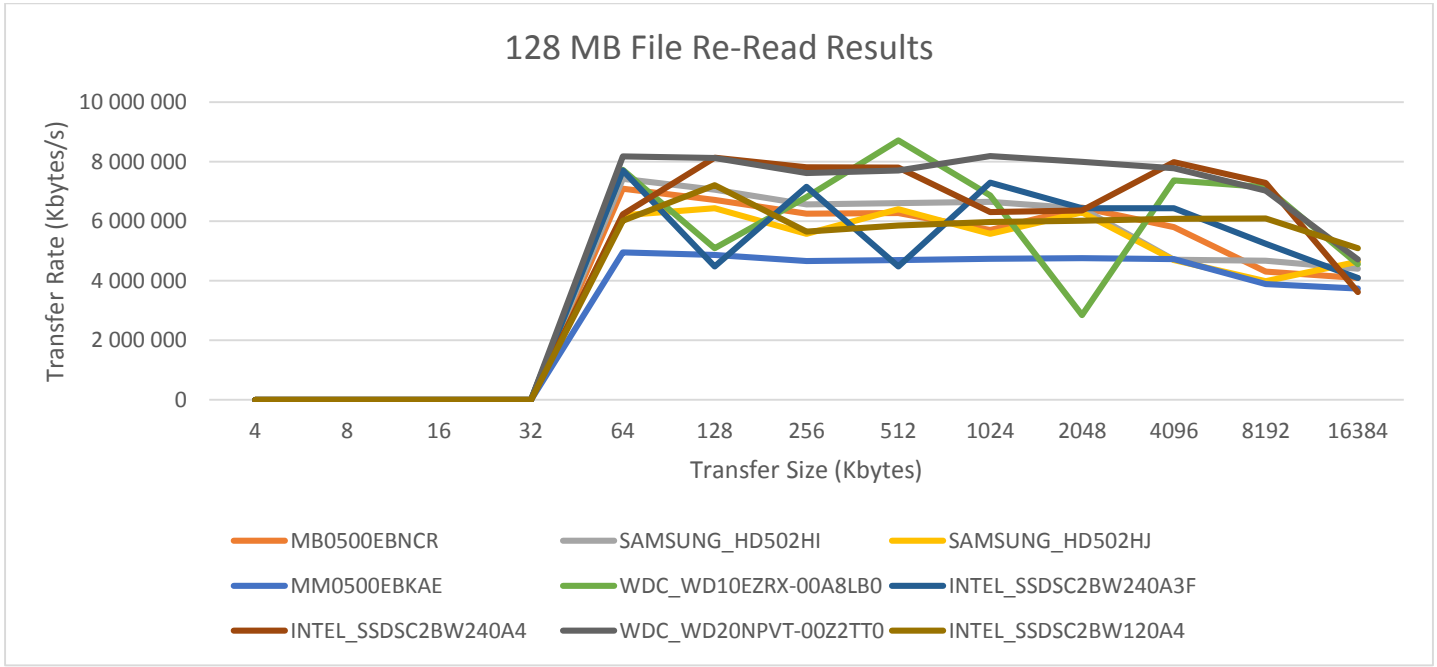
No geral o **WDC_WD20NPVT-00Z2TT0** obteve os melhores resultados.

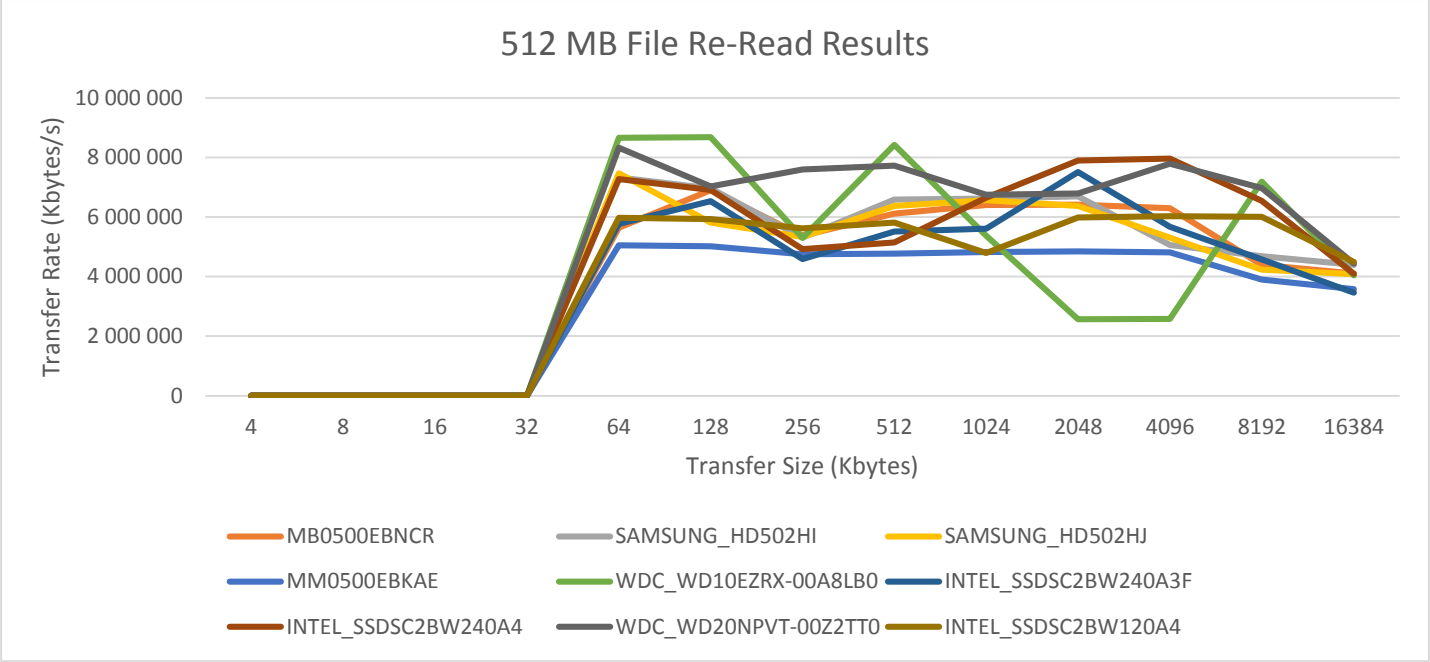
Re-Read

Este teste analisa a performance de leitura de um ficheiro que foi lido recentemente, sendo expectável que a performance seja maior que a simples leitura (Read). As caches e suas latências têm influência neste teste pois como estamos a tratar de lados que foram lidos há pouco tempo só fará sentido, para este teste, se estiverem em cache.

Com os valores obtidos retirei 3 gráficos para 16 MB, 128 MB e 512 MB.







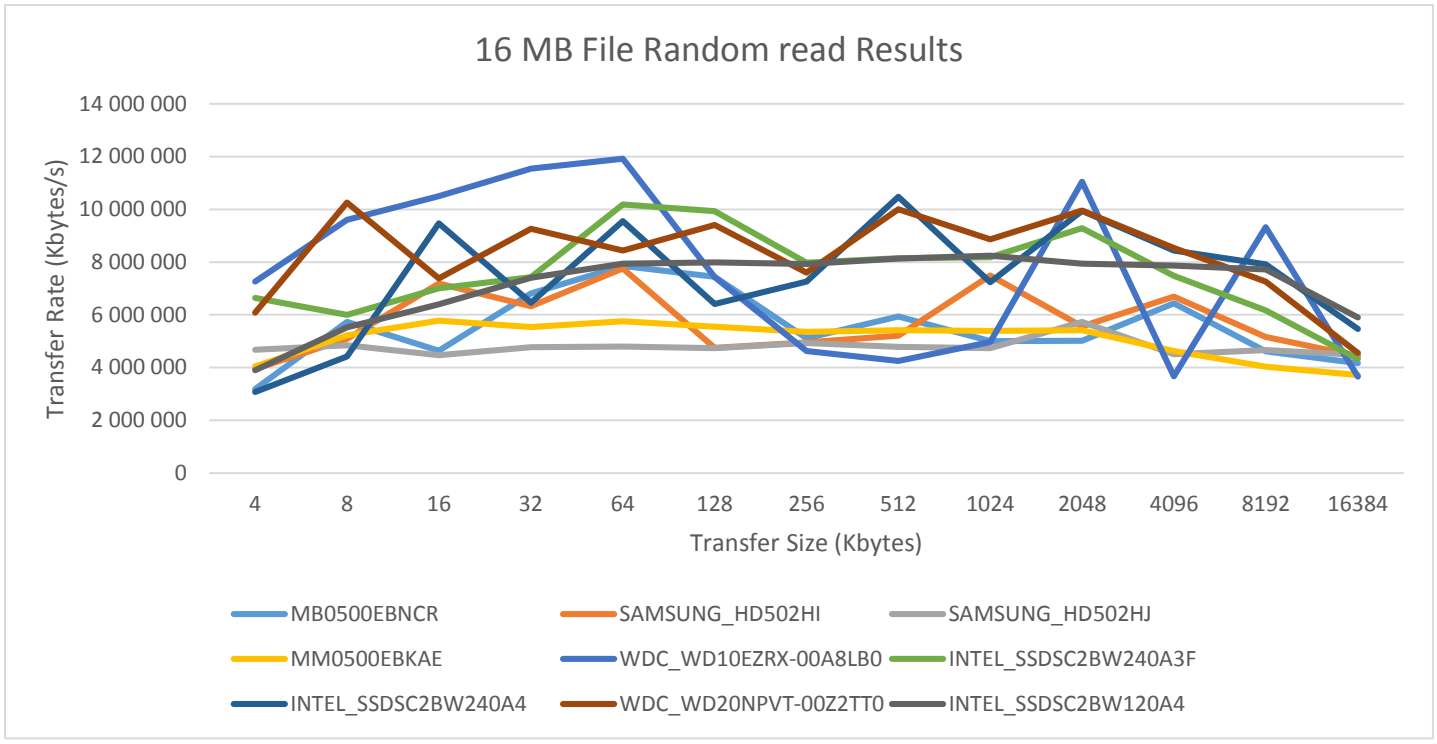
Conclusão

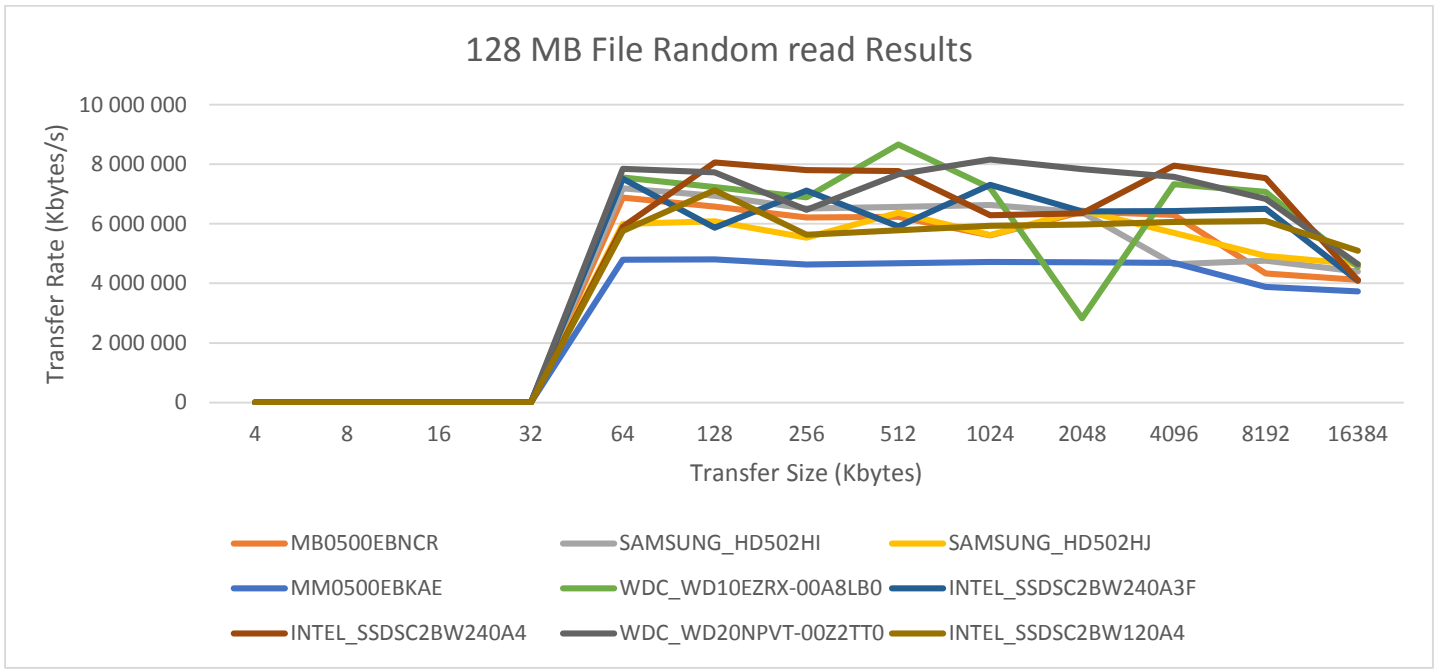
Os testes não foram muito melhores que os de Read, mas nota-se um pequeno aumento.
Nos 3 tamanhos o **WDC_WD20NPVT-00Z2TT0** foi no geral o melhor.

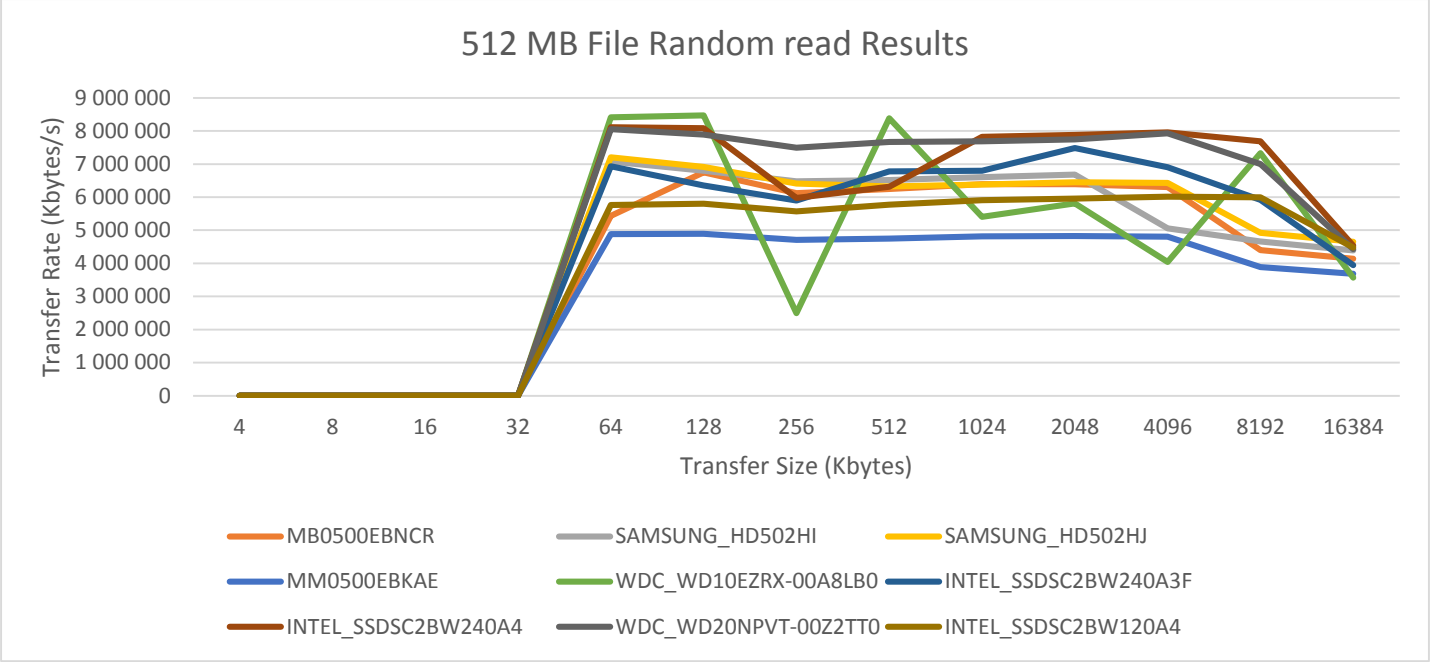
Random Read

Até agora os acessos a um ficheiro eram praticamente contínuos, mas com este teste serão aleatórios. Portanto a cache será um fator que poderá ajudar em certas ocasiões.

