http://blogs.technet.com/b/josebda/archive/2014/10/13/diskspd-powershell-and-storage-performance-measuring-iops-throughput-and-latency-for-both-local-disks-and-smb-file-shares.aspx

• Sign in Server & Tools Blogs > Server & Management Blogs > Jose Barreto's Blog• All About Windows Server • Cloud Platform Blogs • Datacenter Management • Client Management • Virtualization, VDI & Remote Desktop • File & Storage & High Availability • Windows Server Management • Identity & Access

Jose Barreto's Blog A blog by Jose Barreto, a member of the File Server team at Microsoft. • Search this blogSearch all blogs• Subscribe • Comments • Contact Menu• Home • Atom Traduci questa paginaAraboBosniaco (alfabeto latino)BulgaroCatalanoCecoCinese semplificatoCinese tradizionaleCoreanoCroatoDaneseEbraicoEstoneFarsiFinlandeseFranceseGalleseGiapponeseGrecoHaitianoHindiHmong DawIndonesianoItalianoKlingonLettoneLituanoMaleseMalteseNorvegeseOlandesePolaccoPortogheseQuerétaro OtomiRumenoRussoSerbo (alfabeto cirillico)Serbo (alfabeto latino)SlovaccoSlovenoSpagnoloSvedeseTailandeseTedescoTurcoUcrainoUngheresUrduVietnamitaYucatec MayaMicrosoft® TranslatorRecent Posts• Drive Performance Report Generator - PowerShell script using DiskSpd by Arnaud TorresPosted 5 days ago 2Comments • Using PowerShell and Excel PivotTables to understand the files on your diskPosted 1 month ago 7Comments • New PowerShell cmdlets in Windows Server 2016 TP2 (compared to Windows Server 2012 R2)Posted 1 month ago 4Comments • Windows Server 2012 R2 Storage PowerShell cmdlet popularityPosted 1 month ago Tags• Clustering • Events • Hyper-V • Powershell • SMB • SMB3 • SQL Server • Storage • Windows Server • Windows Server 2008 • Windows Server 2008 R2 • Windows Server 2012 More ▼Archives• July 2015 (1) • June 2015 (1) • May 2015 (9) • April 2015 (6) • March 2015 (10) More ▼Blog - NewsAll messages posted to this blog are provided “AS IS” with no warranties, and confer no rights. Information on unreleased products are subject to change without notice. Dates related to unreleased products are estimates and are subject to change without notice. The content of this site are personal opinions and might not represent the Microsoft Corporation view. The information contained in this blog represents my view on the issues discussed as of the date of publication. You should not consider older, out-of-date posts to reflect my current thoughts and opinions. Follow @josebarreto on Twitter for updates on new blog posts.

DiskSpd, PowerShell and storage performance: measuring IOPs, throughput and latency for both local disks and SMB file shares JoseBarreto13 Oct 2014 4:57 PM• 11 1. Introduction I have been doing storage-related demos and publishing blogs with some storage performance numbers for a while, and I commonly get questions such as “How do you run these tests?” or “What tools do you use to generate IOs for your demos?”. While it’s always best to use a real workload to test storage, sometimes that is not convenient. In the past, I frequently used and recommended a free tool from Microsoft to simulate IOs called SQLIO. However, there is a better tool that was recently released by Microsoft called DiskSpd. This is a flexible tool that can simulate many different types of workloads. And you can apply it to several configurations, from a physical host or virtual machine, using all kinds of storage, including local disks, LUNs on a SAN, Storage Spaces or SMB file shares. 2. Download the tool To get started, you need to download and install the DiskSpd. You can get the tool fromhttp:*aka.ms/DiskSpd. It comes in the form of a ZIP file that you can open and copy local folder. There are actually 3 subfolders with different versions of the tool included in the ZIP file: amd64fre (for 64-bit systems), x86fre (for 32-bit systems) and armfre (for ARM systems). This allows you to run it in pretty much every Windows version, client or server. In the end, you really only need one of the versions of DiskSpd.EXE files included in the ZIP (the one that best fits your platform). If you’re using a recent version of Windows Server, you probably want the version in the amd64fre folder. In this blog post, I assume that you copied the correct version of DiskSpd.EXE to the C:\DiskSpd local folder. If you're a developer, you might also want to take a look at the source code for DiskSpd. You can find that at https://github.com/microsoft/diskspd. 3. Run the tool When you’re ready to start running DiskSpd, you want to make sure there’s nothing else running on the computer. Other running process can interfere with your results by putting additional load on the CPU, network or storage. If the disk you are using is shared in any way (like a LUN on a SAN), you want to make sure that nothing else is competing with your testing. If you’re using any form of IP storage (iSCSI LUN, SMB file share), you want to make sure that you’re not running on a network congested with other kinds of traffic. WARNING: You could be generating a whole lot of disk IO, network traffic and/or CPU load when you run DiskSpd. If you’re in a shared environment, you might want to talk to your administrator and ask permission. This could generate a whole lot of load and disturb anyone else using other VMs in the same host, other LUNs on the same SAN or other traffic on the same network. WARNING: If you use DiskSpd to write data to a physical disk, you might destroy the data on that disk. DiskSpd does not ask for confirmation. It assumes you know what you are doing. Be careful when using physical disks (as opposed to files) with DiskSpd. NOTE: You should run DiskSpd from an elevated command prompt. This will make sure file creation is fast. Otherwise, DiskSpd will fall back to a slower method of creating files. In the example below, when you're using a 1TB file, that might take a long time. From an old command prompt or a PowerShell prompt, issue a single command line to start getting some performance results. Here is your first example using 8 threads of execution, each generating 8 outstanding random 8KB unbuffered read IOs: PS C:\DiskSpd> C:\DiskSpd\diskspd.exe -c1000G -d10 -r -w0 -t8 -o8 -b8K -h -L X:\testfile.dat Command Line: C:\DiskSpd\diskspd.exe -c1000G -d10 -r -w0 -t8 -o8 -b8K -h -L X:\testfile.dat Input parameters: timespan: 1 ————- duration: 10s warm up time: 5s cool down time: 0s measuring latency random seed: 0 path: 'X:\testfile.dat' think time: 0ms burst size: 0 software and hardware cache disabled performing read test block size: 8192 using random I/O (alignment: 8192) number of outstanding I/O operations: 8 stride size: 8192 thread stride size: 0 threads per file: 8 using I/O Completion Ports IO priority: normal Results for timespan 1: * **actual test time: 10.01s thread count: 8 proc count: 4 CPU | Usage | User | Kernel | Idle ——————————————- 0| 5.31%| 0.16%| 5.15%| 94.76% 1| 1.87%| 0.47%| 1.40%| 98.19% 2| 1.25%| 0.16%| 1.09%| 98.82% 3| 2.97%| 0.47%| 2.50%| 97.10% ——————————————- avg.| 2.85%| 0.31%| 2.54%| 97.22% Total IO thread | bytes | I/Os | MB/s | I/O per s | AvgLat | LatStdDev | file ——————————————————————————- 0 | 20480000 | 2500 |**

