

TDK-dolgozat

Menyhárt Tamás

DEBRECENI EGYETEM
INFORMATIKAI KAR
INFORMATIKAI RENDSZEREK ÉS HÁLÓZATOK TANSZÉK

Konténerizált folyamatok teljesítményelemzése

TDK-pályamunka

Dr. Kocsis Gergely
Egyetemi docens
Polgár Péter
PhD hallgató

Menyhárt Tamás
Programtervező informatikus MSc
szakos hallgató, 1. évfolyam

DEBRECEN

2024

Bírálatra való beküldéshez hozzájárulok.

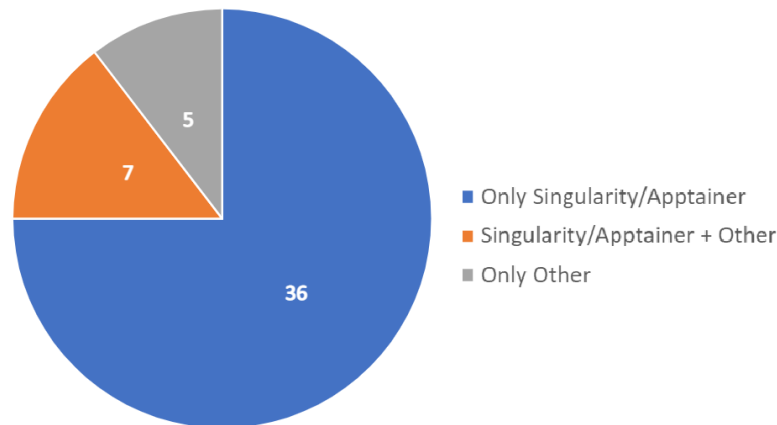
Lezárás dátuma: 2024. 05. 26.

Tartalomjegyzék

1. Bevezetés	3
2. Problémafelvetés és alkalmazott módszerek	4
2.1. A használt hardver specifikációja	5
2.2. Mi a különbség a felhő és a HPC gyakorlati felhasználása között?	7
2.3. Fogalmak a konténerizációval és a virtualizációval kapcsolatban	8
2.3.1. Kernelverzió fontossága (chroot előnye)	11
2.4. Az L1 szinten használt kódokról	12
2.5. Az L2 szinten használt benchmark eszközök	13
2.6. A használt tároló I/O benchmark módszertanról	14
2.7. Az L3 szintű „pipeline” kód ismertetése	15
3. Eredmények	16
3.1. Egyéni mikrobenchmark tesztek (L1)	16
3.1.1. Processzor intenzív benchmarkok egy szálon (L1)	17
3.1.2. Processzor intenzív benchmarkok több szálon (L1)	19
3.1.3. Memória benchmarkok (L1)	22
3.1.4. Kibővített háttértárhasználati benchmarkok (L1)	23
3.1.5. Adatbázis mérések és webserver tesztelések eredményei	27
3.1.6. Videókártyát használó szoftverek a konténerben	29
3.2. Hivatalos benchmark eszközök eredményei (L2)	31
3.2.1. CPU-intenzív benchmarkok egy szálon (L2)	31
3.2.2. CPU-intenzív benchmarkok több szálon (L2)	31
3.2.3. MPI Slurm klaszteren 2 compute node-al	41
3.2.4. Memória benchmarkok (L2)	45
3.2.5. Tárolási I/O benchmarkok (L2)	46
3.2.6. Következtetések (L2)	47
3.3. Pipeline (L3)	48
4. Összefoglalás	49
Köszönetnyilvánítás	51
Kapcsolódó saját publikációk	51
Hivatkozások	52
Saját munka leírása	55