Com os valores obtidos retirei 3 gráficos para 16 MB, 128 MB e 512 MB.







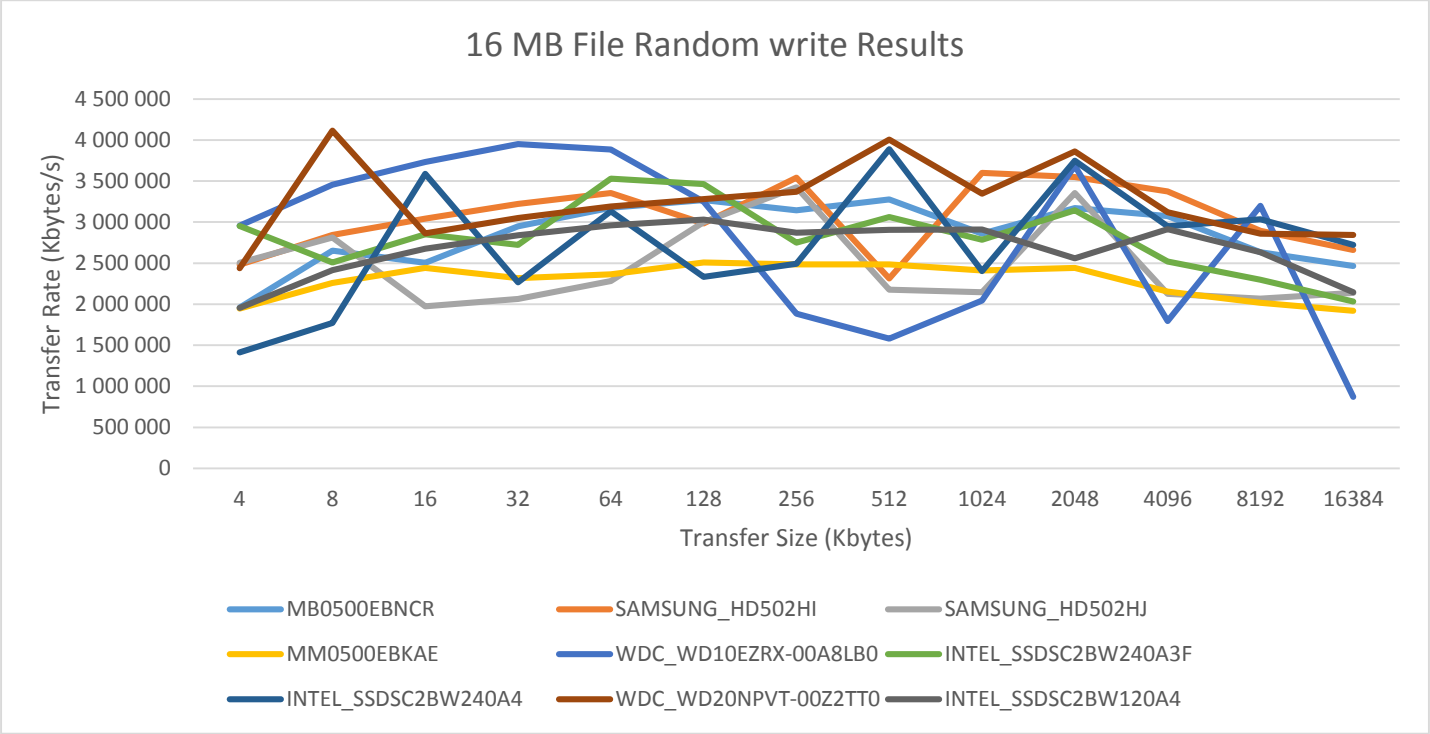
Conclusão

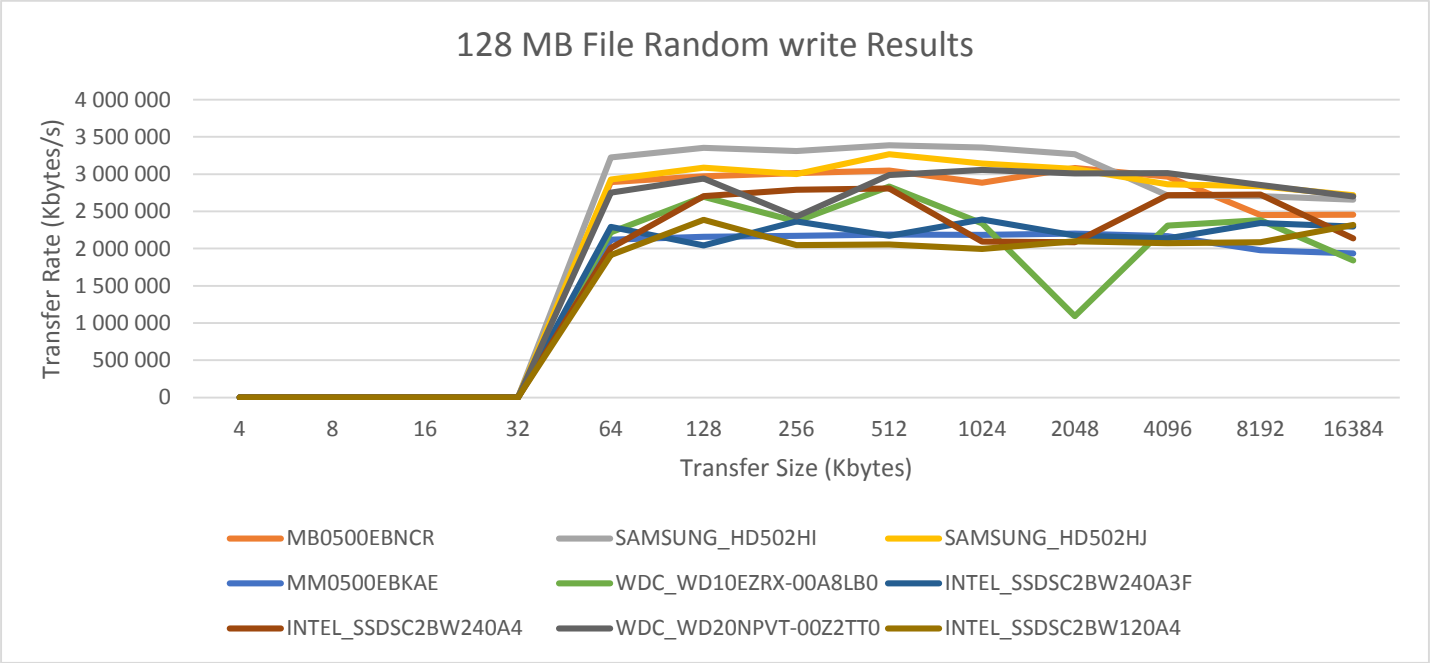
As oscilações eram previsíveis pois os acessos são irregulares.

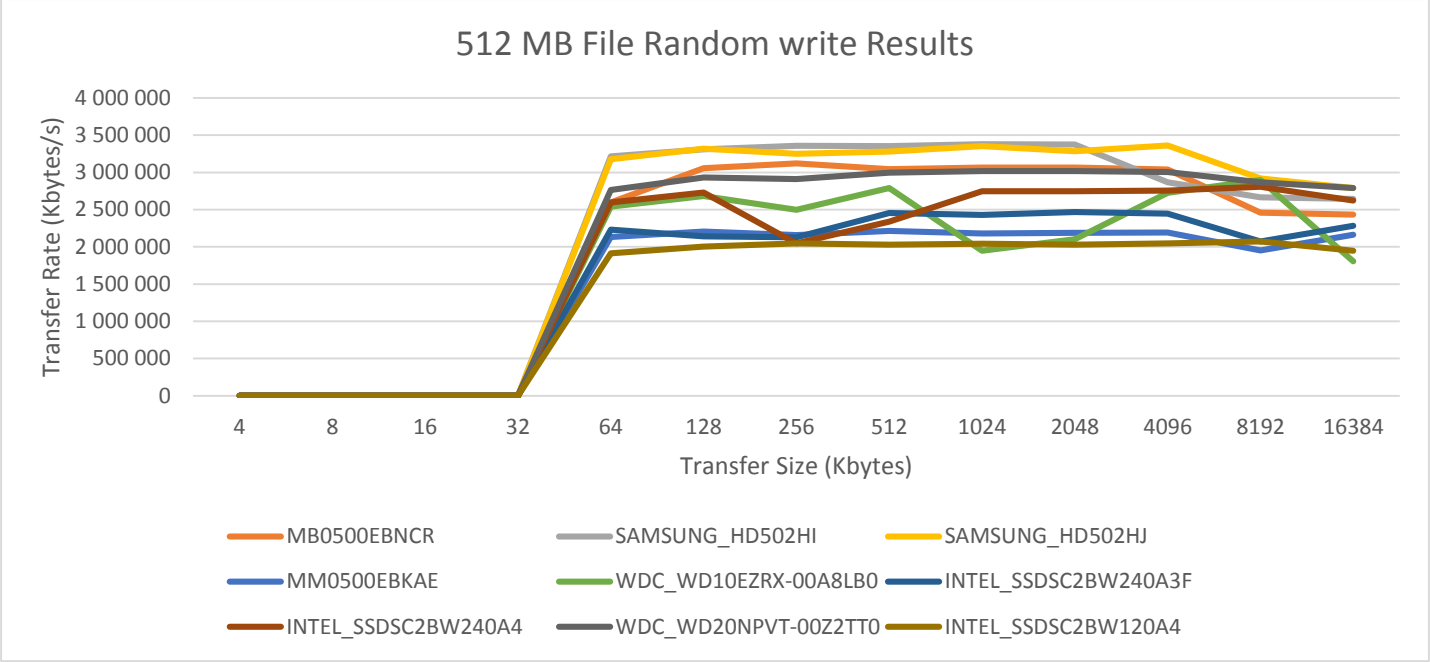
Nos 3 tamanhos o **WDC_WD20NPVT-00Z2TT0** foi no geral o melhor. Sendo que nos 128 MB o **INTEL_SSDSC2BW240A4** esteve perto do topo.

Random Write

Como o teste anterior, os acessos são aleatórios mas neste teste tratam-se de escritas.
Com os valores obtidos retirei 3 gráficos para 16 MB, 128 MB e 512 MB.







Conclusão

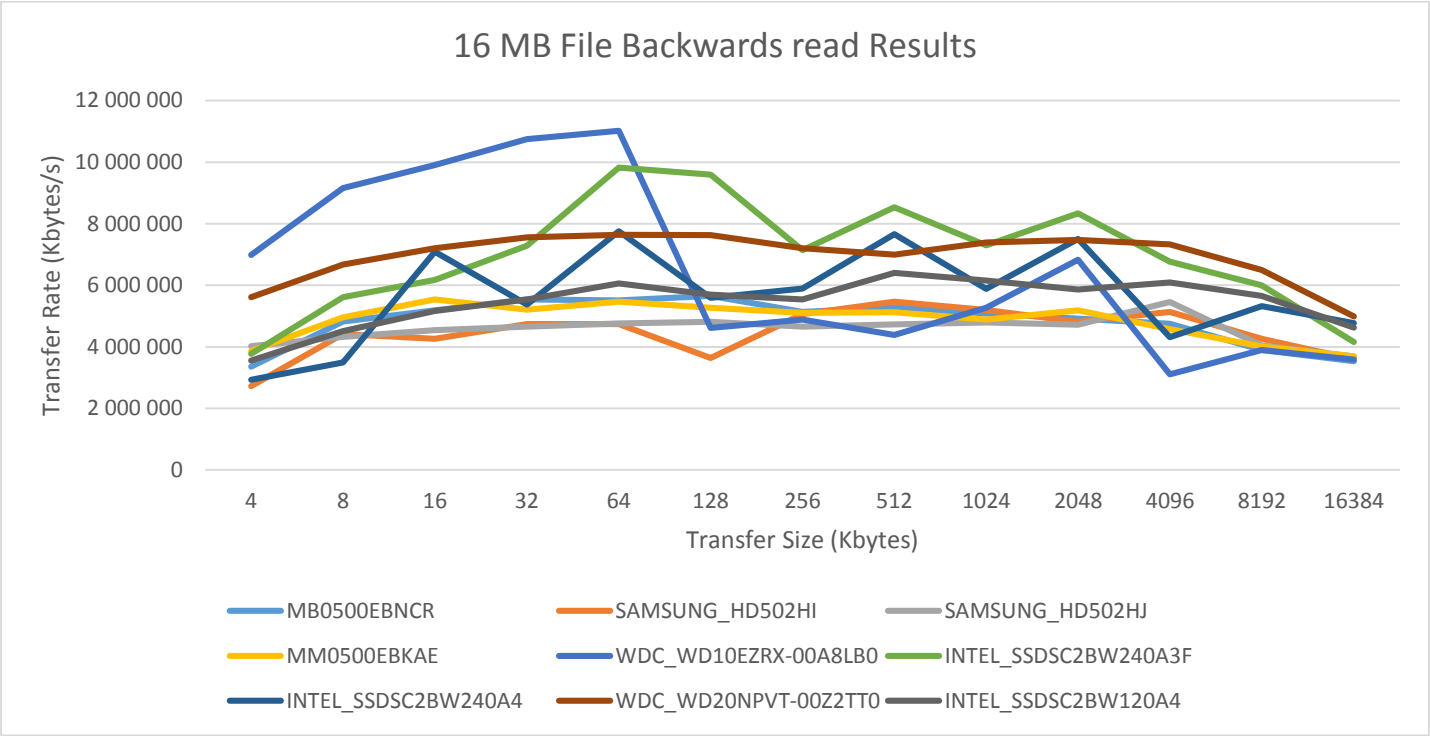
Os valores obtidos estão mais altos que os de Write normal.