***1.95 | 249.77 | 32.502 | 55.200 | X:\testfile.dat (1000GB) 1 | 20635648 | 2519 | 1.97 | 251.67 | 32.146 | 54.405 | X:\testfile.dat (1000GB) 2 | 21094400 | 2575 | 2.01 | 257.26 | 31.412 | 53.410 | X:\testfile.dat (1000GB) 3 | 20553728 | 2509 | 1.96 | 250.67 | 32.343 | 56.548 | X:\testfile.dat (1000GB) 4 | 20365312 | 2486 | 1.94 | 248.37 | 32.599 | 54.448 | X:\testfile.dat (1000GB) 5 | 20160512 | 2461 | 1.92 | 245.87 | 32.982 | 54.838 | X:\testfile.dat (1000GB) 6 | 19972096 | 2438 | 1.90 | 243.58 | 33.293 | 55.178 | X:\testfile.dat (1000GB) 7 | 19578880 | 2390 | 1.87 | 238.78 | 33.848 | 58.472 | X:\testfile.dat (1000GB) ————————————————————————————————————- total: 162840576 | 19878 | 15.52 | 1985.97 | 32.626 | 55.312 Read IO thread | bytes | I/Os | MB/s | I/O per s | AvgLat | LatStdDev | file ————————————————————————————————————- 0 | 20480000 | 2500 | 1.95 | 249.77 | 32.502 | 55.200 | X:\testfile.dat (1000GB) 1 | 20635648 | 2519 | 1.97 | 251.67 | 32.146 | 54.405 | X:\testfile.dat (1000GB) 2 | 21094400 | 2575 | 2.01 | 257.26 | 31.412 | 53.410 | X:\testfile.dat (1000GB) 3 | 20553728 | 2509 | 1.96 | 250.67 | 32.343 | 56.548 | X:\testfile.dat (1000GB) 4 | 20365312 | 2486 | 1.94 | 248.37 | 32.599 | 54.448 | X:\testfile.dat (1000GB) 5 | 20160512 | 2461 | 1.92 | 245.87 | 32.982 | 54.838 | X:\testfile.dat (1000GB) 6 | 19972096 | 2438 | 1.90 | 243.58 | 33.293 | 55.178 | X:\testfile.dat (1000GB) 7 | 19578880 | 2390 | 1.87 | 238.78 | 33.848 | 58.472 | X:\testfile.dat (1000GB) ————————————————————————————————————- total: 162840576 | 19878 | 15.52 | 1985.97 | 32.626 | 55.312 Write IO thread | bytes | I/Os | MB/s | I/O per s | AvgLat | LatStdDev | file ————————————————————————————————————- 0 | 0 | 0 | 0.00 | 0.00 | 0.000 | N/A | X:\testfile.dat (1000GB) 1 | 0 | 0 | 0.00 | 0.00 | 0.000 | N/A | X:\testfile.dat (1000GB) 2 | 0 | 0 | 0.00 | 0.00 | 0.000 | N/A | X:\testfile.dat (1000GB) 3 | 0 | 0 | 0.00 | 0.00 | 0.000 | N/A | X:\testfile.dat (1000GB) 4 | 0 | 0 | 0.00 | 0.00 | 0.000 | N/A | X:\testfile.dat (1000GB) 5 | 0 | 0 | 0.00 | 0.00 | 0.000 | N/A | X:\testfile.dat (1000GB) 6 | 0 | 0 | 0.00 | 0.00 | 0.000 | N/A | X:\testfile.dat (1000GB) 7 | 0 | 0 | 0.00 | 0.00 | 0.000 | N/A | X:\testfile.dat (1000GB) ————————————————————————————————————- total: 0 | 0 | 0.00 | 0.00 | 0.000 | N/A %-ile | Read (ms) | Write (ms) | Total (ms) ————————————————- min | 3.360 | N/A | 3.360 25th | 5.031 | N/A | 5.031 50th | 8.309 | N/A | 8.309 75th | 12.630 | N/A | 12.630 90th | 148.845 | N/A | 148.845 95th | 160.892 | N/A | 160.892 99th | 172.259 | N/A | 172.259 3-nines | 254.020 | N/A | 254.020 4-nines | 613.602 | N/A | 613.602 5-nines | 823.760 | N/A | 823.760 6-nines | 823.760 | N/A | 823.760 7-nines | 823.760 | N/A | 823.760 8-nines | 823.760 | N/A | 823.760 max | 823.760 | N/A | 823.760 NOTE: The -w0 is the default, so you could skip it. I'm keeping it here to be explicit about the fact we're doing all reads. For this specific disk, I am getting 1,985 IOPS, 15.52 MB/sec of average throughput and 32.626 milliseconds of average latency. I'm getting all that information from the blue line above. That average latency looks high for small IOs (even though this is coming from a set of HDDs), but we'll examine that later. Now, let's try now another command using sequential 512KB reads on that same file. I'll use 2 threads with 8 outstanding IOs per thread this time: PS C:\DiskSpd> C:\DiskSpd\diskspd.exe -c1000G -d10 -w0 -t2 -o8 -b512K -h -L X:\testfile.dat Command Line: C:\DiskSpd\diskspd.exe -c1000G -d10 -w0 -t2 -o8 -b512K -h -L X:\testfile.dat Input parameters: timespan: 1 ————- duration: 10s warm up time: 5s cool down time: 0s measuring latency random seed: 0 path: 'X:\testfile.dat' think time: 0ms burst size: 0 software and hardware cache disabled performing read test block size: 524288 number of outstanding I/O operations: 8 stride size: 524288 thread stride size: 0 threads per file: 2 using I/O Completion Ports IO priority: normal Results for timespan 1:* *actual test time: 10.00s thread count: 2 proc count: 4 CPU | Usage | User | Kernel | Idle ————————————- 0| 4.53%| 0.31%| 4.22%| 95.44% 1| 1.25%| 0.16%| 1.09%| 98.72% 2| 0.00%| 0.00%| 0.00%| 99.97% 3| 0.00%| 0.00%| 0.00%| 99.97% ————————————- avg.| 1.44%| 0.12%| 1.33%| 98.52% Total IO thread | bytes | I/Os | MB/s | I/O per s | AvgLat | LatStdDev | file ————————————————————————————————————- 0 | 886046720 | 1690 | 84.47 | 168.95 | 46.749 | 47.545 | X:\testfile.dat (1000GB) 1 | 851443712 | 1624 | 81.17 | 162.35 | 49.497 | 54.084 | X:\testfile.dat (1000GB) ————————————————————————————————————- total: 1737490432 | 3314 | 165.65 | 331.29 | 48.095 | 50.873 Read IO thread | bytes | I/Os | MB/s | I/O per s | AvgLat | LatStdDev | file ————————————————————————————————————- 0 | 886046720 | 1690 | 84.47 | 168.95 | 46.749 | 47.545 | X:\testfile.dat (1000GB) 1 | 851443712 | 1624 | 81.17 | 162.35 | 49.497 | 54.084 | X:\testfile.dat (1000GB) ————————————————————————————————————- total: 1737490432 | 3314 | 165.65 | 331.29 | 48.095 | 50.873 Write IO thread | bytes | I/Os | MB/s | I/O per s | AvgLat | LatStdDev | file ————————————————————————————————————- 0 | 0 | 0 | 0.00 | 0.00 | 0.000 | N/A | X:\testfile.dat (1000GB) 1 | 0 | 0 | 0.00 | 0.00 | 0.000 | N/A | X:\testfile.dat (1000GB) ————————————————————————————————————- total: 0 | 0 | 0.00 | 0.00 | 0.000 | N/A %-ile | Read (ms) | Write (ms) | Total (ms) ————————————————- min | 9.406 | N/A | 9.406 25th | 31.087 | N/A | 31.087 50th | 38.397 | N/A | 38.397 75th | 47.216 | N/A | 47.216 90th | 64.783 | N/A | 64.783 95th | 90.786 | N/A | 90.786 99th | 356.669 | N/A | 356.669 3-nines | 452.198 | N/A | 452.198 4-nines | 686.307 | N/A | 686.307 5-nines | 686.307 | N/A | 686.307 6-nines | 686.307 | N/A | 686.307 7-nines | 686.307 | N/A | 686.307 8-nines | 686.307 | N/A | 686.307 max | 686.307 | N/A | 686.307 With that configuration and parameters, I got about 165.65 MB/sec of throughput with an average latency of 48.095 milliseconds per IO. Again, that latency sounds high even for 512KB IOs and we'll dive into that topic later on. 5. Understand the*