1. Bevezetés

A tudományos számítástechnika eszközei és módszerei végtelen fejlődésen mennek át, lehetővé téve a kutatóknak, hogy egyre bonyolultabb számításokat végezzenek vagy pontosabb eredményeket érjenek el, jobb és jobb eszközök használatával. Ugyanakkor a történelem azt mutatja, hogy az új eszközök és módszerek követése szinte természetesen vezet a kompatibilitási problémákhoz. Másrészt az sem ritka, hogy ugyanazt a számítást vagy szimulációt különböző környezetekben (eltérő hardver- és/vagy szoftverbeállításokban) kell futtatni. A mai válasz ezekre a problémákra a konténerek használata [4]. Ezek használatával a kutatók egységes futtató szoftverkörnyezeteket építhetnek fel egy elkülönített területen belül, és futtathatják számításaikat vagy szimulációikat azokon, a valódi host hardver és szoftverbeállításaira való tekintet nélkül. Sokszor még a grafikus felületről se kell lemondanunk az X-forwarding és a parancssori indítás segítségével. Ebből a szempontból a konténerek úgy működnek, mint a virtuális gépek, azonban, ahogy a legtöbb kutatás eredménye mutatja, mivel sokkal kisebb overheaddel működnek, hatékonyabbak. Ezekre nem teljes értékű virtuális gépekként kell gondolni, hanem sokkal inkább elszigetelt folyamatokként közös kernelen (chroot rendszer). A tudományos számítások esetében a Singularity (manapság úgy ismert, mint Apptainer vagy SingularityCE, amelyekből az utóbbit használtam) konténer a leggyakrabban használt megoldás az inkompatibilitási problémák kezelésére a különböző környezetek között [2]. Mivel az előző állítás a kutatásom kezdetén erősen intuitív jellegű volt csupán (bár általánosan elfogadottnak tűnt), annak bizonyítására megvizsgáltam a Top500 [15] listájának első 172 HPC szolgáltatóját, hogy kiderítsem, milyen konténerizációs technológiákat támogatnak. Ahogy az 1. ábrán látható a 48 megfigyelt szolgáltató közül azok 3/4 része támogatja a Singularity-t- és/vagy az Apptainer-t, míg csak kb. 10% támogat csak más konténerizációs technológiákat. Továbbá ezen 5 szolgáltató között csak két olyan van, amely Singularity-hez hasonló megoldásokat támogat, míg a többiek mindegyike különbözőt használ. Azonban újra és újra felmerül a kérdés, hogy a konténerek alkalmazása hogyan befolyásolja a számítások teljesítményét. Felmerül a kérdés, hogy felhőszolgáltatókkal ellentétben miért marad el például a Docker a HPC környezetekben? Erre is ki fogok térni.



1. Ábra: 48db Top500-as HPC szolgáltató által támogatott konténerizációs technológiák megoszlása. A többség a Singularity/Apptainer konténereket támogatja.

A TDK dolgozat további részében a teljesítményelemzés három szintjén elért eredményeimet az alábbi szerkezetben ismertetem: A 2. szakaszban a vizsgálat részleteit és módszereit írom le először a használt hardver- és operációs rendszer beállításokkal, majd bemutatom a különböző szintekhez használt kódokat és eszközöket. A 3. szakaszban bemutatom az egyes szinteken kapott eredményeket. Az alfejezetek rendre az egyes szinteken kapott eredményeket tartalmazzák. Az L1 és L2 szinten az eredményeket az eltérő erőforrásokra vonatkozóan külön részekre bontom, illetve máshol szerepeltetem az egy- illetve a többszálú futtatásokat. Végül az utolsó szakaszban összegzem az eredményeket.

2. Problémafelvetés és alkalmazott módszerek

A területen végzett legtöbb kutatás azt mutatja, hogy a konténerizáció technológiájának használata általában nem gyakorol jelentős hatást a teljesítményre (sőt, néhány esetben akár felülmúlhatja a natív környezetek teljesítménymutatóit is) [2, 3]. Ugyanakkor nem elfelejtendő, hogy a hardver- és szoftverrendszerek új verzióinak megjelenésével némileg változhatnak az általánosan elfogadott eredmények. Ezért próbáltam ki és teszteltem újra különböző szoftververziókkal, metódusokkal és operációs rendszerekkel, hogy az eredmények ne egy konkrét rendszertől függjenek.