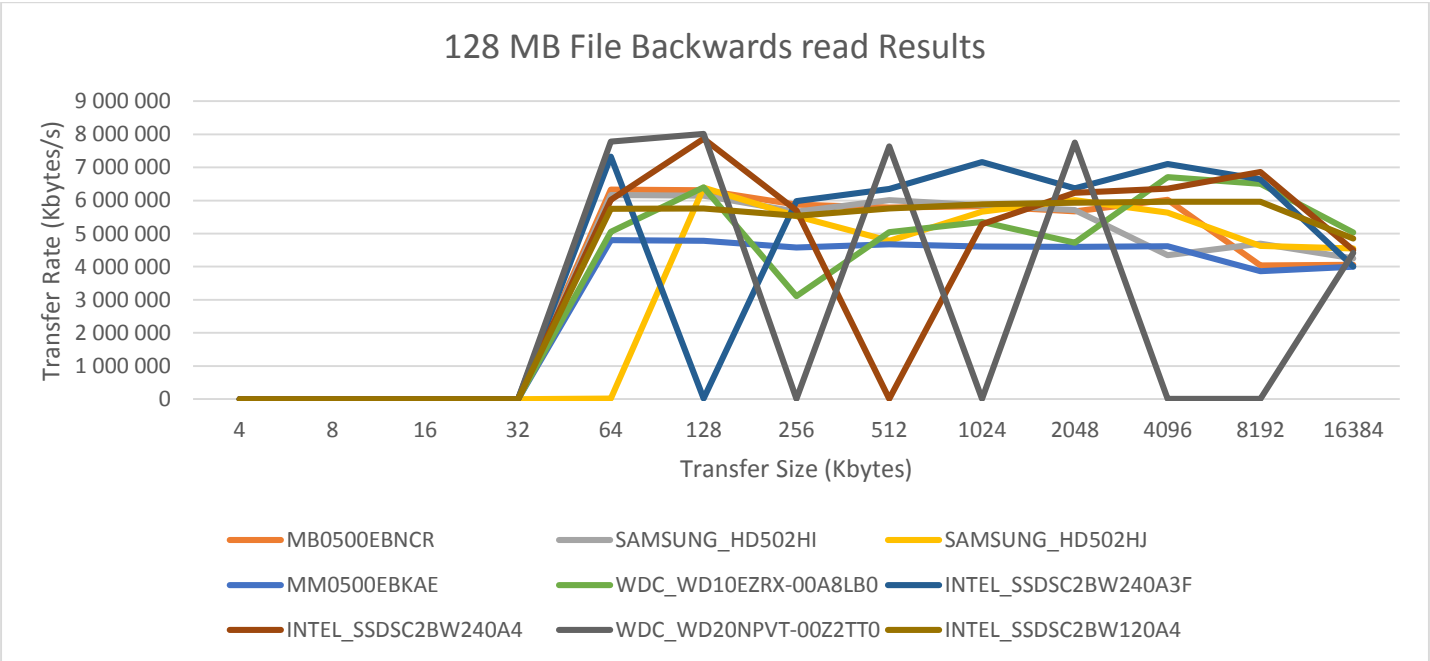
Nos 2 tamanhos iniciais o **WDC_WD20NPVT-00Z2TT0** foi no geral o melhor mas para 512 MB o **SAMSUNG_HD502HJ** leva uma ligeira vantagem.

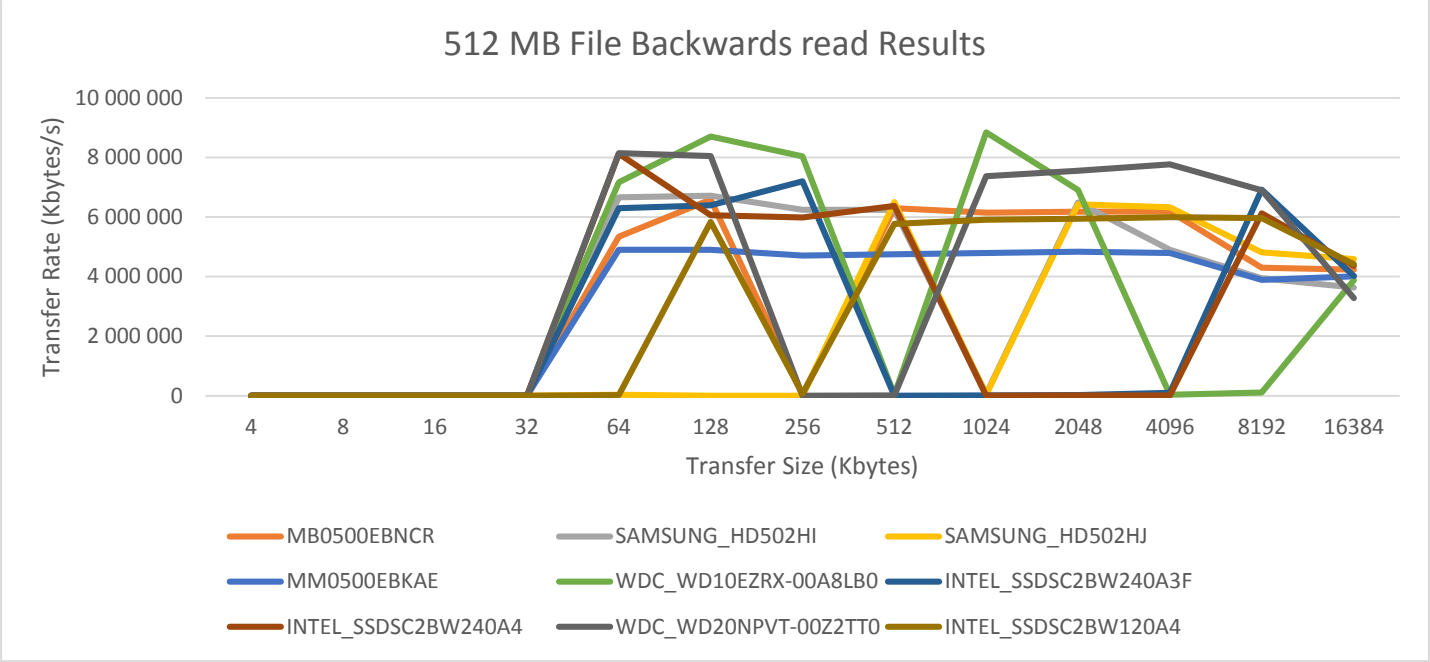
Backwards Read

Este teste irá testar a leitura de um ficheiro para trás. É uma forma estranha de ler um ficheiro mas pode acontecer que um programa tenha necessidade de o fazer. Alguns Sistemas Operativos detetam este tipo de leitura e podem aumentar a performance da leitura para trás.

Com os valores obtidos retirei 3 gráficos para 16 MB, 128 MB e 512 MB.







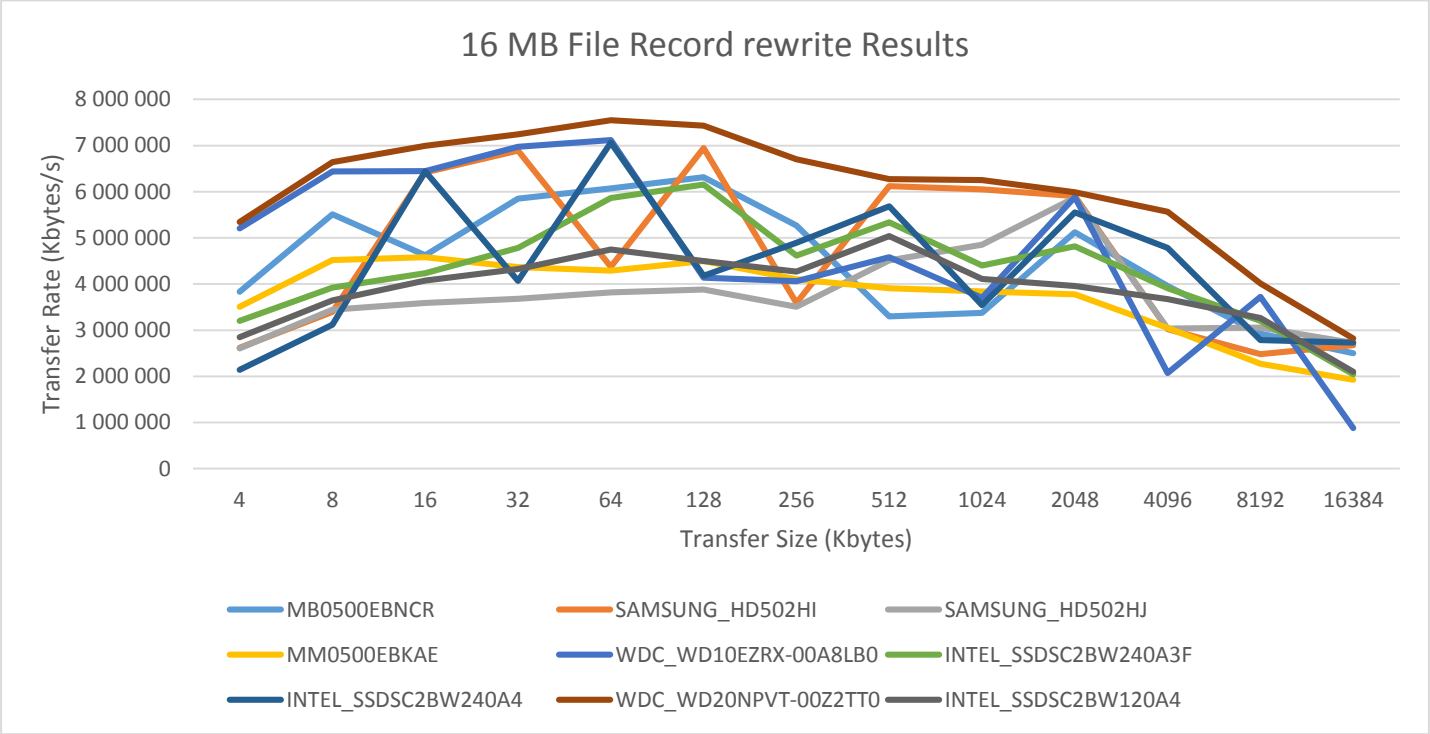
Conclusão

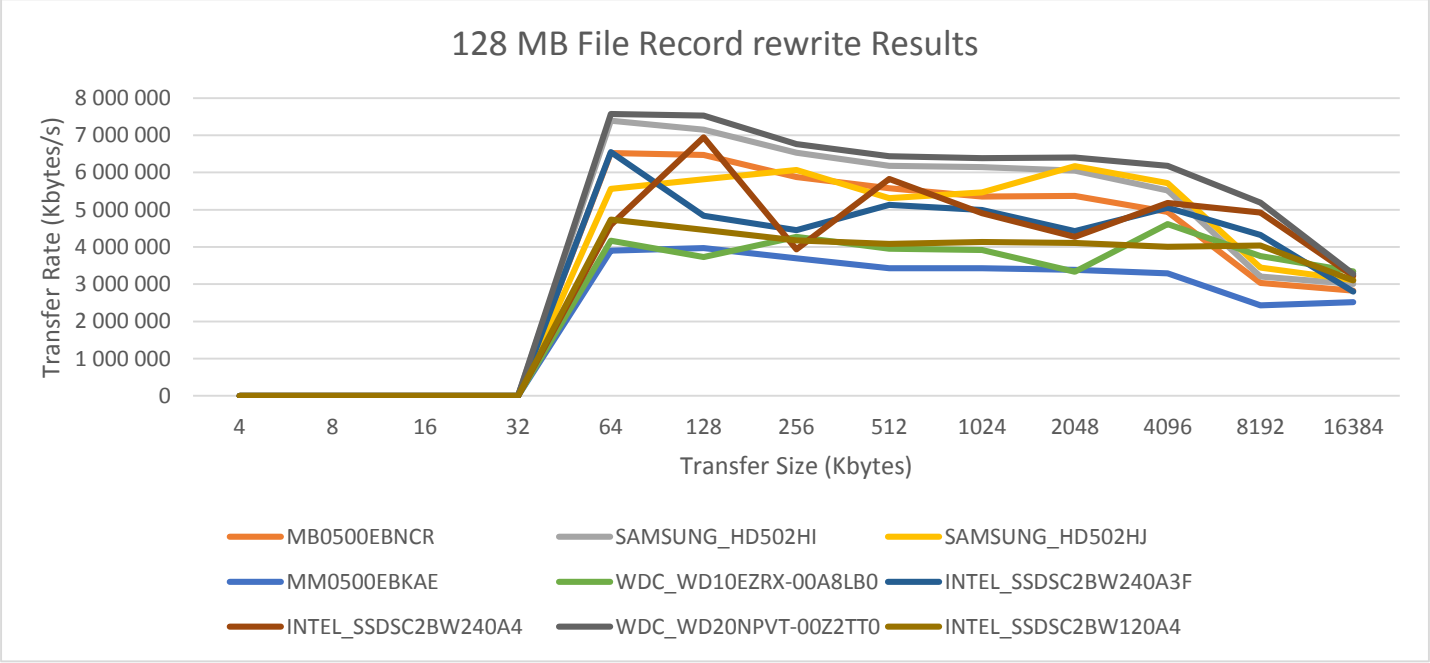
Para 16 MB o **INTEL_SSDSC2BW240A3F** apresenta melhores resultados, mas o **WDC_WD20NPVT-00Z2TT0** foi o que apresentou valores mais altos e constantes. Para 128MB o **WDC_WD10EZRX-00A8LB0** conseguiu valores altos e constantes sem oscilações bruscas. Já para 512MB o **INTEL_SSDSC2BW240A4** foi o que no total obteve maio *throughput*.

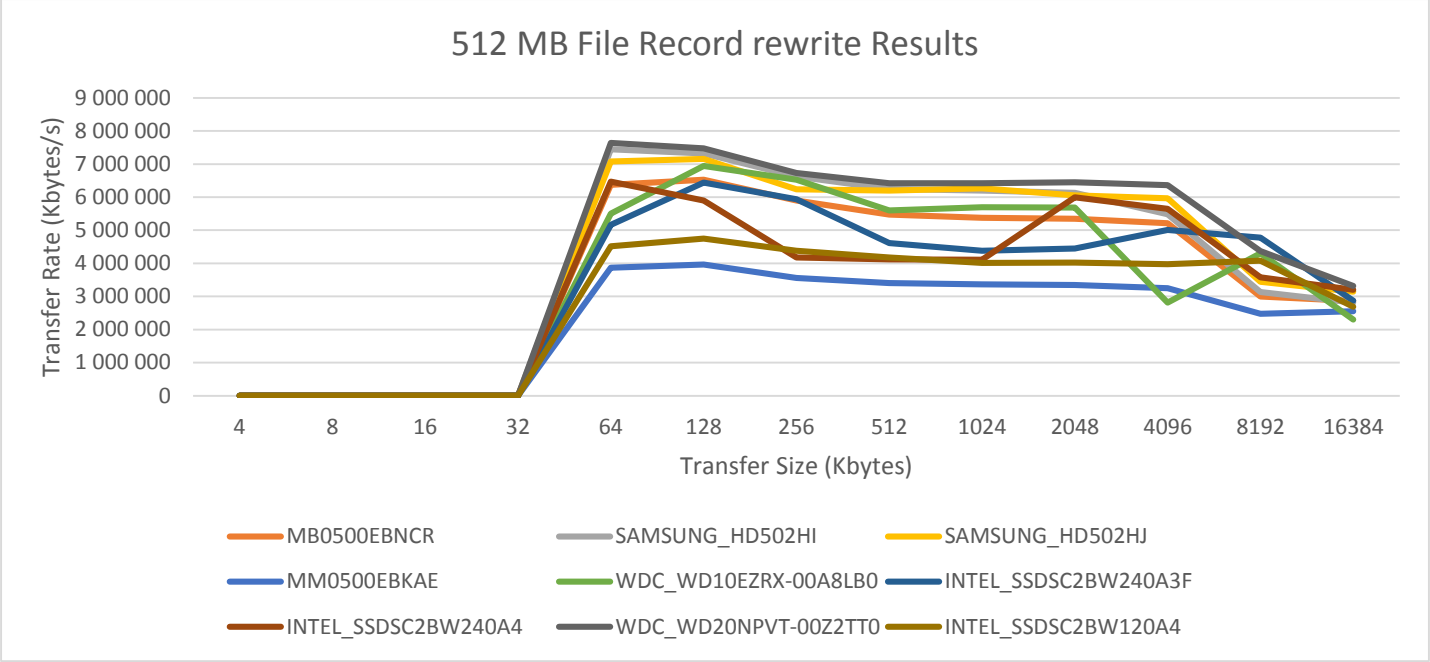
Record Rewrite

Este teste irá medir a performance na escrita e reescrita de um pedaço de informação dentro de um ficheiro. Podem acontecer coisas interessantes, se o pedaço for suficientemente pequeno que caiba na cache a performance será muito alta, sempre que aumentamos o tamanho do ficheiro a performance tenderá a diminuir pois será maior que as diferentes caches no sistema.