parameters used Now let's inspect the parameters on those DiskSpd command lines. I know it's a bit overwhelming at first, but you will get used to it. And keep in mind that, for DiskSpd parameters, lowercase and uppercase mean different things, so be very careful. Here is the explanation for the parameters used above: PS C:\> C:\DiskSpd\diskspd.exe -c1G -d10 -r -w0 -t8 -o8 -b8K -h -L X:\testfile.dat ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ || cliccami || cliccami || cliccami || ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ Parameter || Description || Notes || ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -c || Size of file used. || Specify the number of bytes or use suffixes like K, M or G (KB, MB, or GB). You should use a large size (all of the disk) for HDDs, since small files will show unrealistically high performance (short stroking). || ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -d || The duration of the test, in seconds. || You can use 10 seconds for a quick test. For any serious work, use at least 60 seconds. || ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -w || Percentage of writes. || 0 means all reads, 100 means all writes, 30 means 30% writes and 70% reads. Be careful with using writes on SSDs for a long time, since they can wear out the drive. The default is 0. || ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -r || Random || Random is common for OLTP workloads. Sequential (when –r is not specified) is common for Reporting, Data Warehousing. || ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -b || Size of the IO in KB || Specify the number of bytes or use suffixes like K, M or G (KB, MB, or GB). 8K is the typical IO for OLTP workloads. 512K is common for Reporting, Data Warehousing. || ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -t || Threads per file || For large IOs, just a couple is enough. Sometimes just one. For small IOs, you could need as many as the number of CPU cores. || ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -o || Outstanding IOs or queue depth (per thread) || In RAID, SAN or Storage Spaces setups, a single disk can be made up of multiple physical disks. You can start with twice the number of physical disks used by the volume where the file sits. Using a higher number will increase your latency, but can get you more IOPs and throughput. || ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -L || Capture latency information || Always important to know the average time to complete an IO, end-to-end. || ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -h || Disable hardware and software caching || No hardware or software buffering. Buffering plus a small file size will give you performance of the memory, not the disk. || ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ For OLTP workloads, I commonly start with 8KB random IOs, 8 threads, 16 outstanding per thread. 8KB is the size of the page used by SQL Server for its data files. In parameter form, that would be: –r -b8K -t8 -o16. For reporting or OLAP workloads with large IO, I commonly start with 512KB IOs, 2 threads and 16 outstanding per thread. 512KB is a common IO size when SQL Server loads a batch of 64 data pages when using the read ahead technique for a table scan. In parameter form, that would be: -b512K -t2 -o16. These numbers will need to be adjusted if your machine has many cores and/or if you volume is backed up by a large number of physical disks. If you're curious, here are more details about parameters for DiskSpd, coming from the tool's help itself: PS C:\> C:\DiskSpd\diskspd.exe Usage: C:\DiskSpd\diskspd.exe [options] target1 [target2 [target3 ...]] version 2.0.12 (2014/09/17) Available targets: file_path #<physical drive number> <partition_drive_letter>: Available options: -? display usage information -a#[,#[...]] advanced CPU affinity - affinitize threads to CPUs provided after -a in a round-robin manner within current KGroup (CPU count starts with 0); the same CPU can be listed more than once and the number of CPUs can be different than the number of files or threads (cannot be used with -n) -ag group affinity - affinitize threads in a round-robin manner across KGroups -b<size>[K|M|G] block size in bytes/KB/MB/GB [default=64K] -B<offs>[K|M|G|b] base file offset in bytes/KB/MB/GB/blocks [default=0] (offset from the beginning of the file) -c<size>[K|M|G|b] create files of the given size. Size can be stated in bytes/KB/MB/GB/blocks -C<seconds> cool down time - duration of the test after measurements finished [default=0s]. -D<bucketDuration> Print IOPS standard deviations. The deviations are calculated for samples of duration <bucketDuration>. <bucketDuration> is given in milliseconds and the default value is 1000. -d<seconds> duration (in seconds) to run test [default=10s] -f<size>[K|M|G|b] file size - this parameter can be used to use only the part of the file/disk/partition for example to test only the first sectors of disk -fr open file with the FILE_FLAG_RANDOM_ACCESS hint -fs open file with the FILE_FLAG_SEQUENTIAL_SCAN hint -F<count> total number of threads (cannot be used with -t) -g<bytes per ms> throughput per thread is throttled to given bytes per millisecond note that this can not be specified when using completion routines -h disable both software and hardware caching -i<count> number of IOs (burst size) before thinking. must be specified with -j -j<duration> time to think in ms before issuing a burst of IOs (burst size). must be specified with -i -I<priority> Set IO priority to <priority>. Available values are: 1-very low, 2-low, 3-normal (default) -l Use large pages for IO buffers -L measure latency statistics -n disable affinity (cannot be used with -a) -o<count> number of overlapped I/O requests per file per thread (1=synchronous I/O, unless more than 1 thread is specified with -F) [default=2] -p start async (overlapped) I/O operations with the same offset (makes sense only with -o2 or grater) -P<count> enable printing a progress dot after each <count> completed I/O operations (counted separately by each thread) [default count=65536] -r<align>[K|M|G|b] random I/O aligned to <align> bytes (doesn't make sense with -s). <align> can be stated in bytes/KB/MB/GB/blocks [default access=sequential, default alignment=block size] -R<text|xml> output format. Default is text. -s<size>[K|M|G|b] stride size (offset between starting positions of subsequent I/O operations) -S disable OS caching -t<count> number of threads per file (cannot be used with -F) -T<offs>[K|M|G|b] stride between I/O operations performed on the same file by different threads [default=0] (starting offset = base file offset + (thread number * <offs>) it makes sense only with -t or -F -v verbose mode -w<percentage> percentage of write requests (-w and -w0 are equivalent). absence of this switch indicates 100% reads IMPORTANT: Your data will be destroyed without a warning -W<seconds> warm up time - duration of the test before measurements start [default=5s]. -x use completion routines instead of I/O Completion Ports -X<path> use an XML file for configuring the workload. Cannot be used with other parameters. -z set random seed [default=0 if parameter not provided, GetTickCount() if value not provided] Write buffers: -Z zero buffers used for write tests -Z<size>[K|M|G|b] use a global <size> buffer filled with random data as a source for write operations. -Z<size>[K|M|G|b],<file> use a global <size> buffer filled with data from <file> as a source for write operations.