Ezért ebben a munkában a tudományos számításokat három különböző bonyolultsági szinten véve összehasonlítom az natív operációs rendszer és a konténeres környezeteket, három

különböző szempontból (CPU, memória, I/O), hogy kiderülhessen, a 2023 elején aktuális konfigurációk változtatnak-e a széles körben elfogadott eredményeken.

Az első szinten (L1) mikrobenchmarkingot végzek, nagy mennyiségű egyszerű számítás sokszori futtatásával. Ezen a szinten könnyen megtalálhatjuk és felmérhetjük a Singularity / Apptainer konténerek és a natív környezetek teljesítményének erősségeit és gyengeségeit. Ennek a szintnek a másik előnye, hogy ezek a tesztek nagyon jól skálázhatók, így elősegítve, hogy tanulmányozzuk az időfelhasználást a bemeneti paraméterek nagyságának függvényében.

A benchmarkolás második szintjén (L2) jól ismert benchmark eszközök kerülnek használatra [3, 16]. Ez a szint lehetővé teszi, hogy a kapott eredményeket más hasonló eredményekkel is össze lehessen vetni. A jelenlegi felmérésben a CPU, memória és I/O intenzív számításokra és eszközökre kerül a hangsúly, mivel ezek az L1- és mások eredményeivel könnyen összehasonlíthatóak, de a GPU- és a Cuda használatára is ki fogok térni. Továbbá a teljesség kedvéért ezen a szinten is külön vizsgáltam egy és több szálon (vagy processzen) futó teszteket.

A legfelső szinten (L3) egy meglévő társadalomtudományokhoz kapcsolódó tudományos számításokból álló „pipeline”-t futtatok, HPC-hez (High Performance Computing – Szuperszámítógépes környezetre) tervezve. A konkrét eredmények és a számítások célja itt nem érdekes számunkra, de a futási idők és más teljesítménymutatók rálátást adnak arra, hogy milyen főbb különbségek vannak a valós számítási forgatókönyvek viselkedése- és a mikrobenchmarkok és jól ismert benchmark eszközök által mutatott eredmények között.

Fontos megjegyezni, hogy természetesen minden szinten végtelen különböző tesztet végezhetnék az eredmények pontosítására. Ugyanakkor nem lehet az a cél, hogy nyertest hirdessünk a natív és konténeres környezetek közül. Azt szeretném csupán megmutatni, hogy a konténerek hordozhatósága, skálázhatósága, egyszerűsége és karbantarthatósága nem jár számottevő teljesítménycsökkenéssel a számítási folyamatok futtatásakor.

2.1. A használt hardver specifikációja

Az éppen aktuálisan (2022-2023) megjelent legmodernebb processzor és memóriamodul és háttértár került használatra a tesztek végrehajtására egy laptopon, és ehhez 2022-es LTS

kiadású Ubuntut választottam a tesztek hardver- és szoftverkörnyezeteként, így az eredmények érvényesek lehetnek a 2022 végén vagy 2023 elején rendelkezésre álló rendszerekre is. A részletes leírásért lásd az 1. és 2. táblázatot.