Com os valores obtidos retirei 3 gráficos para 16 MB, 128 MB e 512 MB.







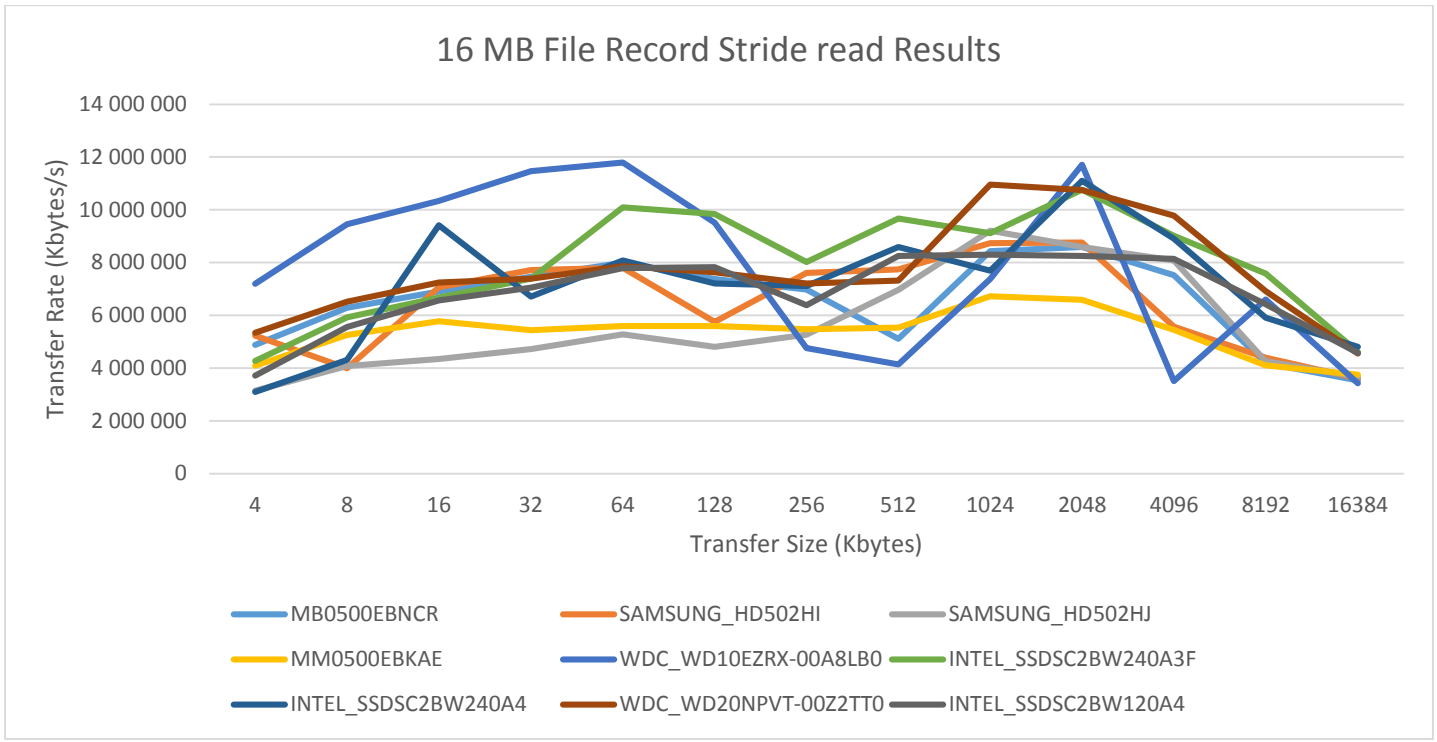
Conclusão

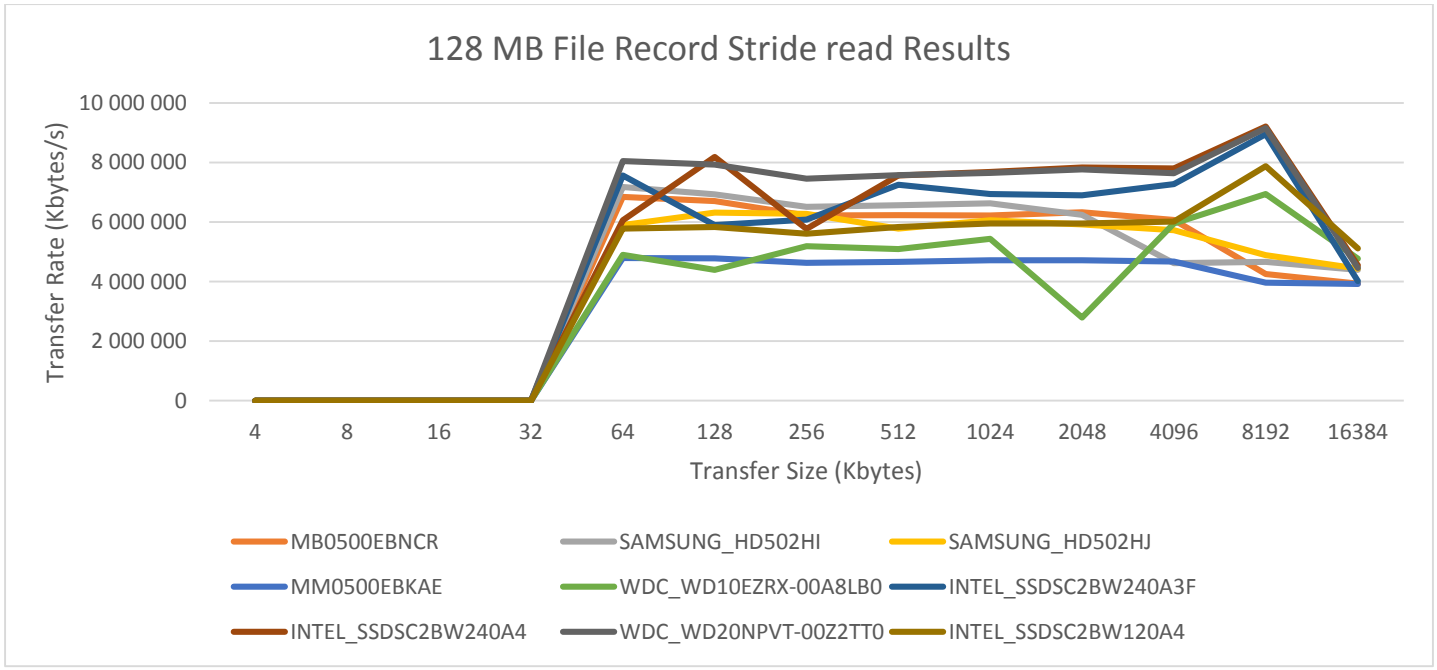
Por mais uma vez o **WDC_WD20NPVT-00Z2TT0** foi o que apresentou melhores resultados nos 3 tamanhos, resultados esses bastante constantes.

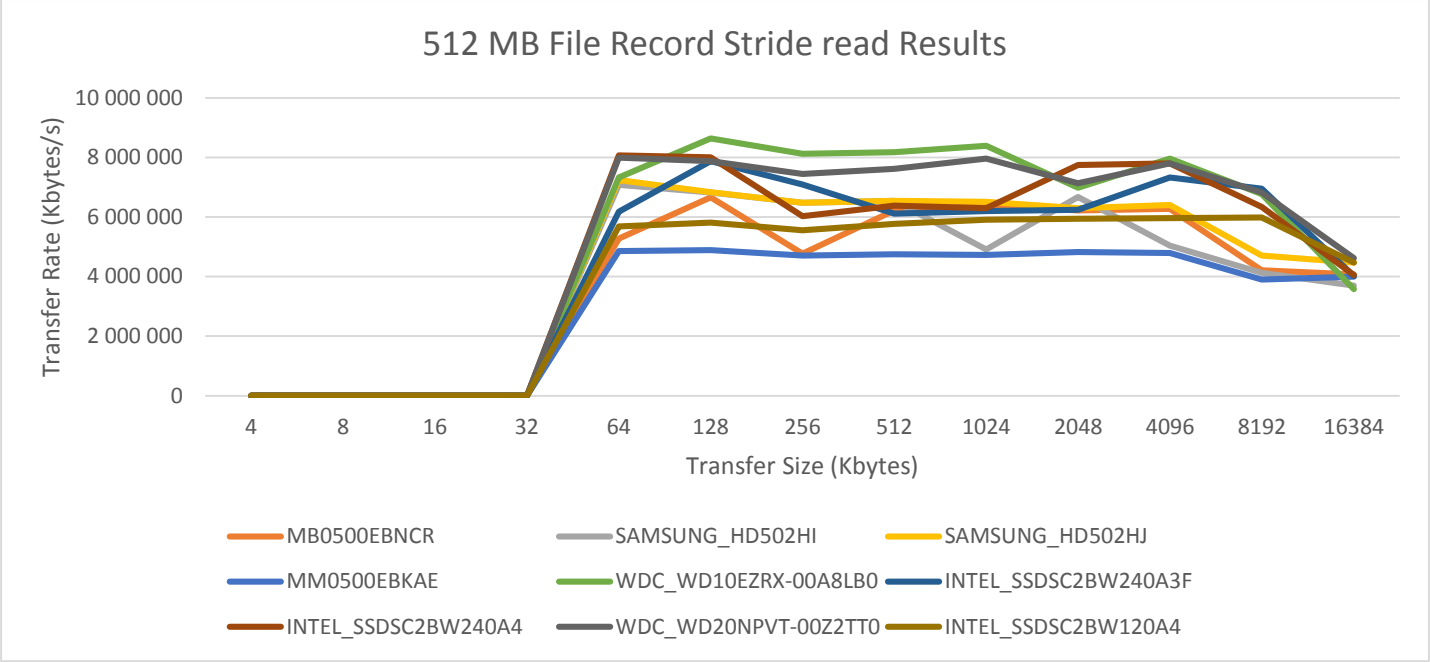
Strided Read

Este teste analisa a performance na leitura de um ficheiro com saltos de x Bytes como um padrão. Este acesso é usual em programas que acedem a regiões particulares em estruturas de dados. É uma acesso difícil de detetar pelo SO, produzindo interessantes anomalias de performance.

Com os valores obtidos retirei 3 gráficos para 16 MB, 128 MB e 512 MB.







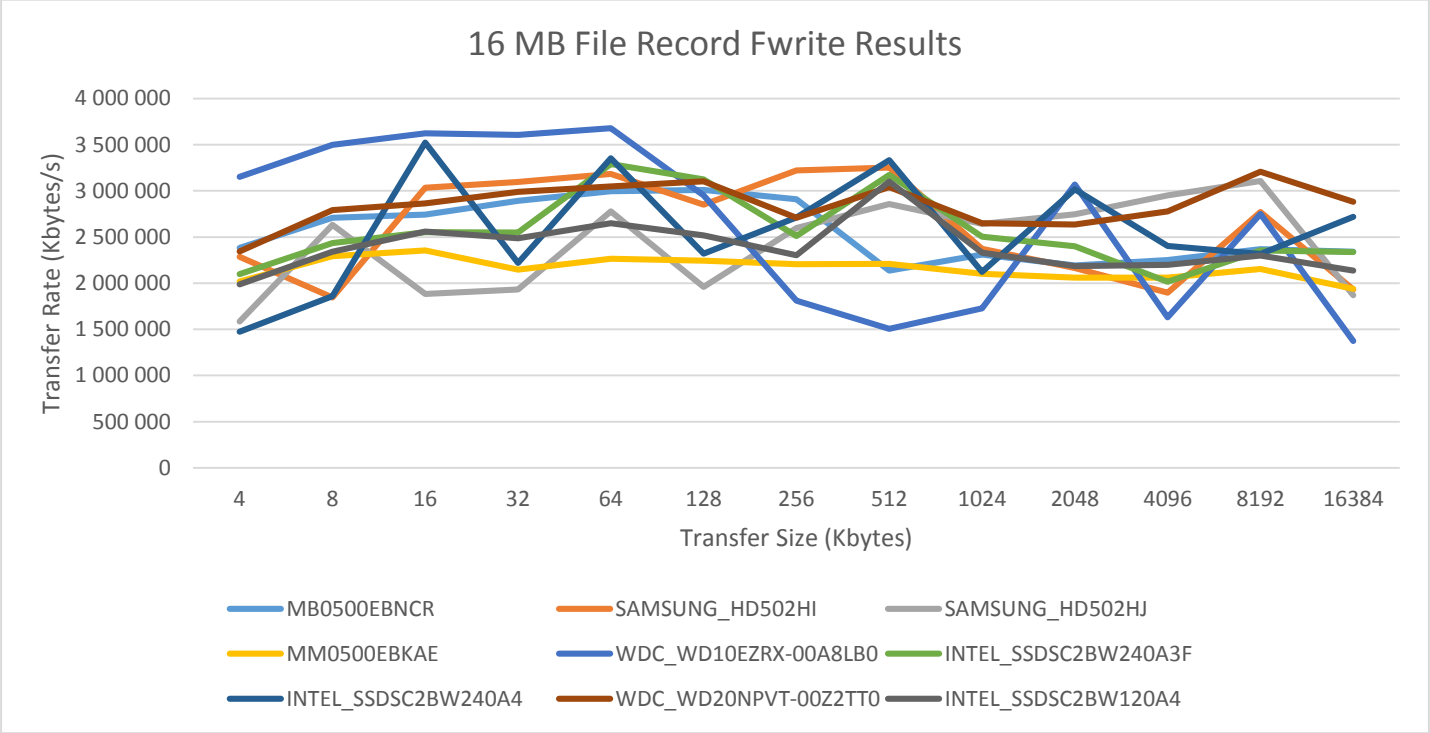
Conclusão

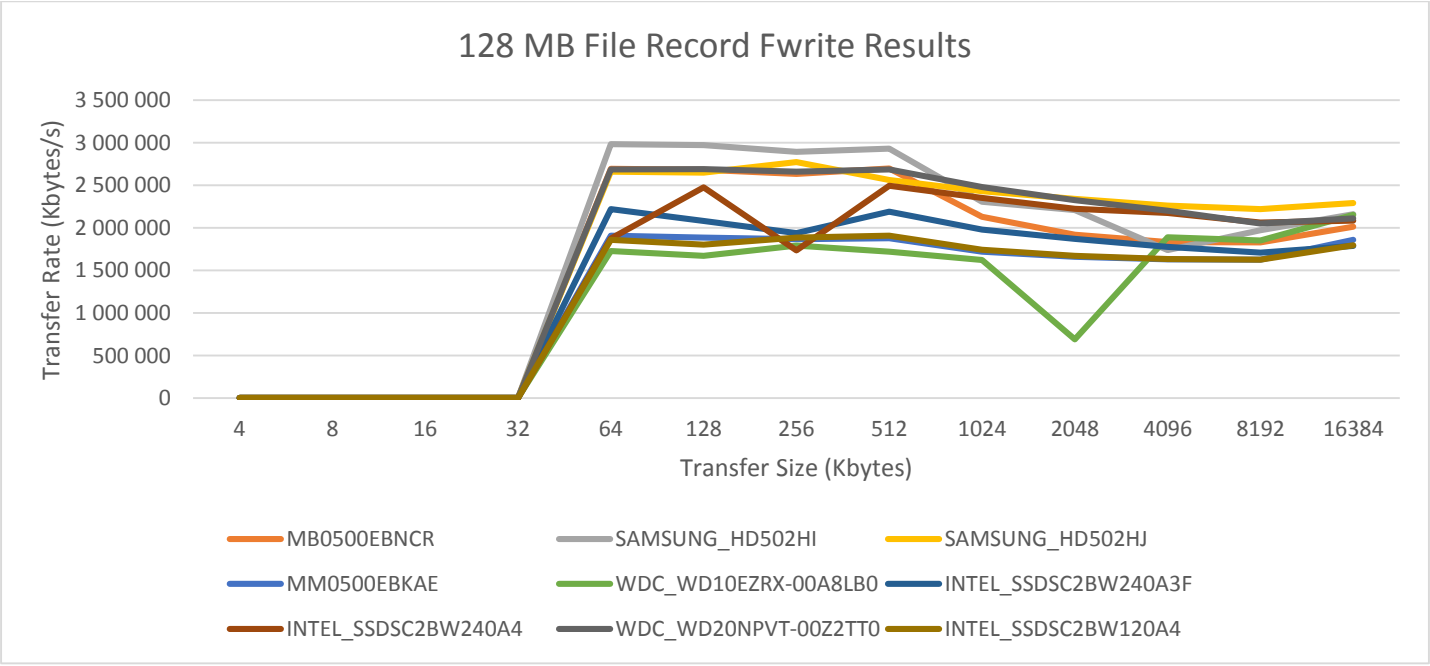
A oscilação de valores é muito acentuada neste teste.
O disco **WDC_WD10EZRX-00A8LB0** na totalidade obteve maiores resultados em 16MB e 512MB.
Para 128 MB o **WDC_WD20NPVT-00Z2TT0** foi o melhor.