If <file> is smaller than <size>, its content will be repeated multiple times in the buffer. By default, the write buffers are filled with a repeating pattern (0, 1, 2, …, 255, 0, 1, …) Synchronization: -ys<eventname> signals event <eventname> before starting the actual run (no warmup) (creates a notification event if <eventname> does not exist) -yf<eventname> signals event <eventname> after the actual run finishes (no cooldown) (creates a notification event if <eventname> does not exist) -yr<eventname> waits on event <eventname> before starting the run (including warmup) (creates a notification event if <eventname> does not exist) -yp<eventname> allows to stop the run when event <eventname> is set; it also binds CTRL+C to this event (creates a notification event if <eventname> does not exist) -ye<eventname> sets event <eventname> and quits Event Tracing: -ep use paged memory for NT Kernel Logger (by default it uses non-paged memory) -eq use perf timer -es use system timer (default) -ec use cycle count -ePROCESS process start & end -eTHREAD thread start & end -eIMAGE_LOAD image load -eDISK_IO physical disk IO -eMEMORY_PAGE_FAULTS all page faults -eMEMORY_HARD_FAULTS hard faults only -eNETWORK TCP/IP, UDP/IP send & receive -eREGISTRY registry calls Examples: Create 8192KB file and run read test on it for 1 second: C:\DiskSpd\diskspd.exe -c8192K -d1 testfile.dat Set block size to 4KB, create 2 threads per file, 32 overlapped (outstanding) I/O operations per thread, disable all caching mechanisms and run block-aligned random access read test lasting 10 seconds: C:\DiskSpd\diskspd.exe -b4K -t2 -r -o32 -d10 -h testfile.dat Create two 1GB files, set block size to 4KB, create 2 threads per file, affinitize threads to CPUs 0 and 1 (each file will have threads affinitized to both CPUs) and run read test lasting 10 seconds: C:\DiskSpd\diskspd.exe -c1G -b4K -t2 -d10 -a0,1 testfile1.dat testfile2.dat 6. Tune the parameters for large sequential IO Now that you have the basics down, we can spend some time looking at how you can refine your number of threads and queue depth for your specific configuration. This might help us figure out why we had those higher than expected latency numbers in the initial runs. You basically need to experiment with the -t and the -o parameters until you find the one that give you the best results. You first want to find out the latency for a given system with a queue depth of 1. Then you can increase the queue depth and check what happens in terms of IOPs, throughput and latency. Keep in mind that many logical (and "physical") disks may have multiple IO paths. That's the case in the examples mentioned here, but also true for most cloud storage systems and some physical drives (especially SSDs). In general, increasing outstanding IOs will have minimal impact on latency until the IO paths start to saturate. Then latency will start to increase dramatically. Here's a sample script that measures queue depth from 1 to 16, parsing the output of DiskSpd to give us just the information we need. The results for each DiskSpd run are stored in the $result variable and parsed to show IOPs, throughput, latency and CPU usage on a single line. There is some fun string parsing going on there, first to find the line that contains the information we're looking for, and then using the Split() function to break that line into the individual metrics we need. DiskSpd has the -Rxml option to output XML instead of text, but for me it was easier to parse the text. 1..16 | % { $param = "-o $_" $result = C:\DiskSpd\diskspd.exe -c1000G -d10 -w0 -t1 $param -b512K -h -L X:\testfile.dat foreach ($line in $result) {if ($line -like "total:*") { $total=$line; break } } foreach ($line in $result) {if ($line -like "avg.*") { $avg=$line; break } } $mbps = $total.Split("|")[2].Trim() $iops = $total.Split("|")[3].Trim() $latency = $total.Split("|")[4].Trim() $cpu = $avg.Split("|")[1].Trim() "Param $param, $iops iops, $mbps MB/sec, $latency ms, $cpu CPU" } Here is the output: Param -o 1, 61.01 iops, 30.50 MB/sec, 16.355 ms, 0.20% CPU Param -o 2, 140.99 iops, 70.50 MB/sec, 14.143 ms, 0.31% CPU Param -o 3, 189.00 iops, 94.50 MB/sec, 15.855 ms, 0.47% CPU Param -o 4, 248.20 iops, 124.10 MB/sec, 16.095 ms, 0.47% CPU Param -o 5, 286.45 iops, 143.23 MB/sec, 17.431 ms, 0.94% CPU Param -o 6, 316.05 iops, 158.02 MB/sec, 19.052 ms, 0.78% CPU Param -o 7, 332.51 iops, 166.25 MB/sec, 21.059 ms, 0.66% CPU Param -o 8, 336.16 iops, 168.08 MB/sec, 23.875 ms, 0.82% CPU Param -o 9, 339.95 iops, 169.97 MB/sec, 26.482 ms, 0.55% CPU Param -o 10, 340.93 iops, 170.46 MB/sec, 29.373 ms, 0.70% CPU Param -o 11, 338.58 iops, 169.29 MB/sec, 32.567 ms, 0.55% CPU Param -o 12, 344.98 iops, 172.49 MB/sec, 34.675 ms, 1.09% CPU Param -o 13, 332.09 iops, 166.05 MB/sec, 39.190 ms, 0.82% CPU Param -o 14, 341.05 iops, 170.52 MB/sec, 41.127 ms, 1.02% CPU Param -o 15, 339.73 iops, 169.86 MB/sec, 44.037 ms, 0.39% CPU Param -o 16, 335.43 iops, 167.72 MB/sec, 47.594 ms, 0.86% CPU For large sequential IOs, we typically want to watch the throughput (in MB/sec). There is a significant increase until we reach 6 outstanding IOs, which gives us around 158 MB/sec with 19 millisecond of latency per IO. You can clearly see that if you don't queue up some IO, you're not extracting the full throughput of this disk, since we'll be processing the data while the disks are idle waiting for more work. If we queue more than 6 IOs, we really don't get much more throughput, we only manage to increase the latency, as the disk subsystem is unable to give you much more throughput. You can queue up 10 IOs to reach 170 MB/sec, but we increase latency to nearly 30 milliseconds (a latency increase of 50% for a gain of only 8% in throughput). At this point, it is clear that using multiple outstanding IOs is a great idea. However, using more outstanding IOs than what your target application can drive will be misleading as it will achieve throughput the application isn't architected to achieve. Using less outstanding IOs than what the application can drive may lead to an incorrect conclusion that the disk can't achieve the necessary throughput, because the full parallelism of the disk isn't being utilized. You should try to find what your specific application does to make sure that your DiskSpd simulation is a good approximation of your real workload. So, looking at the data above, we can conclude that 6 outstanding IOs is a reasonable number for this storage subsystem. Now we can see if we can gain by spreading the work across multiple threads. What we want to avoid here is bottlenecking on a single CPU core, which is very common we doing lots and lots of IO. A simple experiment is to double the number of threads while reducing the queue depth by half. Let's now try 2 threads instead of 1. 1..8 | % { $param = "-o $_" $result = C:\DiskSpd\diskspd.exe -c1000G -d10 -w0 -t2 $param -b512K -h -L X:\testfile.dat foreach ($line in $result) {if ($line -like "total:*") { $total=$line; break } } foreach ($line in $result) {if ($line -like "avg.*") { $avg=$line; break } } $mbps = $total.Split("|")[2].Trim() $iops = $total.Split("|")[3].Trim() $latency = $total.Split("|")[4].Trim() $cpu = $avg.Split("|")[1].Trim() "Param –t2 $param, $iops iops, $mbps MB/sec, $latency ms, $cpu CPU" } Here is the output with 2 threads and a queue depth of 1: Param –t2 -o 1, 162.01 iops, 81.01 MB/sec, 12.500 ms, 0.35% CPU Param –t2 -o 2, 250.47 iops, 125.24 MB/sec, 15.956 ms, 0.82% CPU Param –t2 -o 3, 312.52 iops, 156.26 MB/sec, 19.137 ms, 0.98% CPU Param –t2 -o 4, 331.28 iops, 165.64 MB/sec, 24.136 ms, 0.82% CPU Param –t2 -o 5, 342.45 iops, 171.23 MB/sec, 29.180 ms, 0.74% CPU Param –t2 -o 6, 340.59 iops, 170.30 MB/sec, 35.391 ms, 1.17% CPU Param –t2 -o 7, 337.75 iops, 168.87 MB/sec, 41.400 ms, 1.05% CPU Param –t2 -o 8, 336.15 iops, 168.08 MB/sec, 47.859 ms, 0.90% CPU Well, it seems like we were not bottlenecked on CPU after all (we sort of knew that already). So, with 2 threads and 3 outstanding IOs per thread, we effective get 6 total

*outstanding IOs and the performance numbers match what we got with 1 thread and queue depth of 6 in terms of throughput and latency. That pretty much proves that 1 thread was enough for this kind of configuration and workload and that increasing the number of threads yields no gain. This is not surprising for large IO. However, for smaller IO size, the CPU is more taxed and we might hit a single core bottleneck. We can look at the full DiskSpd output to confirm that no single core has pegged with 1 thread: PS C:\DiskSpd> C:\DiskSpd\diskspd.exe -c1000G -d10 -w0 -t1 -o6 -b512K -h -L X:\testfile.dat Command Line: C:\DiskSpd\diskspd.exe -c1000G -d10 -w0 -t1 -o6 -b512K -h -L X:\testfile.dat Input parameters: timespan: 1 ————- duration: 10s warm up time: 5s cool down time: 0s measuring latency random seed: 0 path: 'X:\testfile.dat' think time: 0ms burst size: 0 software and hardware cache disabled performing read test block size: 524288 number of outstanding I/O operations: 6 stride size: 524288 thread stride size: 0 threads per file: 1 using I/O Completion Ports IO priority: normal Results for timespan 1: * **actual test time: 10.00s thread count: 1 proc count: 4 CPU | Usage | User | Kernel | Idle ——————————————- 0| 2.03%| 0.16%| 1.87%| 97.96% 1| 0.00%| 0.00%| 0.00%| 99.84% 2| 0.00%| 0.00%| 0.00%| 100.15% 3| 0.00%| 0.00%| 0.00%| 100.31% ——————————————- avg.| 0.51%| 0.04%| 0.47%| 99.56% Total IO thread | bytes | I/Os | MB/s | I/O per s | AvgLat | LatStdDev | file ——————————————————————————- 0 | 1664614400 | 3175 | 158.74 | 317.48 | 18.853 | 21.943 | X:\testfile.dat (1000GB) ——————————————————————————- total: 1664614400 | 3175 | 158.74 | 317.48 | 18.853 | 21.943 Read IO thread | bytes | I/Os | MB/s | I/O per s | AvgLat | LatStdDev | file ——————————————————————————- 0 | 1664614400 | 3175 | 158.74 | 317.48 | 18.853 | 21.943 | X:\testfile.dat (1000GB) ——————————————————————————- total: 1664614400 | 3175 | 158.74 | 317.48 | 18.853 | 21.943 Write IO thread | bytes | I/Os | MB/s | I/O per s | AvgLat | LatStdDev | file ——————————————————————————- 0 | 0 | 0 | 0.00 | 0.00 | 0.000 | N/A | X:\testfile.dat (1000GB) ——————————————————————————- total: 0 | 0 | 0.00 | 0.00 | 0.000 | N/A %-ile | Read (ms) | Write (ms) | Total (ms) ——————————————- min | 7.743 | N/A | 7.743 25th | 13.151 | N/A | 13.151 50th | 15.301 | N/A | 15.301 75th | 17.777 | N/A | 17.777 90th | 22.027 | N/A | 22.027 95th | 29.791 | N/A | 29.791 99th | 102.261 | N/A | 102.261 3-nines | 346.305 | N/A | 346.305 4-nines | 437.603 | N/A | 437.603 5-nines | 437.603 | N/A | 437.603 6-nines | 437.603 | N/A | 437.603 7-nines | 437.603 | N/A | 437.603 8-nines | 437.603 | N/A | 437.603 max | 437.603 | N/A | 437.603 This confirms we're not bottleneck on any of CPU cores. You can see above that the busiest CPU core is at only around 2% use. 7. Tune queue depth for small random IOs Performing the same tuning exercise for small random IOS is typically more interesting, especially when you have fast storage. For this one, we'll continue to use the same PowerShell script. However, for small IOs, we'll try a larger number for queue depth. This might take a while to run, though… Here's a script that you can run from a PowerShell prompt, trying out many different queue depths: 1..24 | % { $param = "-o $_" $result = C:\DiskSpd\DiskSpd.exe -c1000G -d10 -w0 -r -b8k $param -t1 -h -L X:\testfile.dat foreach ($line in $result) {if ($line -like "total:*") { $total=$line; break } } foreach ($line in $result) {if ($line -like "avg.*") { $avg=$line; break } } $mbps = $total.Split("|")[2].Trim() $iops = $total.Split("|")[3].Trim() $latency = $total.Split("|")[4].Trim() $cpu = $avg.Split("|")[1].Trim() "Param $param, $iops iops, $mbps MB/sec, $latency ms, $cpu CPU" } As you can see, the script runs DiskSpd 24 times, using different queue depths. Here's the sample output: Param -o 1, 191.06 iops, 1.49 MB/sec, 5.222 ms, 0.27% CPU Param -o 2, 361.10 iops, 2.82 MB/sec, 5.530 ms, 0.82% CPU Param -o 3, 627.30 iops, 4.90 MB/sec, 4.737 ms, 1.02% CPU Param -o 4, 773.70 iops, 6.04 MB/sec, 5.164 ms, 1.02% CPU Param -o 5, 1030.65 iops, 8.05 MB/sec, 4.840 ms, 0.86% CPU Param -o 6, 1191.29 iops, 9.31 MB/sec, 5.030 ms, 1.33% CPU Param -o 7, 1357.42 iops, 10.60 MB/sec, 5.152 ms, 1.13% CPU Param -o 8, 1674.22 iops, 13.08 MB/sec, 4.778 ms, 2.07% CPU Param -o 9, 1895.25 iops, 14.81 MB/sec, 4.745 ms, 1.60% CPU Param -o 10, 2097.54 iops, 16.39 MB/sec, 4.768 ms, 1.95% CPU Param -o 11, 2014.49 iops, 15.74 MB/sec, 5.467 ms, 2.03% CPU Param -o 12, 1981.64 iops, 15.48 MB/sec, 6.055 ms, 1.84% CPU Param -o 13, 2000.11 iops, 15.63 MB/sec, 6.517 ms, 1.72% CPU Param -o 14, 1968.79 iops, 15.38 MB/sec, 7.113 ms, 1.79% CPU Param -o 15, 1970.69 iops, 15.40 MB/sec, 7.646 ms, 2.34% CPU Param -o 16, 1983.77 iops, 15.50 MB/sec, 8.069 ms, 1.80% CPU Param -o 17, 1976.84 iops, 15.44 MB/sec, 8.599 ms, 1.56% CPU Param -o 18, 1982.57 iops, 15.49 MB/sec, 9.049 ms, 2.11% CPU Param -o 19, 1993.13 iops, 15.57 MB/sec, 9.577 ms, 2.30% CPU Param -o 20, 1967.71 iops, 15.37 MB/sec, 10.121 ms, 2.30% CPU Param -o 21, 1964.76 iops, 15.35 MB/sec, 10.699 ms, 1.29% CPU Param -o 22, 1984.55 iops, 15.50 MB/sec, 11.099 ms, 1.76% CPU Param -o 23, 1965.34 iops, 15.35 MB/sec, 11.658 ms, 1.37% CPU Param -o 24, 1983.87 iops, 15.50 MB/sec, 12.161 ms, 1.48% CPU As you can see, for small IOs, we got consistently better performance as we increased the queue depth for the first few runs. After a certain number of outstanding IOs, adding more started giving us very little improvement until things flatten out completely. As we kept adding more queue depth, all we had was more latency with no additional benefit in IOPS or throughput. If you have a better storage subsystem, you might need to try even higher queue depths. If you don't hit an IOPS plateau with increasing average latency, you did not queue enough IO to fully exploit the capabilities of your storage subsystem. So, in this setup, we seem to reach a limit at around 10 outstanding IOs and latency starts to ramp up more dramatically after that. Let's see the full output for queue depth of 10 to get a good sense: PS C:\DiskSpd> C:\DiskSpd\DiskSpd.exe -c1000G -d10 -w0 -r -b8k -o10 -t1 -h -L X:\testfile.dat Command Line: C:\DiskSpd\DiskSpd.exe -c1000G -d10 -w0 -r -b8k -o10 -t1 -h -L X:\testfile.dat Input parameters: timespan: 1 ————- duration: 10s warm up time: 5s cool down time: 0s measuring latency random seed: 0 path: 'X:\testfile.dat' think time: 0ms burst size: 0 software and hardware cache disabled performing read test block size: 8192 using random I/O (alignment: 8192)***