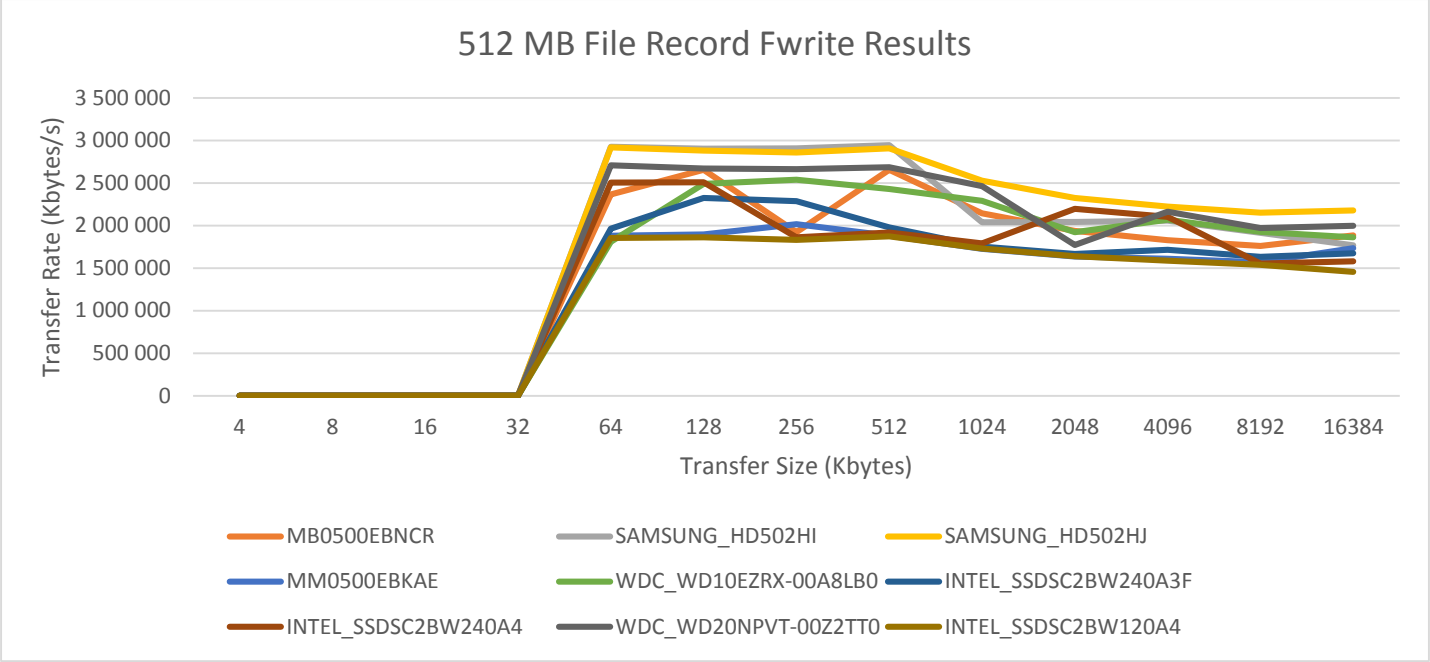
Fwrite

Este teste utiliza como operação principal a `fwrite()`. É uma rotina do sistema que permite fazer escritas com buffers, estando esse buffer em endereçamento do utilizador. Se a aplicação for escrever com transferências mais pequenas então as funcionalidades de buffer e bloqueio de E/S do *fwrite* podem aumentar a performance da aplicação reduzindo o número de chamadas ao sistema e aumentando o tamanho das transferências quando essas chamadas são feitas. O teste inclui a escrita de um ficheiro novo por isso os metadados são contabilizados.

Com os valores obtidos retirei 3 gráficos para 16 MB, 128 MB e 512 MB.







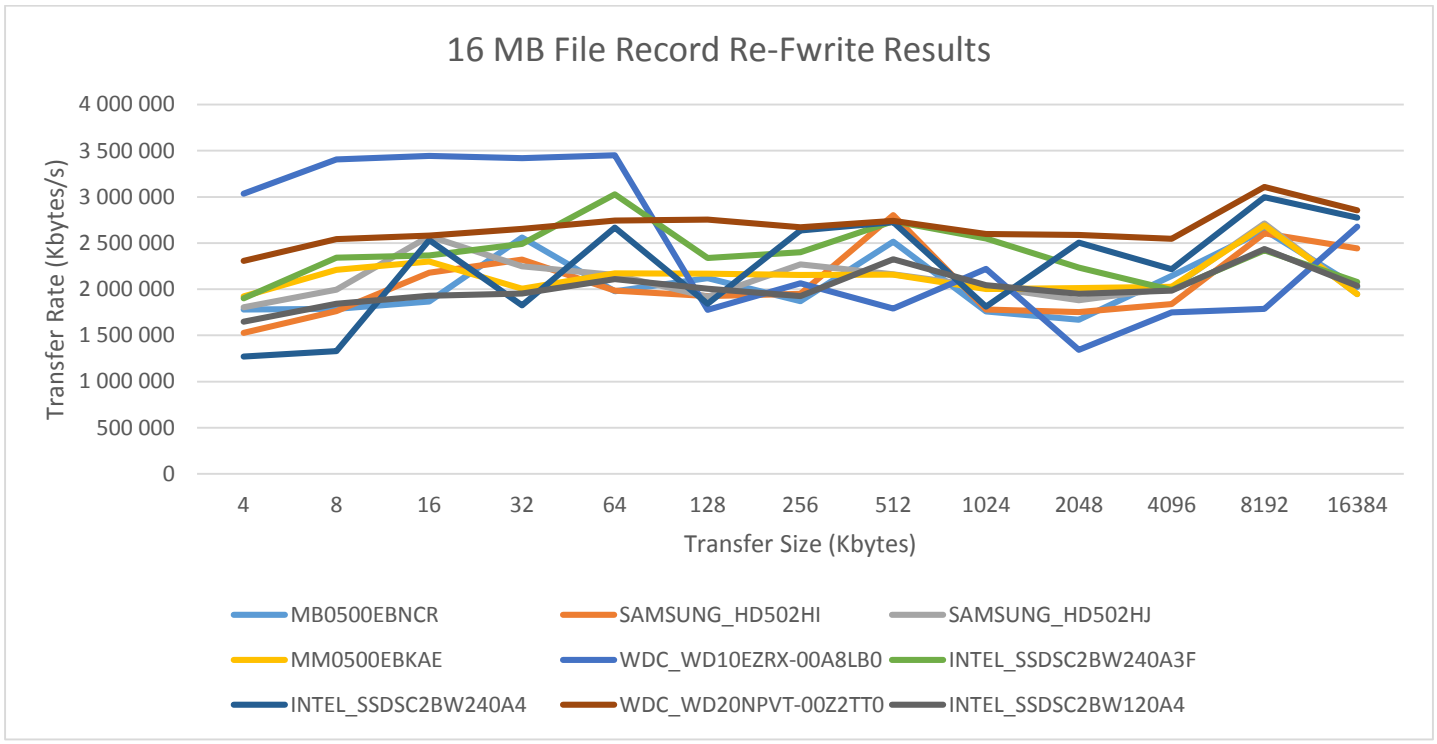
Conclusão

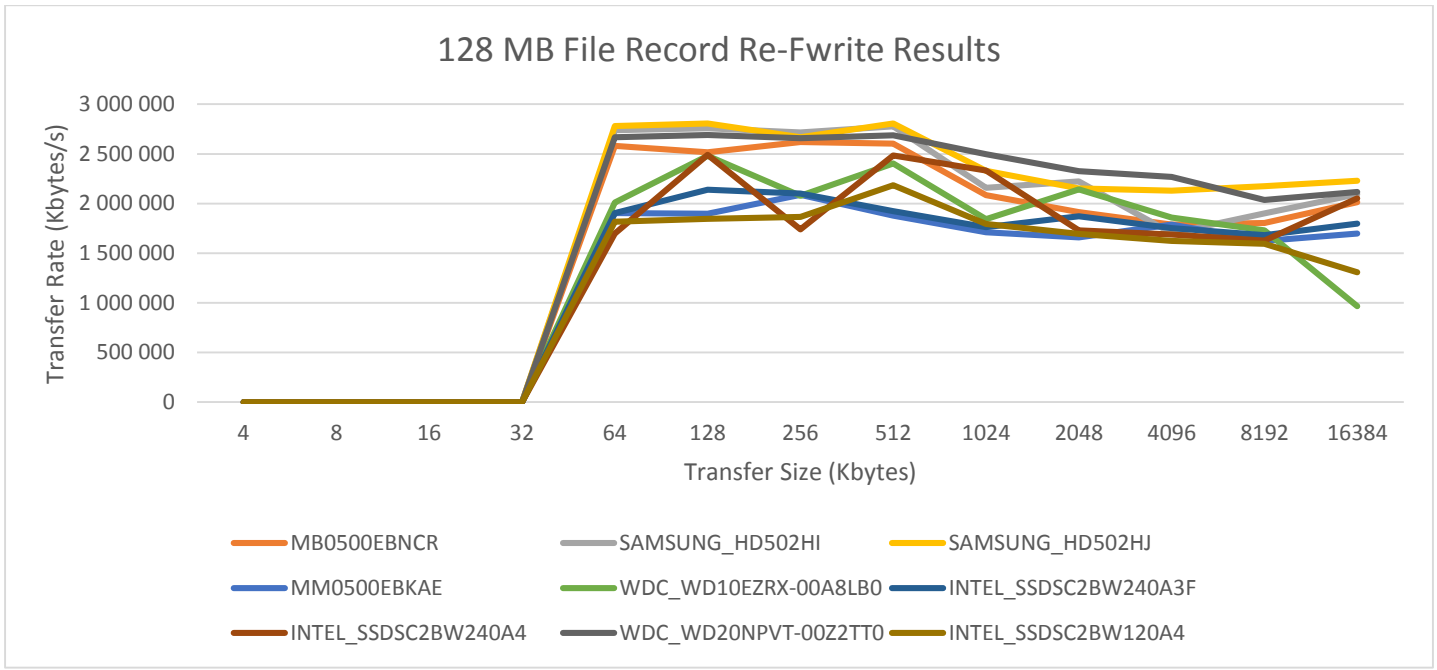
O disco **WDC_WD20NPVT-00Z2TT0** na totalidade obteve maiores resultados em 16MB.
Para os restantes tamanhos de teste o **SAMSUNG_HD502HJ** aparece como pioneiro em resultados.

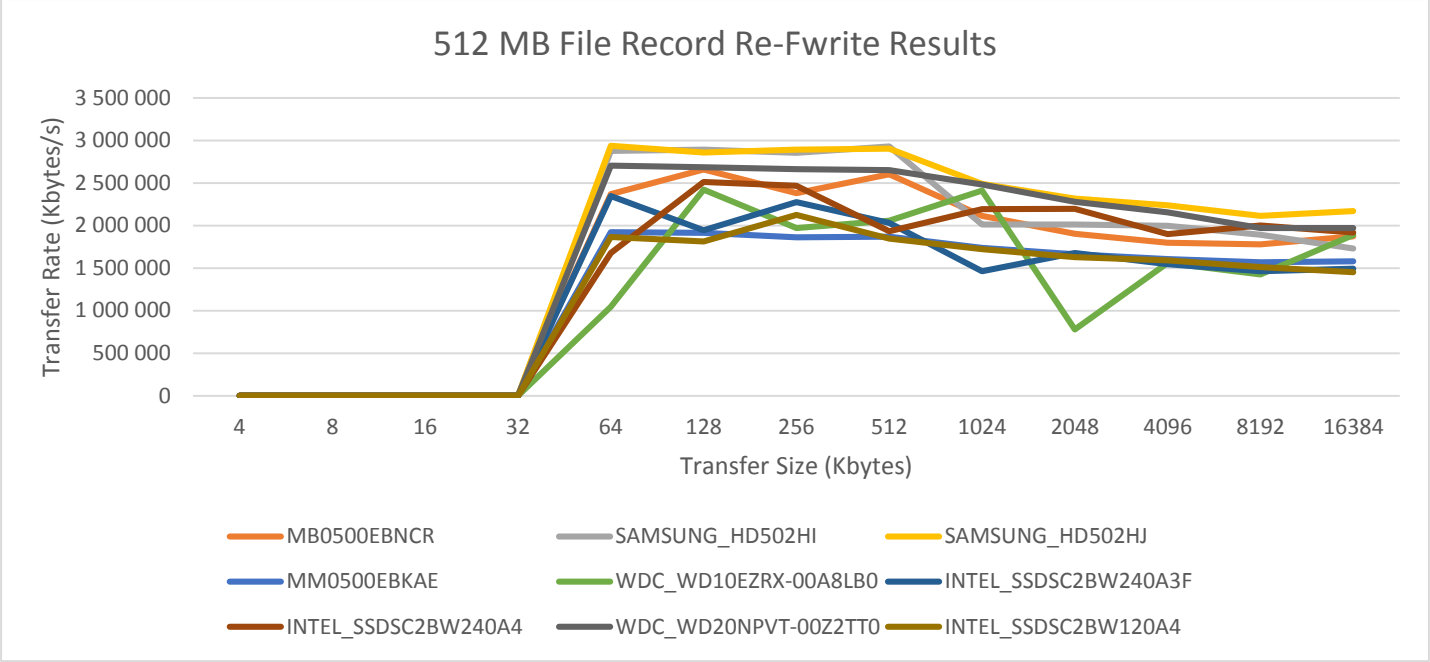
Re-Fwrite

Como o teste anterior, o *Re-Fwrite* utiliza o *fwrite* só que desta vez é escrever um ficheiro já existente por isso não há metadados envolvidos proporcionando teoricamente melhores resultados.

Com os valores obtidos retirei 3 gráficos para 16 MB, 128 MB e 512 MB.







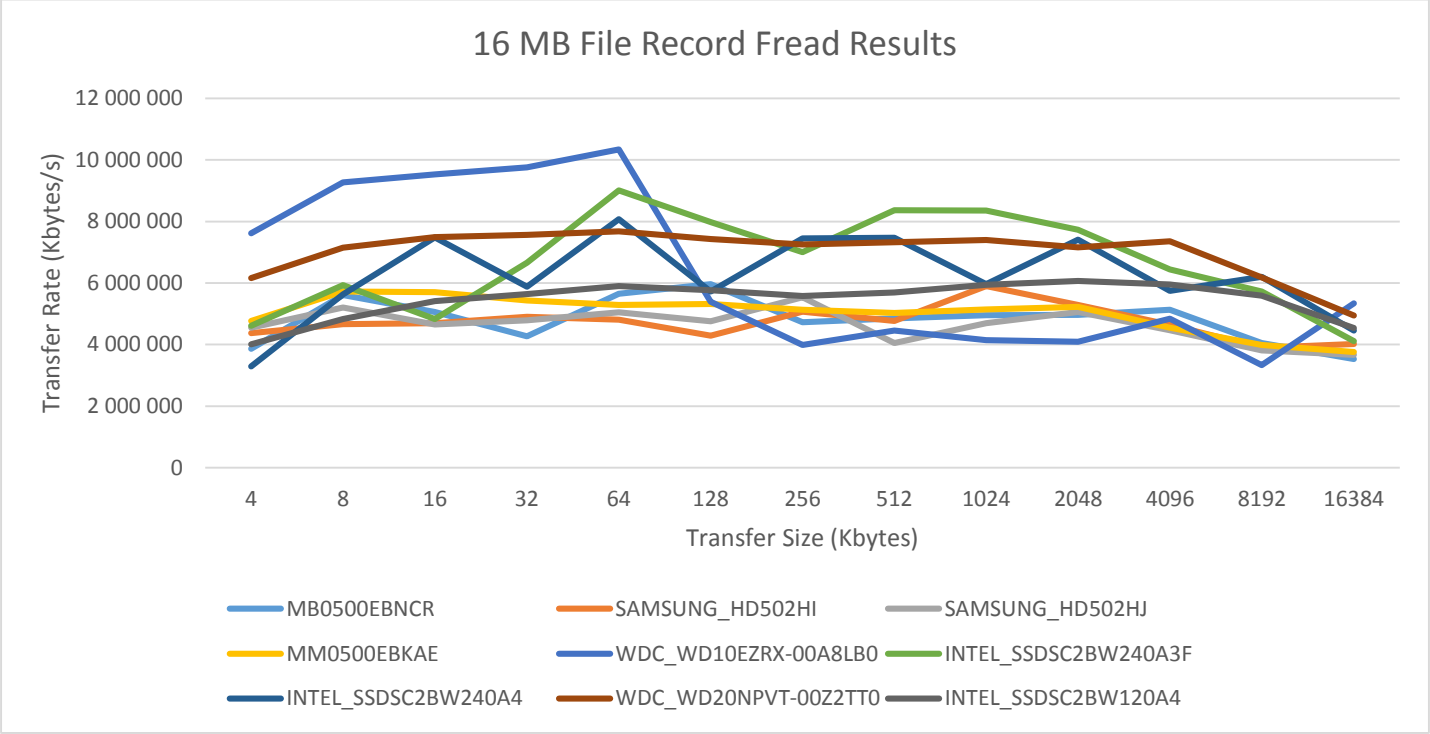
Conclusão

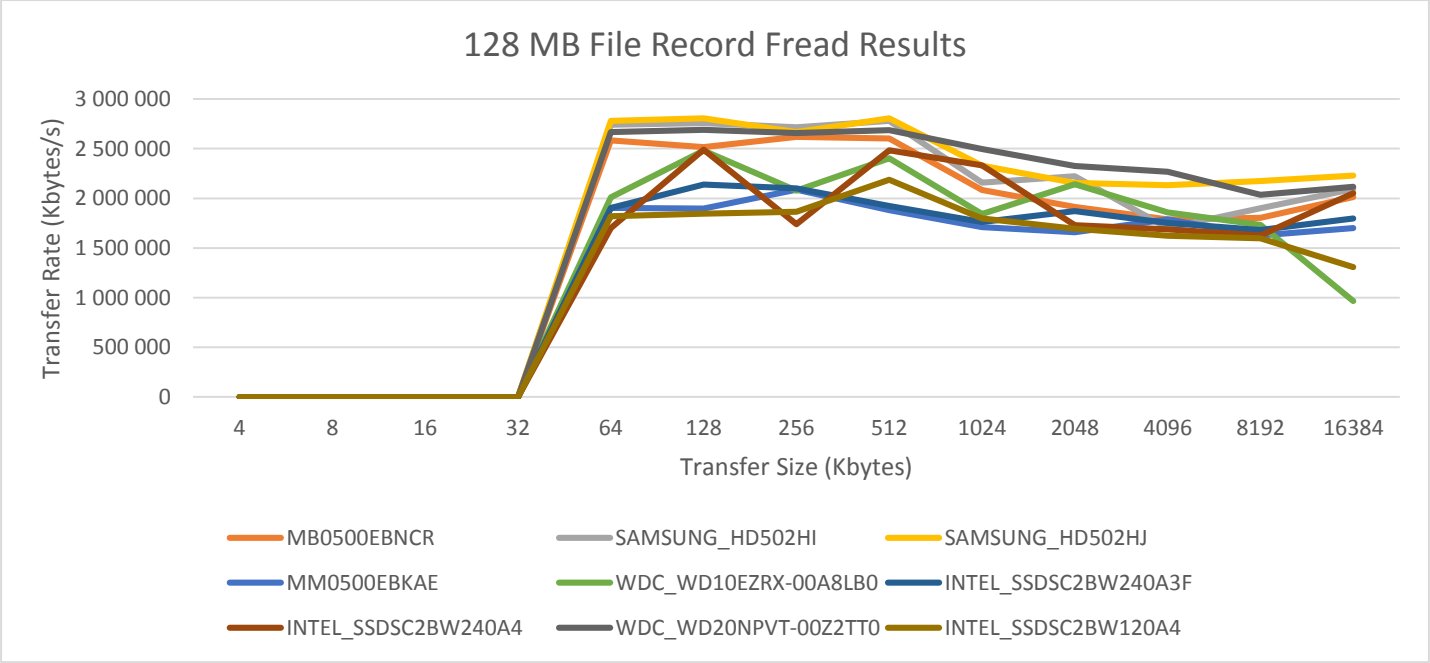
O disco **WDC_WD20NPVT-00Z2TT0** na totalidade obteve melhores resultados em 16MB e 128 MB.

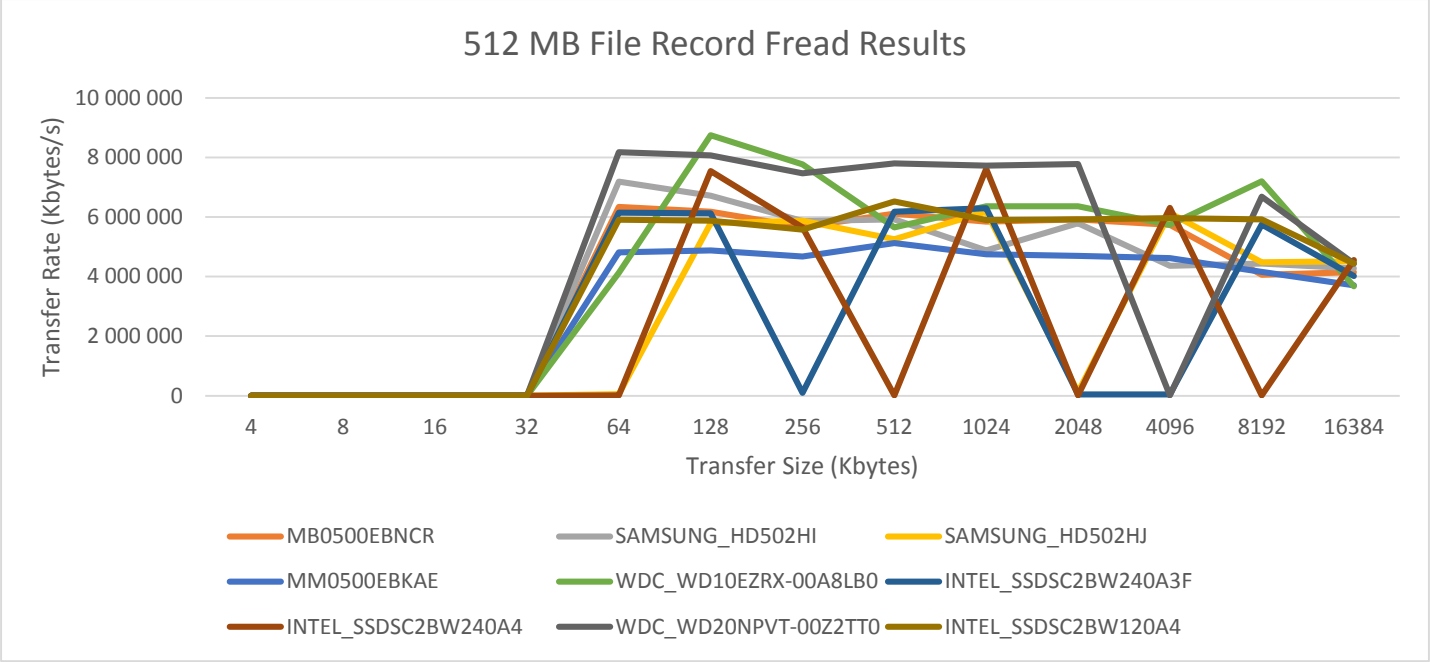
Já para 512MB o **SAMSUNG_HD502HJ** foi ligeiramente melhor que o vencedor dos 2 tamanhos anteriores.

Fread

Este teste utiliza a função `fread` que utiliza operações com buffer e bloqueios.
Com os valores obtidos retirei 3 gráficos para 16 MB, 128 MB e 512 MB.







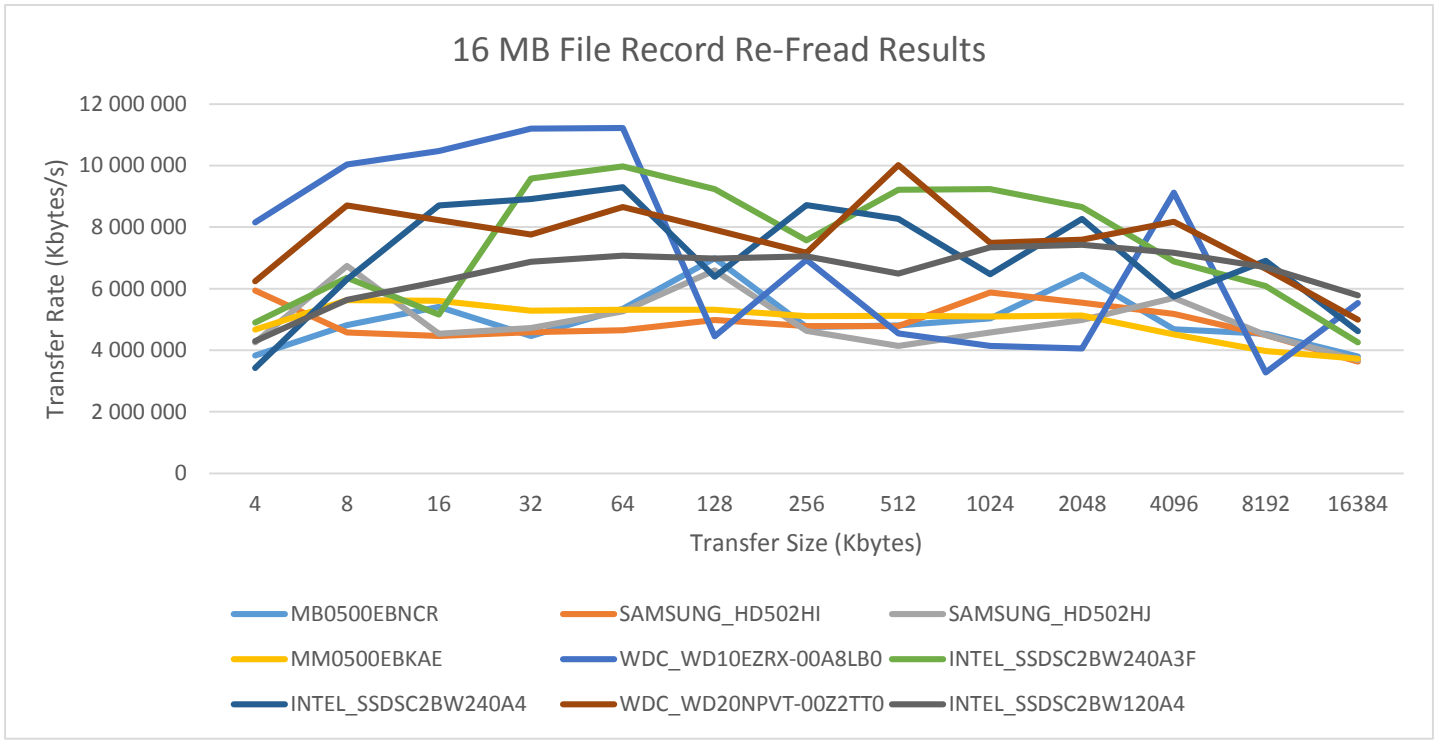
Conclusão

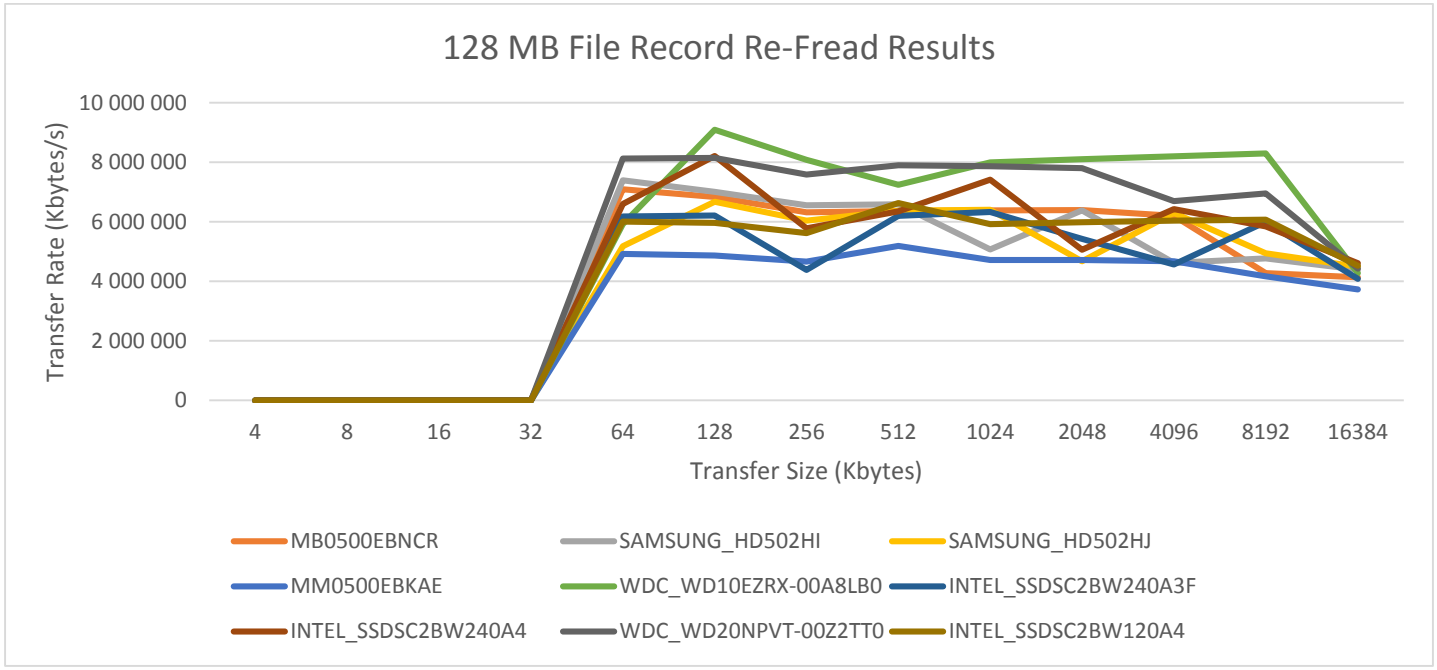
O disco **WDC_WD20NPVT-00Z2TT0** analiticamente foi superior em 16MB e 128 MB, o **SAMSUNG_HD502HJ** esteve com resultados muito próximos mas só foi efetivamente superior ao WDC para 512 MB.