number of outstanding I/O operations: 10 stride size: 8192 thread stride size: 0 threads per file: 1 using I/O Completion Ports IO priority: normal Results for timespan 1: ** actual test time: 10.01s thread count: 1 proc count: 4 CPU | Usage | User | Kernel | Idle ————————————————- 0| 8.58%| 1.09%| 7.49%| 91.45% 1| 0.00%| 0.00%| 0.00%| 100.03% 2| 0.00%| 0.00%| 0.00%| 99.88% 3| 0.00%| 0.00%| 0.00%| 100.03% ————————————————- avg.| 2.15%| 0.27%| 1.87%| 97.85% Total IO thread | bytes | I/Os | MB/s | I/O per s | AvgLat | LatStdDev | file ————————————————————————————————- 0 | 160145408 | 19549 | 15.25 | 1952.47 | 5.125 | 8.135 | X:\testfile.dat (1000GB) ————————————————————————————————- total: 160145408 | 19549 | 15.25 | 1952.47 | 5.125 | 8.135 Read IO thread | bytes | I/Os | MB/s | I/O per s | AvgLat | LatStdDev | file ————————————————————————————————- 0 | 160145408 | 19549 | 15.25 | 1952.47 | 5.125 | 8.135 | X:\testfile.dat (1000GB) ————————————————————————————————- total: 160145408 | 19549 | 15.25 | 1952.47 | 5.125 | 8.135 Write IO thread | bytes | I/Os | MB/s | I/O per s | AvgLat | LatStdDev | file ————————————————————————————————- 0 | 0 | 0 | 0.00 | 0.00 | 0.000 | N/A | X:\testfile.dat (1000GB) ————————————————————————————————- total: 0 | 0 | 0.00 | 0.00 | 0.000 | N/A %-ile | Read (ms) | Write (ms) | Total (ms) ————————————————- min | 3.101 | N/A | 3.101 25th | 3.961 | N/A | 3.961 50th | 4.223 | N/A | 4.223 75th | 4.665 | N/A | 4.665 90th | 5.405 | N/A | 5.405 95th | 6.681 | N/A | 6.681 99th | 21.494 | N/A | 21.494 3-nines | 123.648 | N/A | 123.648 4-nines | 335.632 | N/A | 335.632 5-nines | 454.760 | N/A | 454.760 6-nines | 454.760 | N/A | 454.760 7-nines | 454.760 | N/A | 454.760 8-nines | 454.760 | N/A | 454.760 max | 454.760 | N/A | 454.760 Note that there is some variability here. This second run with the same parameters (1 thread, 10 outstanding IOs) yielded slightly fewer IOPS. You can reduce this variability by running with longer duration or averaging multiple runs. More on that later. With this system, we don't seem to have a CPU bottleneck. The overall CPU utilization is around 2% and the busiest core is under 9% of usage. This system has 4 cores and anything with less than 25% (1/4) overall CPU utilization is probably not an issue. In other configurations, you might run into CPU core bottlenecks, though… 8. Tune queue depth for small random IOs, part 2 Now let's perform the same tuning exercise for small random IOS with a system with better storage performance and less capable cores. For this one, we'll continue to use the same PowerShell script. However, this is on system using an SSD for storage and 8 slower CPU cores. Here's that same script again: 1..16 | % { $param = "-o $_" $result = C:\DiskSpd\DiskSpd.exe -c1G -d10 -w0 -r -b8k $param -t1 -h -L C:\testfile.dat foreach ($line in $result) {if ($line -like "total:*") { $total=$line; break } } foreach ($line in $result) {if ($line -like "avg.*") { $avg=$line; break } } $mbps = $total.Split("|")[2].Trim() $iops = $total.Split("|")[3].Trim() $latency = $total.Split("|")[4].Trim() $cpu = $avg.Split("|")[1].Trim() "Param $param, $iops iops, $mbps MB/sec, $latency ms, $cpu CPU" } Here's the sample output from our second system: Param -o 1, 7873.26 iops, 61.51 MB/sec, 0.126 ms, 3.96% CPU Param -o 2, 14572.54 iops, 113.85 MB/sec, 0.128 ms, 7.25% CPU Param -o 3, 23407.31 iops, 182.87 MB/sec, 0.128 ms, 6.76% CPU Param -o 4, 31472.32 iops, 245.88 MB/sec, 0.127 ms, 19.02% CPU Param -o 5, 32823.29 iops, 256.43 MB/sec, 0.152 ms, 20.02% CPU Param -o 6, 33143.49 iops, 258.93 MB/sec, 0.181 ms, 20.71% CPU Param -o 7, 33335.89 iops, 260.44 MB/sec, 0.210 ms, 20.13% CPU Param -o 8, 33160.54 iops, 259.07 MB/sec, 0.241 ms, 21.28% CPU Param -o 9, 36047.10 iops, 281.62 MB/sec, 0.249 ms, 20.86% CPU Param -o 10, 33197.41 iops, 259.35 MB/sec, 0.301 ms, 20.49% CPU Param -o 11, 35876.95 iops, 280.29 MB/sec, 0.306 ms, 22.36% CPU Param -o 12, 32955.10 iops, 257.46 MB/sec, 0.361 ms, 20.41% CPU Param -o 13, 33548.76 iops, 262.10 MB/sec, 0.367 ms, 20.92% CPU Param -o 14, 34728.42 iops, 271.32 MB/sec, 0.400 ms, 24.65% CPU Param -o 15, 32857.67 iops, 256.70 MB/sec, 0.456 ms, 22.07% CPU Param -o 16, 33026.79 iops, 258.02 MB/sec, 0.484 ms, 21.51% CPU As you can see, this SSD can deliver many more IOPS than the previous system which used multiple HDDs. We got consistently better performance as we increased the queue depth for the first few runs. As usual, after a certain number of outstanding IOs, adding more started giving us very little improvement until things flatten out completely and all we do is increase latency. This is coming from a single SSD. If you have multiple SSDs in Storage Spaces Pool or a RAID set, you might need to try even higher queue depths. Always make sure you increase –o parameter to reach the point where IOPS hit a peak and only latency increases. So, in this setup, we seem to start losing steam at around 6 outstanding IOs and latency starts to ramp up more dramatically after queue depth reaches 8. Let's see the full output for queue depth of 8 to get a good sense: PS C:\> C:\DiskSpd\DiskSpd.exe -c1G -d10 -w0 -r -b8k -o8 -t1 -h -L C:\testfile.dat Command Line: C:\DiskSpd\DiskSpd.exe -c1G -d10 -w0 -r -b8k -o8 -t1 -h -L C:\testfile.dat Input parameters: timespan: 1 ————- duration: 10s warm up time: 5s cool down time: 0s measuring latency random seed: 0 path: 'C:\testfile.dat' think time: 0ms burst size: 0 software and hardware cache disabled performing read test block size: 8192 using random I/O (alignment: 8192) number of outstanding I/O operations: 8 stride size: 8192 thread stride size: 0 threads per file: 1 using I/O Completion Ports IO priority: normal Results for timespan 1:* **** actual test time: 10.00s thread count: 1 proc count: 8 CPU | Usage | User | Kernel | Idle ————————————————- 0| 99.06%| 2.97%| 96.09%| 0.94% 1| 5.16%| 0.62%| 4.53%| 94.84% 2| 14.53%| 2.81%| 11.72%| 85.47% 3| 17.97%| 6.41%| 11.56%| 82.03% 4| 24.06%| 5.16%| 18.91%| 75.94% 5| 8.28%| 1.56%| 6.72%| 91.72% 6| 16.09%| 3.91%| 12.19%| 83.90% 7| 8.91%| 0.94%| 7.97%| 91.09% ————————————————- avg.| 24.26%| 3.05%| 21.21%| 75.74% Total IO thread | bytes | I/Os | MB/s | I/O per s | AvgLat | LatStdDev | file ————————————————————————————————- 0 | 2928967680 | 357540 | 279.32 | 35753.26 | 0.223 | 0.051 | C:\testfile.dat (1024MB) ————————————————————————————————- total: 2928967680 | 357540 | 279.32 | 35753.26 | 0.223 | 0.051 Read IO thread | bytes | I/Os | MB/s | I/O per s | AvgLat | LatStdDev | file ————————————————————————————————- 0 | 2928967680 | 357540 | 279.32 | 35753.26 | 0.223 | 0.051 | C:\testfile.dat (1024MB) ————————————————————————————————- total: 2928967680 | 357540 | 279.32 | 35753.26 | 0.223 | 0.051 Write IO thread | bytes | I/Os | MB/s | I/O per s | AvgLat | LatStdDev | file ————————————————————————————————- 0 | 0 | 0 | 0.00 | 0.00 |***

0.000 | N/A | C:\testfile.dat (1024MB) ————————————————————————————- total: 0 | 0 | 0.00 | 0.00 | 0.000 | N/A %-ile | Read (ms) | Write (ms) | Total (ms) ————————————- min | 0.114 | N/A | 0.114 25th | 0.209 | N/A | 0.209 50th | 0.215 | N/A | 0.215 75th | 0.224 | N/A | 0.224 90th | 0.245 | N/A | 0.245 95th | 0.268 | N/A | 0.268 99th | 0.388 | N/A | 0.388 3-nines | 0.509 | N/A | 0.509 4-nines | 2.905 | N/A | 2.905 5-nines | 3.017 | N/A | 3.017 6-nines | 3.048 | N/A | 3.048 7-nines | 3.048 | N/A | 3.048 8-nines | 3.048 | N/A | 3.048 max | 3.048 | N/A | 3.048 Again you note that there is some variability here. This second run with the same parameters (1 thread, 8 outstanding IOs) yielded a few more IOPS. We'll later cover some tips on how to average out multiple runs. You can also see that apparently one of the CPU cores is being hit harder than others. There is clearly a potential bottleneck. Let's look into that… 9. Tune threads for small random IOs with CPU bottleneck In this 8-core system, any overall utilization above 12.5% (1/8 of the total) means a potential core bottleneck when using a single thread. You can actually see in the CPU table in our last run that our core 0 is pegged at 99%. We should be able to do better with multiple threads. Let's try increasing the number of threads with a matching reduction of queue depth so we end up with the same number of total outstanding IOs. $o = 8 $t = 1 While ($o -ge 1) { $paramo = "-o $o" $paramt = "-t $t" $result = C:\DiskSpd\DiskSpd.exe -c1G -d10 -w0 -r -b8k $paramo $paramt -h -L C:\testfile.dat foreach ($line in $result) {if ($line -like "total:*") { $total=$line; break } } foreach ($line in $result) {if ($line -like "avg.*") { $avg=$line; break } } $mbps = $total.Split("|")[2].Trim() $iops = $total.Split("|")[3].Trim() $latency = $total.Split("|")[4].Trim() $cpu = $avg.Split("|")[1].Trim() "Param $paramo $paramt, $iops iops, $mbps MB/sec, $latency ms, $cpu CPU" $o = $o / 2 $t = $t * 2 } Here's the output: Param -o 8 -t 1, 35558.31 iops, 277.80 MB/sec, 0.225 ms, 22.36% CPU Param -o 4 -t 2, 37069.15 iops, 289.60 MB/sec, 0.215 ms, 25.23% CPU Param -o 2 -t 4, 34592.04 iops, 270.25 MB/sec, 0.231 ms, 27.99% CPU Param -o 1 -t 8, 34621.47 iops, 270.48 MB/sec, 0.230 ms, 26.76% CPU As you can see, in my system, adding a second thread improved things a bit, reaching our best yet 37,000 IOPS without much of a change in latency. It seems like we were a bit limited by the performance of a single core. We call that being "core bound". See below the full output for the run with two threads: PS C:\> C:\DiskSpd\DiskSpd.exe -c1G -d10 -w0 -r -b8k -o4 -t2 -h -L C:\testfile.dat Command Line: C:\DiskSpd\DiskSpd.exe -c1G -d10 -w0 -r -b8k -o4 -t2 -h -L C:\testfile.dat Input parameters: timespan: 1 ————- duration: 10s warm up time: 5s cool down time: 0s measuring latency random seed: 0 path: 'C:\testfile.dat' think time: 0ms burst size: 0 software and hardware cache disabled performing read test block size: 8192 using random I/O (alignment: 8192) number of outstanding I/O operations: 4 stride size: 8192 thread stride size: 0 threads per file: 2 using I/O Completion Ports IO priority: normal Results for timespan 1:

** actual test time: 10.00s thread count: 2 proc count: 8 CPU | Usage | User | Kernel | Idle ————————————————- 0| 62.19%| 1.87%| 60.31%| 37.81% 1| 62.34%| 1.87%| 60.47%| 37.66% 2| 11.41%| 0.78%| 10.62%| 88.75% 3| 26.25%| 0.00%| 26.25%| 73.75% 4| 8.59%| 0.47%| 8.12%| 91.56% 5| 16.25%| 0.00%| 16.25%| 83.75% 6| 7.50%| 0.47%| 7.03%| 92.50% 7| 3.28%| 0.47%| 2.81%| 96.72% ————————————————- avg.| 24.73%| 0.74%| 23.98%| 75.31% Total IO thread | bytes | I/Os | MB/s | I/O per s | AvgLat | LatStdDev | file ———————————————————————————— 0 | 1519640576 | 185503 | 144.92 | 18549.78 | 0.215 | 0.419 | C:\testfile.dat (1024MB) 1 | 1520156672 | 185566 | 144.97 | 18556.08 | 0.215 | 0.404 | C:\testfile.dat (1024MB) ———————————————————————————— total: 3039797248 | 371069 | 289.89 | 37105.87 | 0.215 | 0.411 Read IO thread | bytes | I/Os | MB/s | I/O per s | AvgLat | LatStdDev | file ———————————————————————————— 0 | 1519640576 | 185503 | 144.92 | 18549.78 | 0.215 | 0.419 | C:\testfile.dat (1024MB) 1 | 1520156672 | 185566 | 144.97 | 18556.08 | 0.215 | 0.404 | C:\testfile.dat (1024MB) ———————————————————————————— total: 3039797248 | 371069 | 289.89 | 37105.87 | 0.215 | 0.411 Write IO thread | bytes | I/Os | MB/s | I/O per s | AvgLat | LatStdDev | file ———————————————————————————— 0 | 0 | 0 | 0.00 | 0.00 | 0.000 | N/A | C:\testfile.dat (1024MB) 1 | 0 | 0 | 0.00 | 0.00 | 0.000 | N/A | C:\testfile.dat (1024MB) ———————————————————————————— total: 0 | 0 | 0.00 | 0.00 | 0.000 | N/A %-ile | Read (ms) | Write (ms) | Total (ms) ————————————- min | 0.088 | N/A | 0.088 25th | 0.208 | N/A | 0.208 50th | 0.210 | N/A | 0.210 75th | 0.213 | N/A | 0.213 90th | 0.219 | N/A | 0.219 95th | 0.231 | N/A | 0.231 99th | 0.359 | N/A | 0.359 3-nines | 0.511 | N/A | 0.511 4-nines | 1.731 | N/A | 1.731 5-nines | 80.959 | N/A | 80.959 6-nines | 90.252 | N/A | 90.252 7-nines | 90.252 | N/A | 90.252 8-nines | 90.252 | N/A | 90.252 max | 90.252 | N/A | 90.252 You can see now that cores 0 and 1 are being used, with both at around 62% utilization. So we have effectively eliminated the core bottleneck that we had before. For systems with more capable storage, it's easier to get "core bound" and adding more threads can make a much more significant difference. As I mentioned, it's important to keep an eye on the per-core CPU utilization (not only the total CPU utilization) to look out for these bottlenecks. 10. Multiple runs are better than one One thing you might have notice with DiskSpd (or any other tools like it) is that the results are not always the same given the same parameters. Each run is a little different. For instance, let's try running our "-b8K –o4 -t2" with the very same parameters a few times to see what happens: 1..8 | % { $result = C:\DiskSpd\DiskSpd.exe -c1G -d10 -w0 -r -b8k -o4 -t2 -h -L C:\testfile.dat foreach ($line in $result) {if ($line -like "total:*") { $total=$line; break } } foreach ($line in $result) {if ($line -like "avg.*") { $avg=$line; break } } $mbps = $total.Split("|")[2].Trim() $iops = $total.Split("|")[3].Trim() $latency = $total.Split("|")[4].Trim() $cpu = $avg.Split("|")[1].Trim() "Run $_, $iops iops, $mbps MB/sec, $latency ms, $cpu CPU" } Here are the results: Run 1, 34371.97 iops, 268.53 MB/sec, 0.232 ms, 24.53% CPU Run 2, 37138.29 iops, 290.14 MB/sec, 0.215 ms, 26.72% CPU Run 3, 36920.81 iops, 288.44 MB/sec, 0.216 ms, 26.66% CPU Run 4, 34538.00 iops, 269.83 MB/sec, 0.231 ms, 36.85% CPU Run 5, 34406.91 iops, 268.80 MB/sec, 0.232 ms, 37.09% CPU Run 6, 34393.72 iops, 268.70 MB/sec, 0.214 ms, 33.71% CPU Run 7, 34451.48 iops, 269.15 MB/sec, 0.232*