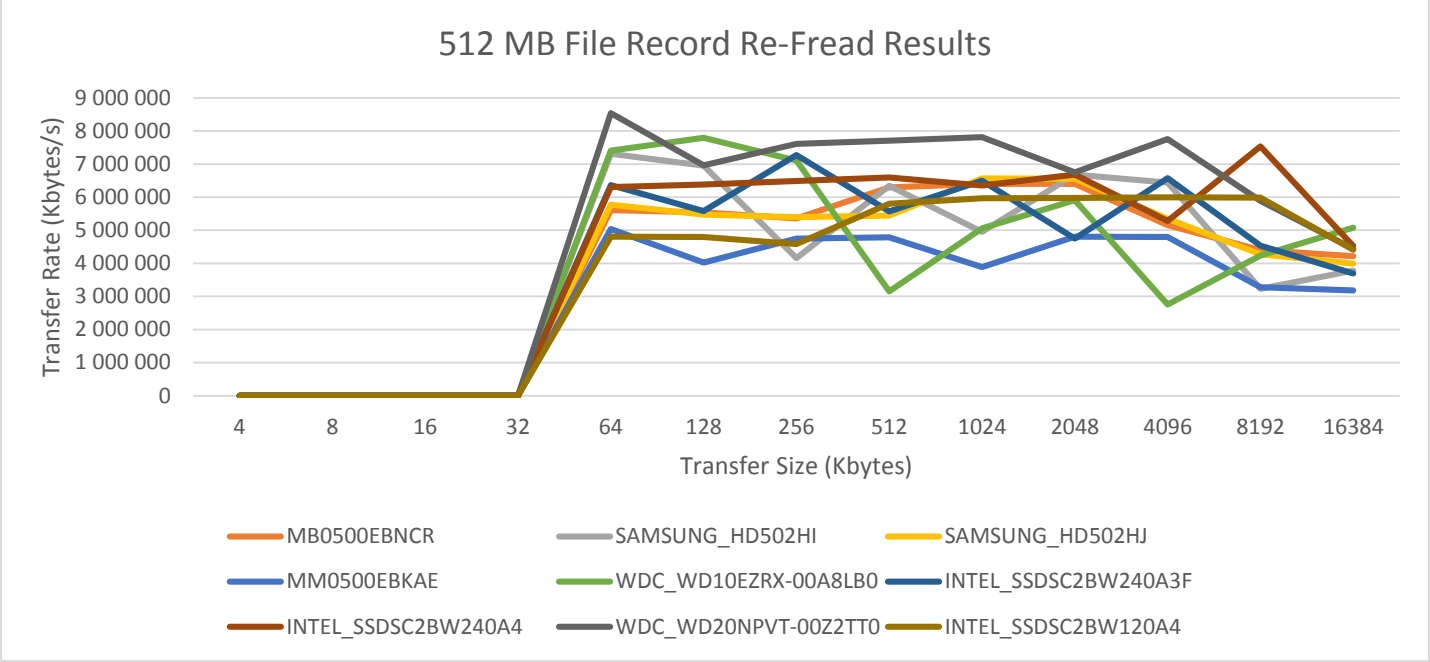
Re-Fread

Tal como o teste anterior a operação `fread` é utlizada, mas agora é lendo um ficheiro já lido previamente esperando que já estejam dados do ficheiro em cache.

Com os valores obtidos retirei 3 gráficos para 16 MB, 128 MB e 512 MB.







Conclusão

O disco **WDC_WD20NPVT-00Z2TT0** foi superior em 16MB e 128 MB, já o **WDC_WD10EZRX-00A8LB0** apenas o superou nos 512 MB.

Scripts, Diretorias e Gráficos

Cada nó tem uma pasta respetiva, contendo os ficheiros output do IOzone em .txt, ficheiros excel em .xls e ainda os gráficos gerados separados por pasta 0 (até 512MB) e 8 (até 8GB).

|-- 431-3 | |-- 0 | `-- 8 |-- 431-5 | |-- 0 | `-- 8 |-- 431-6 | |-- 0 | `-- 8	|-- 432-1 | |-- 0 | `-- 8 |-- 541-1 | |-- 0 | `-- 8 |-- 641-19 | |-- 0 | `-- 8	|-- 641-8 | |-- 0 | `-- 8 |-- 652-1 | |-- 0 | `-- 8 |-- 662-6 | |-- 0 | `-- 8

Os Gráficos foram gerados graças à ferramenta disponibilizada intitulada de *Generate_Graphs* que utiliza o *gnuplot*. Os gráficos individuais não foram inseridos no relatório pois são muitos (117 gráficos só para o teste de 512 MB !), portanto estão nas pastas para consulta posterior.

Os testes de 8GB não terminaram como previsto para todos os nós portanto não pude utilizar esses valores para análise.

Análise Final de Resultados e Conclusão

O teste do IOzone é muito intensivo operacionalmente e demorou várias horas para concluir os testes. Após foi necessário analisar os valores e gerar gráficos comparativos para todos os discos analisados.

O disco que obteve melhores resultados foi claramente o **WDC_WD20NPVT-00Z2TT0** pertencente aos nós 652-1 e 652-2. Estava à espera que um SSD tivesse os resultados com clara liderança mas tal não aconteceu, apenas obtendo 2 testes vitoriosos com SSD’s diferentes.

Análise de Traçados - Introdução

O *strace* é uma ferramenta para a depuração de programas cujos traçados quando filtrados e analisados podem também ser usados para estudar o padrão de execução das aplicações. Monitoriza interações entre os programas e o kernel do Linux, como chamadas ao sistema, sinais e mudanças no estado do processo. O trabalho do *strace* só é possível devido ao *ptrace*.

Com a ajuda do ficheiro *refazStFd.py* disponibilizado pelo professor foi possível obter melhores informações sobre a utilização dos vários ficheiros temporários criados pelo *iozone*.

Strace e Parâmetros de teste

Para este trabalho utilizei o `strace` versão `4.5.19` instalada no cluster. Primeiramente foi necessário utilizar o *strace* da seguinte forma num job sobre o iozone com 256 MB:

```
strace -T -ttt -o strace.out /opt/iozone/bin/iozone –R -a -i0 -i1 -i2 -i5 -g 256M
```

Com o ficheiro output *strace.out* passei-o pelo *refazStFd.py* para tratar dos ficheiros temporários.

```
/share/apps/IOAPPS/refazStFd.py < strace.out > ref.rsf
```

O *ref.rsf* já está tratado e então é altura de o passar pelo *strace_analyzer* para gerar algumas estatísticas para melhor compreender o que se passou naquela execução do *iozone*.

```
/share/apps/IOAPPS/strace_analyzer_ng_0.09.pl ref.rsf > stanREF.txt
```

Estatísticas

Tendo já o ficheiro pronto para análise, é possível obter as seguintes informações.

Tempo

O programa na totalidade demorou cerca de 191.96 segundos a concluir, sendo que 83.65% desse tempo foi em operações de E/S ao sistema perfazendo 160.587 segundos.

Elapsed Time for run	***191.959035 (secs)***
Total IO Time	*160.587606 (secs)*
Total IO Time Counter	*324461*
Percentage of Total Time	*83.657227%*

Total de Operações E/S

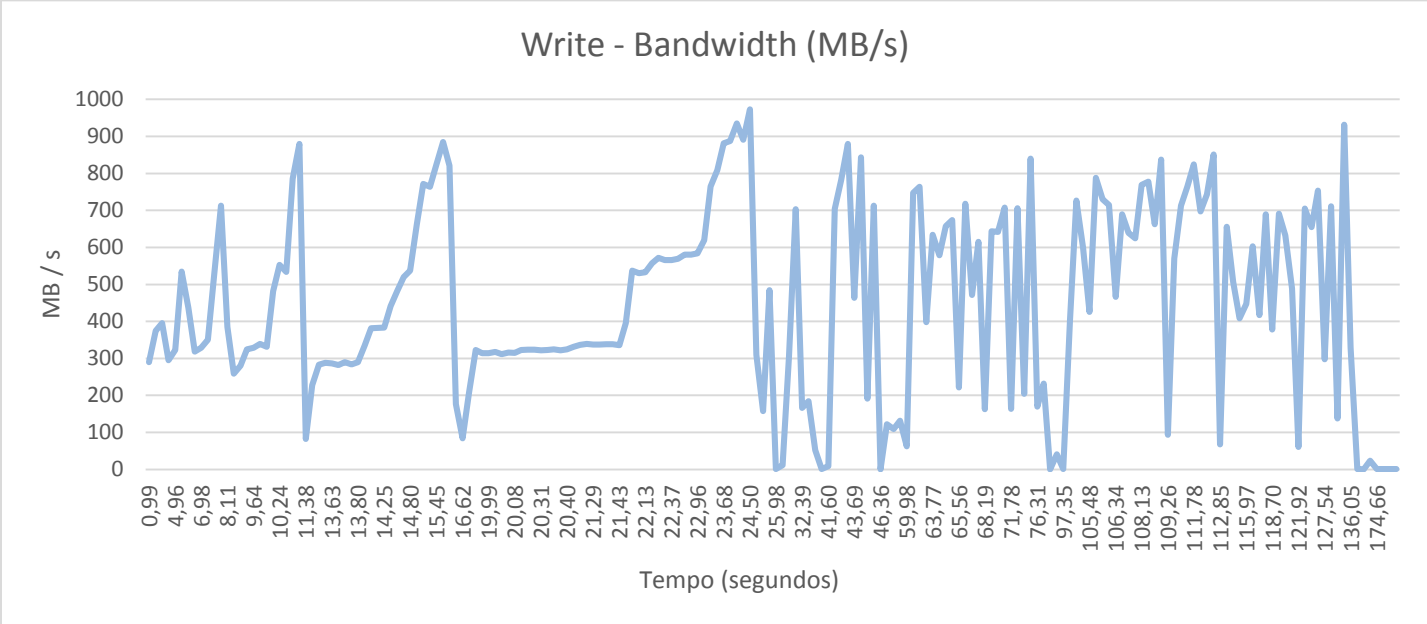
As seguintes chamadas ao sistema via operações de E/S foram contabilizadas da seguinte forma:

Command	***Count***
access	*1*
lseek	*95254*
stat	*404*
unlink	*234*
open	*941*
close	*1175*
creat	*117*
fstat	*6*
fsync	*1404*
read	*126931*
write	*97920*

Como se trata de um teste de escrita em disco, é normal que os comandos *lseek*, *read* e *write* sejam os mais utilizados.

Banda utilizada pelo write

Com o histórico da chamada ao sistema write, foi possível gerar o seguinte gráfico onde podemos analisar a quantidade de informação escrita por segundo (*Banda*/*Bandwidth*) ao longo da execução. Os dados obtidos resultaram em 97921 linhas, resultando num gráfico ilegível, portanto fiz uma média a cada 512 linhas para ficar à volta dos 191 segundos, tempo de duração da execução. Ficando assim um gráfico mais percetível e de melhor interpretação.



Tamanho de Blocos de Ficheiros no write

A tabela seguinte representa a dispersão entre os tamanhos de blocos utilizados nas chamadas ao sistema em operações de escrita.

IO Size Range	**Number of syscalls**
0KB < < 1KB	2901
1KB < < 8KB	24528
8KB < < 32KB	18396
32KB < < 128KB	27639
128KB < < 256KB	12285
256KB < < 512KB	6141
512KB < < 1000KB	3069
1000KB < < 10MB	2868
10MB < < 100MB	93
100MB < < 1GB	0
1GB < < 10GB	0
10GB < < 100GB	0
100GB < < 1TB	0
1TB < < 10TB	0

Resumo do write

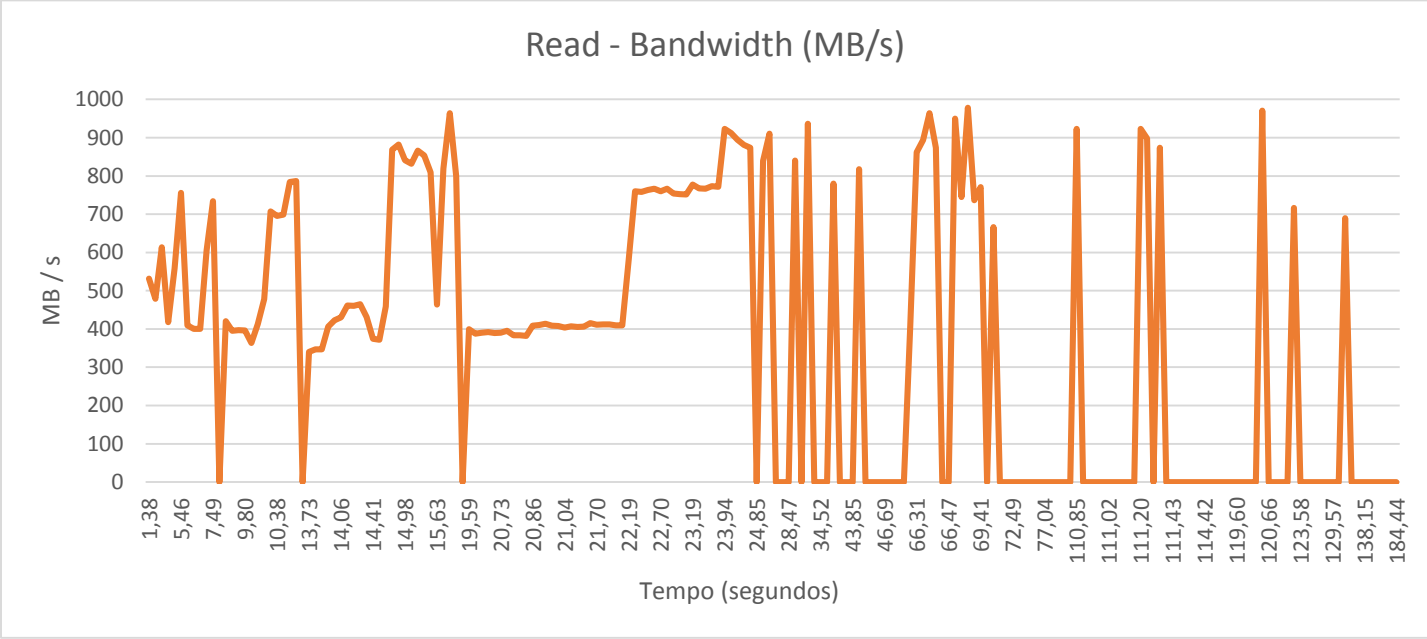
Concluídas as análises anteriores à chamada *write* é possível tirar uma conclusão geral e algumas estatísticas interessantes com os valores obtidos.

-- WRITE SUMMARY --	
Total number of Bytes written	14,796,940,713 (14,796.940713 MB)
Number of Write syscalls	97920
Average (mean) Bytes per syscall	151,112.548131127 (bytes) (0.151112548131127 MB)
Standard Deviation	718,723.650115704 (bytes) (0.718723650115704 MB)
Mean Absolute Deviation	686,634.90476273 (bytes) (0.68663490476273 MB)
Median Bytes per syscall	65,536 (bytes) (0.065536 MB)
Median Absolute Deviation	142,811.593045343 (bytes) (0.142811593045343 MB)
Time for slowest write syscall (secs)	0.164958
Line location in file	304974
Smallest write syscall size	1 (Byte)
Largest write syscall size	16777216 (Bytes)

No total foram escritos 14 796.9 MB (+/- 14 GB), num total de 97920 chamadas de escrita (como o *write*), como mostra a primeira estatística (*Total de Operações E/S*). A Mediana de Bytes por chamada ao sistema está nos 65536, valor esse presente nos blocos de teste do *iozone*, não sendo um valor estranho às estatísticas.

Banda utilizada pelo read

Com o histórico da chamada ao sistema read, foi possível gerar o seguinte gráfico onde podemos analisar a quantidade de informação lida por segundo (*Banda*/*Bandwidth*) ao longo da execução. Tal como no *Write*, o gráfico gerado diretamente a partir dos dados ficaria gigantesco, portanto fiz uma média a cada 650 linhas para ficar à volta dos 191 segundos, tempo de duração da execução. Ficando assim um gráfico mais percetível e de melhor interpretação.



Tamanho de Blocos de Ficheiros no read

A tabela seguinte representa a dispersão entre os tamanhos de blocos utilizados nas chamadas ao sistema em operações de escrita.

IO Size Range	**Number of syscalls**
0KB < < 1KB	3
1KB < < 8KB	32724
8KB < < 32KB	24564
32KB < < 128KB	36896
128KB < < 256KB	16404
256KB < < 512KB	8210
512KB < < 1000KB	4112
1000KB < < 10MB	3884
10MB < < 100MB	134
100MB < < 1GB	0
1GB < < 10GB	0
10GB < < 100GB	0
100GB < < 1TB	0
1TB < < 10TB	0

Resumo do read

Concluídas as análises anteriores ao *read* é possível tirar uma conclusão geral e algumas estatísticas interessantes com os valores obtidos.

-- READ SUMMARY --	
Total number of Bytes read	20,131,020.718 (20,131.020718 MB)
Number of Read syscalls	126,931
Average (mean) Bytes per syscall	158,598.141651764 (bytes) (0.158598141651764 MB)
Standard Deviation	749,136.288122469 (bytes) (0.749136288122469 MB)
Mean Absolute Deviation	716,227.68995956 (bytes) (0.71622768995956 MB)
Median Bytes per syscall	65,536 (bytes) (0.065536 MB)
Median Absolute Deviation	148,001,871520747 (bytes) (0.148001871520747 MB)
Time for slowest read syscall (secs)	0.006456
Line location in file	199947
Smallest read syscall size	832 (Bytes)
Largest read syscall size	16777216 (Bytes)

No total foram lidos 20 131 MB (+/- 20 GB), num total de 126931 chamadas de leitura (como o *read*), como mostra a primeira estatística (*Total de Operações E/S*). A Mediana de Bytes por chamada ao sistema está nos 65536, valor esse presente nos blocos de teste do *iozone*, não sendo um valor estranho às estatísticas.

Conclusão

Com a ferramenta *strace* foi possível ter uma perspetiva diferente de como o *iozone* trabalha. Analisando de um ponto de vista de chamadas ao sistema, *syscalls*, como *lseek*, *write* ou *read*. Podendo assim obter melhores informações do seu funcionamento e até detetar erros ou anomalias.

Análise de Desempenho - Introdução

O *perf* é uma ferramenta para a avaliação de desempenho de programas, permite analisar os contadores do sistema: Software, Hardware, Tracepoint e Dynamic Probes. Disponível no Linux, analisando espaço de utilizador (código) e kernel.

naive.c - Multiplicação de Matrizes

O programa é muito básico, tem 2 funções principais, a de inicialização da matriz, `initialize_matrices` e a da multiplicação, `multiply_matrices`.

```
#define MAX_MSIZE 1000
#define MSIZE     500

void initialize_matrices()
{
  int i, j ;

  for (i = 0 ; i < MSIZE ; i++) {
    for (j = 0 ; j < MSIZE ; j++) {
      matrix_a[i][j] = (float) rand() / RAND_MAX ;
      matrix_b[i][j] = (float) rand() / RAND_MAX ;
             matrix_r[i][j] = 0.0 ;
    }
  }
}

void multiply_matrices()
{
  int i, j, k ;

  for (i = 0 ; i < MSIZE ; i++) {
    for (j = 0 ; j < MSIZE ; j++) {
      float sum = 0.0 ;
      for (k = 0 ; k < MSIZE ; k++) {
        sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
      }
      matrix_r[i][j] = sum ;
    }
  }
```

Eventos suportados

Utilizei o nó `compute-321-6` e obtive os seguintes eventos disponíveis (apenas os mais relevantes):

List of pre-defined events:	
cpu-cycles OR cycles	[Hardware event]
instructions	[Hardware event]
cache-references	[Hardware event]
cache-misses	[Hardware event]
branch-instructions OR branches	[Hardware event]
branch-misses	[Hardware event]
stalled-cycles-frontend OR idle-cycles-frontend	[Hardware event]
stalled-cycles-backend OR idle-cycles-backend	[Hardware event]
cpu-clock	[Software event]
task-clock	[Software event]
page-faults OR faults	[Software event]
context-switches OR cs	[Software event]
cpu-migrations OR migrations	[Software event]
minor-faults	[Software event]
major-faults	[Software event]
alignment-faults	[Software event]
emulation-faults	[Software event]
L1-dcache-loads	[Hardware cache event]
L1-dcache-load-misses	[Hardware cache event]
L1-dcache-stores	[Hardware cache event]
L1-dcache-store-misses	[Hardware cache event]
L1-dcache-prefetches	[Hardware cache event]
L1-dcache-prefetch-misses	[Hardware cache event]
L1-icache-loads	[Hardware cache event]
L1-icache-load-misses	[Hardware cache event]
LLC-loads	[Hardware cache event]
LLC-load-misses	[Hardware cache event]
LLC-stores	[Hardware cache event]
LLC-store-misses	[Hardware cache event]
LLC-prefetches	[Hardware cache event]
LLC-prefetch-misses	[Hardware cache event]
dTLB-loads	[Hardware cache event]
dTLB-load-misses	[Hardware cache event]
dTLB-stores	[Hardware cache event]
dTLB-store-misses	[Hardware cache event]
iTLB-loads	[Hardware cache event]
iTLB-load-misses	[Hardware cache event]
branch-loads	[Hardware cache event]
branch-load-misses	[Hardware cache event]

Eventos medidos

Com o perf stat obtive os seguintes valores para os Contadores dos Eventos. De notar o baixo número de branch-misses pois o duplo ciclo for da multiplicação é fácil de prever pois é muito normal o ciclo não terminar, só termina no fim de toda a iteração da linha.

Performance counter stats for './naive':

task-clock (msec)	104.835969	0.966 CPUs utilized
context-switches	40	0.382 K/sec
cpu-migrations	0	0.000 K/sec
page-faults	985	0.009 M/sec
cycles	259879909	2.479 GHz [49.32%]
stalled-cycles-frontend	<not supported>	
stalled-cycles-backend	<not supported>	
instructions	347172524	1.34 insns per cycle [75.43%]
branches	40230769	383.750 M/sec [75.38%]
branch-misses	99269	0.25% of all branches [75.32%]

Perfis de Execução

Como esperado a invocação da multiplicação das matrizes foi a mais utilizada pelo sistema, seguindo-se o *random* utilizado na inicialização.

Overhead	Command	Shared	Object	Symbol
84,47%	naive	naive	[,]	multiply_matrices()
3,25%	naive	libc-2,12,so	[,]	__random
2,43%	naive	naive	[,]	initialize_matrices()
1,19%	naive	[kernel,kallsyms]	[k]	0xffffffff8127d387
1,12%	naive	[kernel,kallsyms]	[k]	0xffffffff81189f41
0,88%	naive	ld-2,12,so	[,]	open64
0,79%	naive	ld-2,12,so	[,]	strcmp
0,72%	naive	libc-2,12,so	[,]	__random_r
0,68%	naive	[kernel,kallsyms]	[k]	0xffffffff8118c7e7
0,55%	naive	naive	[,]	rand@plt
0,48%	naive	[kernel,kallsyms]	[k]	0xffffffff8104e431
0,46%	naive	[kernel,kallsyms]	[k]	0xffffffff810a097d
0,38%	naive	[kernel,kallsyms]	[k]	0xffffffff81184dd0
0,35%	naive	[kernel,kallsyms]	[k]	0xffffffff8116bc74
0,33%	naive	[kernel,kallsyms]	[k]	0xffffffff81062e5d
0,32%	naive	[kernel,kallsyms]	[k]	0xffffffff812833f3
0,32%	naive	[kernel,kallsyms]	[k]	0xffffffff8116d4e5
0,32%	naive	ld-2,12,so	[,]	_dl_fini
0,31%	naive	[kernel,kallsyms]	[k]	0xffffffff8112b78a
0,31%	naive	[kernel,kallsyms]	[k]	0xffffffff8110872a
0,29%	naive	[kernel,kallsyms]	[k]	0xffffffff812834cc
0,02%	naive	[kernel,kallsyms]	[k]	0xffffffff8103891a
0,02%	naive	[kernel,kallsyms]	[k]	0xffffffff8103891c

Para 313 amostras de eventos ‘cycles’, Event count (approx,): 190099749

Annotate

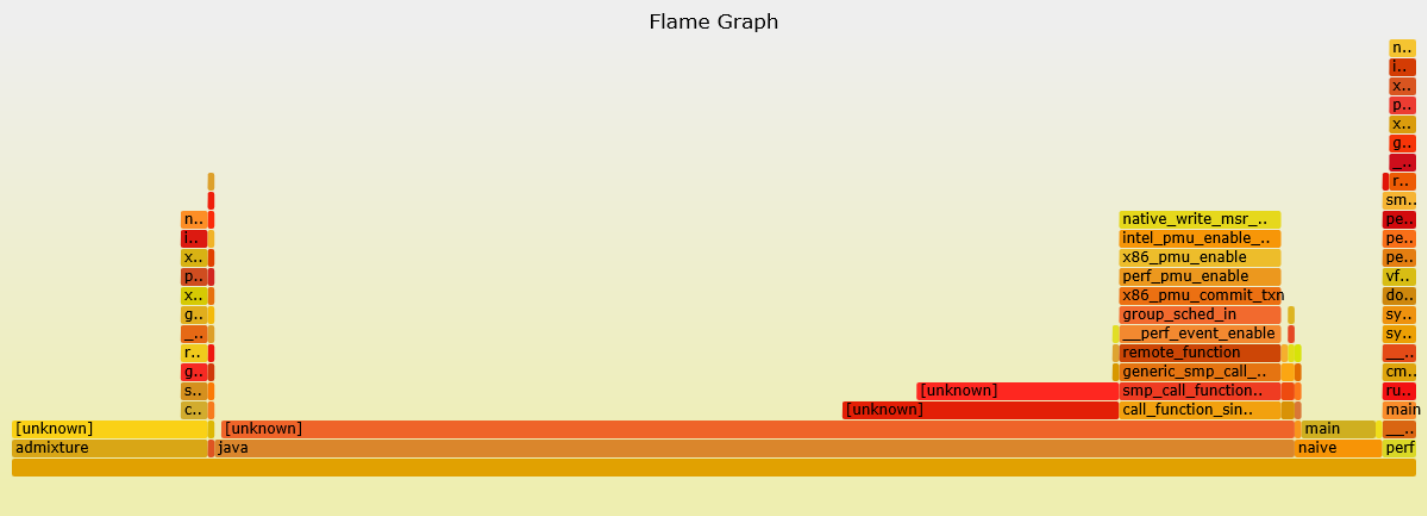
Com a ajuda do perf annotate foi possível extrair utilização de chamadas em assembly do código gerado. A multiplicação e o controlo do ciclo é a parte mais pesada da função `multiply_matrices`, função essa como vista anteriormente é a mais utilizada por isso digna de uma melhor análise.

```
Percent |      Source code & Disassembly of naive for cycles
------------------------------------------------------------
        :      void multiply_matrices()
        :      {
        :        int i, j, k ;
        :        for (i = 0 ; i < MSIZE ; i++) {
   0.00 :        400a06:       xor    %ebx,%ebx
   0.00 :        400a08:       nopl   0x0(%rax,%rax,1)
   0.00 :        400a10:       xorps  %xmm1,%xmm1
   0.00 :        400a13:       lea    0x6f5540(%rbx),%rcx
   0.00 :        400a1a:       mov    %rsi,%rdx
   0.00 :        400a1d:       xor    %eax,%eax
   0.00 :        400a1f:       nop
        :        for (i = 0 ; i < MSIZE ; i++) {
        :          for (j = 0 ; j < MSIZE ; j++) {
        :            float sum = 0.0 ;
        :            for (k = 0 ; k < MSIZE ; k++) {
        :              sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
  10.26 :        400a20:       movaps %xmm2,%xmm0
   0.00 :        400a23:       movlps (%rdx),%xmm0
  13.59 :        400a26:       movhps 0x8(%rdx),%xmm0
  29.74 :        400a2a:       add    $0x4,%rdx
   0.00 :        400a2e:       shufps $0x0,%xmm0,%xmm0
   2.31 :        400a32:       mulps  (%rcx,%rax,1),%xmm0
  30.26 :        400a36:       add    $0x7d0,%rax
   0.00 :        400a3c:       cmp    $0xf4240,%rax
   0.00 :        400a42:       addps  %xmm0,%xmm1
  12.56 :        400a45:       jne    400a20 <main+0xd0>
        :            }
        :            matrix_r[i][j] = sum ;
   0.00 :        400a47:       addps  %xmm2,%xmm1
   0.00 :        400a4a:       movaps %xmm1,(%rdi,%rbx,1)
   0.77 :        400a4e:       add    $0x10,%rbx
   0.00 :        400a52:       cmp    $0x7d0,%rbx
   0.00 :        400a59:       jne    400a10 <main+0xc0>
   0.00 :        400a5b:       add    $0x7d0,%rsi
   0.00 :        400a62:       add    $0x7d0,%rdi
```

Flame Graphs

Estes gráficos são a visualização de um software analisado, permitindo que os *code-paths* mais frequentes sejam identificados mais rapidamente. Podem ser gerados usando a distribuição *open-source* que gera gráficos em *SVG*.

No canto inferior direito do gráfico podemos encontrar a execução do *naive*. Não consegui isolar a execução do *naive* para podermos ver as suas invocações, portanto seria semelhante ao Perfil de Execução.



Conclusão

Com a ferramenta *perf* foi possível ter uma análise de execução de um código simples, multiplicação de matrizes. Com o Flame Graphs ter uma ideia visual da distribuição temporal das execuções naquela altura no sistema Linux do nó.