ms, 25.74% CPU Run 8, 36964.47 iops, 288.78 MB/sec, 0.216 ms, 30.21% CPU The results have a good amount of variability. You can look at the standard deviations by specifying the -D option to check how stable things are. But, in the end, how can you tell which measurements are the most accurate? Ideally, once you settle on a specific set of parameters, you should run DiskSpd a few times and average out the results. Here's a sample PowerShell script to do it, using the last set of parameters we used for the 8KB IOs: $tiops=0 $tmbps=0 $tlatency=0 $tcpu=0 $truns=10 1..$truns | % { $result = C:\DiskSpd\DiskSpd.exe -c1G -d10 -w0 -r -b8k -o4 -t2 -h -L C:\testfile.dat foreach ($line in $result) {if ($line -like "total:*") { $total=$line; break } } foreach ($line in $result) {if ($line -like "avg.*") { $avg=$line; break } } $mbps = $total.Split("|")[2].Trim() $iops = $total.Split("|")[3].Trim() $latency = $total.Split("|")[4].Trim() $cpu = $avg.Split("|")[1].Trim() "Run $_, $iops iops, $mbps MB/sec, $latency ms, $cpu CPU" $tiops += $iops $tmbps += $mbps $tlatency += $latency $tcpu += $cpu.Replace("%","") } $aiops = $tiops / $truns $ambps = $tmbps / $truns $alatency = $tlatency / $truns $acpu = $tcpu / $truns "Average, $aiops iops, $ambps MB/sec, $alatency ms, $acpu % CPU" The script essentially runs DiskSpd 10 times, totaling the numbers for IOPs, throughput, latency and CPU usage, so it can show an average at the end. The $truns variable represents the total number of runs desired. Variables starting with $t hold the totals. Variables starting with $a hold averages. Here's a sample output: Run 1, 37118.31 iops, 289.99 MB/sec, 0.215 ms, 35.78% CPU Run 2, 34311.40 iops, 268.06 MB/sec, 0.232 ms, 38.67% CPU Run 3, 36997.76 iops, 289.04 MB/sec, 0.215 ms, 38.90% CPU Run 4, 34463.16 iops, 269.24 MB/sec, 0.232 ms, 24.16% CPU Run 5, 37066.41 iops, 289.58 MB/sec, 0.215 ms, 25.14% CPU Run 6, 37134.21 iops, 290.11 MB/sec, 0.215 ms, 26.02% CPU Run 7, 34430.21 iops, 268.99 MB/sec, 0.232 ms, 23.61% CPU Run 8, 35924.20 iops, 280.66 MB/sec, 0.222 ms, 25.21% CPU Run 9, 33387.45 iops, 260.84 MB/sec, 0.239 ms, 21.64% CPU Run 10, 36789.85 iops, 287.42 MB/sec, 0.217 ms, 25.86% CPU Average, 35762.296 iops, 279.393 MB/sec, 0.2234 ms, 28.499 % CPU As you can see, it's a good idea to capture multiple runs. You might also want to run each iteration for a longer time, like 60 seconds instead of just 10 second. Using 10 runs of 60 seconds (10 minutes total) might seem a little excessive, but that was the minimum recommended by one of our storage performance engineers. The problem with shorter runs is that they often don't give the IO subsystem time to stabilize. This is particularly true when testing virtual file systems (such as those in cloud storage or virtual machines) when files are allocated dynamically. Also, SSDs exhibit write degradation and can sometimes take hours to reach a steady state (depending on how full the SSD is). So it's a good idea to run the test for a few hours in these configurations on a brand new system, since this could drop your initial IOPs number by 30% or more. 11. DiskSpd and SMB file shares You can use DiskSpd to get the same type of performance information for SMB file shares. All you have to do is run DiskSpd from an SMB client with access to a file share. It is as simple as mapping the file share to a drive letter using the old "NET USE" command or the new PowerShell cmdlet "New-SmbMapping". You can also use a UNC path directly in the command line, instead of using drive letters. Here are an example using the HDD-based system we used as our first few examples, now running remotely: PS C:\diskspd> C:\DiskSpd\DiskSpd.exe -c1000G -d10 -w0 -r -b8k -o10 -t1 -h -L \\jose1011-st1\Share1\testfile.dat Command Line: C:\DiskSpd\DiskSpd.exe -c1000G -d10 -w0 -r -b8k -o10 -t1 -h -L \\jose1011-st1\Share1\testfile.dat Input parameters: timespan: 1 ————- duration: 10s warm up time: 5s cool down time: 0s measuring latency random seed: 0 path: '\\jose1011-st1\Share1\testfile.dat' think time: 0ms burst size: 0 software and hardware cache disabled performing read test block size: 8192 using random I/O (alignment: 8192) number of outstanding I/O operations: 10 stride size: 8192 thread stride size: 0 threads per file: 1 using I/O Completion Ports IO priority: normal Results for timespan 1: *** actual test time: 10.01s thread count: 1 proc count: 4 CPU | Usage | User | Kernel | Idle ——————————————————- 0| 12.96%| 0.62%| 12.34%| 86.98% 1| 0.00%| 0.00%| 0.00%| 99.94% 2| 0.00%| 0.00%| 0.00%| 99.94% 3| 0.00%| 0.00%| 0.00%| 99.94% ——————————————— avg.| 3.24%| 0.16%| 3.08%| 96.70% Total IO thread | bytes | I/Os | MB/s | I/O per s | AvgLat | LatStdDev | file ————————————————————————————————– 0 | 158466048 | 19344 | 15.10 | 1933.25 | 5.170 | 6.145 | \\jose1011-st1\Share1\testfile.dat (1000GB) ———————————————————————————————– total: 158466048 | 19344 | 15.10 | 1933.25 | 5.170 | 6.145 Read IO thread | bytes | I/Os | MB/s | I/O per s | AvgLat | LatStdDev | file ————————————————————————————————– 0 | 158466048 | 19344 | 15.10 | 1933.25 | 5.170 | 6.145 | \\jose1011-st1\Share1\testfile.dat (1000GB) ———————————————————————————————– total: 158466048 | 19344 | 15.10 | 1933.25 | 5.170 | 6.145 Write IO thread | bytes | I/Os | MB/s | I/O per s | AvgLat | LatStdDev | file ————————————————————————————————– 0 | 0 | 0 | 0.00 | 0.00 | 0.000 | N/A | \\jose1011-st1\Share1\testfile.dat (1000GB) ———————————————————————————————– total: 0 | 0 | 0.00 | 0.00 | 0.000 | N/A %-ile | Read (ms) | Write (ms) | Total (ms) ——————————————————- min | 3.860 | N/A | 3.860 25th | 4.385 | N/A | 4.385 50th | 4.646 | N/A | 4.646 75th | 5.052 | N/A | 5.052 90th | 5.640 | N/A | 5.640 95th | 6.243 | N/A | 6.243 99th | 12.413 | N/A | 12.413 3-nines | 63.972 | N/A | 63.972 4-nines | 356.710 | N/A | 356.710 5-nines | 436.406 | N/A | 436.406 6-nines | 436.406 | N/A | 436.406 7-nines | 436.406 | N/A | 436.406 8-nines | 436.406 | N/A | 436.406 max | 436.406 | N/A | 436.406 This is an HDD-based storage system, so most of the latency comes from the local disk, not the remote SMB access. In fact, we achieved numbers similar to what we had locally before. 12. Conclusion I hope you have learned how to use DiskSpd to perform some storage testing of your own. I encourage you to use it to look at the performance of the storage features in Windows Server 2012, Windows Server 2012 R2 and Windows Server Technical Preview. That includes Storage Spaces, SMB3 shares, Scale-Out File Server, Storage Replica and Storage QoS. Let me know if you were able to try it out and feel free to share some of your experiments via blog comments. Thanks to Bartosz Nyczkowski, Dan Lovinger, David Berg and Scott Lee for their contributions to this blog post. • Storage, Windows Server 2012 R2, Powershell, SMB, SMB3, Windows Server 2012, Applications, The Basics, Windows ServerinShare37 Save this on DeliciousLeave a Comment• Name • Comment • Post Comments• Jeff Stokes15 Oct 2014 7:34 AMWow. Great post. I am curious how much memory you could test with -h and a 'low' amount of file size. Thoughts here? • Isabelle16 Oct 2014 12:48 PMThis is a very nice post. Sounds more interesting than SQLIO. However, I'm using IOMeter a very free, rich and powerfull tool. One question: To simulate the OLTP SQL load, I use 64K Random not 8k Random since SQL Server

*reads and writes to data files in 64K blocks. • Dan Lovinger [MSFT]16 Oct 2014 7:55 PM@Jeff: -h disables the OS cache and hardware write cache, so the content it will access in that mode is explicitly coming from the device under test. Note that SATA devices generally do not honor requests to disable the hardware write cache. If you want to look at behavior with content in memory, yes, with OS caching enabled (no -h or -S), you'll get that once cache is warmed up. As interestingly, you can disable just the OS cache (-S) and look at the behavior the hardware cache introduces. @Isabelle: DISKSPD source is freely available under the MIT License athttps:*github.com/microsoft/diskspd. I agree IOMeter is a good tool.

Under the classic OLTP loads SQL Server is not 64K random. The SQL Server buffer cache is managed in pages of 8K. If there is ample buffer cache available, a read fault for a single 8K page can cluster up to 56K of adjacent data for a total of a 64K read, but once the buffer cache is warmed up with data, reads under an OLTP load generally fall back to 8K. As with any workload pattern there is a distribution of sizes, but it is dominated by 8K.

Buffer cache writes can cluster up to 256K if there are enough adjacent dirty pages to the page which SQL is trying to commit to the database file; however, again, in my experience 8K dominates.

Log file writes generally range from 10K-20K, in units of sector size. These aren't accounted for in the conventional modeling statements like 2:1 8K random read/write, 70:30, 60:40, 90:10, and so forth (all of which have fan bases; the former cases being in some sense related to the mix under a TPC-C load, the latter under TPC-E).

You can trace the behavior you get under your own workload with the Windows Performance Analyzer.

This paper we did a few years ago covers some of the under-the-hood IO behaviors of SQL Server we observed when working on the SMB CA File Server for Windows Server 2012. You may find it
interesting: http://www.microsoft.com/en-us/download/details.aspx?id=36793

• Mathieu Isabel17 Oct 2014 7:47 PMHi Jose,

We're currently investigating how tiering could be benchmarked more realistically. We use SQLIO to do this but I think DISKSPD has the same issue. Basically we're trying to find a way to provide the benchmarking tool with the notion of hot data. i.e. 10% of the test data file would have high utilization rate while 90% would be rarely accessed. i.e. We need different distributions based on size.

Any ideas how we could achieve that?

Thanks!

• Mathieu Isabel17 Oct 2014 8:11 PMI'm thinking that in SQLIO we can have more than 1 data file in the parameter file which could potentially have different sizes. If SQLIO (or DISKSPD) does the same number of IO on each data file we might be able to have the different density. Would you be able to confirm that?

• Mustafa YUKSEL17 Oct 2014 9:09 PMThanks.

• Philip Elder18 Oct 2014 5:31 AMWhen we get ready to bench test a disk system we start by creating a baseline.

That is we run a series of tests against one disk of each type to get our base.

We can then accurately assess a group of the same disks as we would know optimal queue depths, write sizes, and thread counts.

Our primary testing platform is SOFS (Scale-Out File Server) and Storage Spaces.

IMNSHO if one does not know the baseline performance characteristics of one disk then results via any form of multi-disk testing would be highly suspect.

• tsw20 Oct 2014 5:44 PMgreat article, I'm now using this tool to measure SAN performance.. very helpful.. thank you!

• David Nelson22 Oct 2014 12:39 PMDo you have any recommendations for simulating Hyper-V workloads?