Hardware	Típus / Specifikáció
CPU	Intel i7-12700H, 14 core, 20 threads, 5 GHz turbo
Memória	32 GB, DDR5, 4800 MHz
Háttértár	500 GB, Samsung 980 PRO, 7 GB/s, PCIe 4.0
Grafikus egységek	NVIDIA RTX 3050, Intel Alder Lake-P

*1. Táblázat: A mérésekhez használt hardver erőforrások.
Az alkalmazott eszközök aktuálisnak tekinthetők 2022 végén / 2023 elején.*

Software / tulajdonság	Natív (host)	Konténer
Userspace	22.10 UwUntu	22.04 Ubuntu
Kernel verzió	5.15.0.60-generic	5.15.0.60-generic
Fájlrendszer	ext4, mountolt ext3 overlay image	ext3 overlay, host ext4 fájlrendszere
Szabad terület	ext4 host: ~200GB	ext3 overlay: ~70GB
GCC / G++ verzió	Ubuntu 11.3.0	Ubuntu 11.3.0
CPU ütemező	CFS (Completely Fair Scheduler)	

*2. Táblázat: A mérésekhez használt szoftver erőforrások.
Az alkalmazott eszközök aktuálisnak tekinthetők 2022 végén / 2023 elején.*

Bár az alapvető tesztekhez a laptopot használtam, HPC környezetet nehéz elképzelni MPI és több compute node nélkül, így későbbi processzkommunikációs tesztekhez már egy asztali gépet is használtam a Slurm (2023.02) ütemező segítségével. Az MPI teszteknel a laptopon és az asztali gépen is az akkor legújabb 23.04 kiadású Ubuntu futott (kipróbálva a 6-os Linux kernel újdonságait). Sok (nem MPI) tesztet Debian 12.4 (2023 december) rendszeren és később a 6.7 kernellel is megismételtem (természetesen ugyanazon alapú konténerrel, bár chroot rendszer miatt a kernel így is közös). Így a sokszor cserélt és újonnan feltelepített operációs rendszerekkel egy árnyaltabb képet kaphatunk, hogy a különbségek operációs rendszerbéli, szoftverbéli vagy kernelbéli különbségek-e. A lentebb látható konfigurációt (2., 3. táblázat) az MPI tesztekhez használtam (akkor a laptop is 23.04 Ubuntu volt). A Debreceni Egyetem Képalkotó Klinika és Radiológia 2023-tól kezdve már több, mint egy éve alkalmaz általam buildelt Singularity konténereket (Freesurfer, Nnunet, Desmond, Moose, torch etc.) a

Komondor HPC klaszteren, amin (csak) a 4 LTS kernel fut és Azure-ön is. Kompatibilitási probléma és teljesítménybeli különbség (1-2%) nem jelentkezett. (A teljesítménybeli elhanyagolható különbség Azure T4-GPU gépekre vonatkozik. Nem fognak tesztelés céljából szoftvereket telepíteni a klaszterre, ezért építettünk a témavezetőimmel egy kisebb sajátot.) Egy genetikai kutatócsoport most tervezi elkezdni használni ezt a módszertant.

Hardware	Típus / Specifikáció
CPU	Intel i7-12700K, 12 core, 20 threads, 5 GHz turbo
Memória	96 GB, DDR5, 4800 MHz
Háttértár	1000 GB, Gigabyte Aorus 7000, 7 GB/s, PCIe 4.0
Grafikus egységek	NVIDIA RTX 3080, Intel UHD Graphics 770

3. Táblázat: Az MPI mérésekhez használt hardver erőforrások.
Az alkalmazott eszközök aktuálisnak tekinthetők 2023 elején.

Software / tulajdonság	Natív (host)	Konténer
Userspace	23.04 Ubuntu	23.04 Ubuntu
Kernel verzió	6.2.16	6.2.16
Fájlrendszer	ext4, mountolt ext3 overlay image	ext3 overlay, host ext4 fájlrendszere
Szabad terület	ext4 host: ~700GB	ext3 overlay: ~70GB
GCC / G++ verzió	Ubuntu 12	Ubuntu 12
CPU ütemező	CFS (Completely Fair Scheduler)	

4. Táblázat: Az MPI mérésekhez használt szoftver erőforrások.
Az alkalmazott eszközök aktuálisnak tekinthetők 2023 elején.

2.2. Mi a különbség a felhő- és a HPC gyakorlati felhasználása között?

Az előbbi sokkal inkább szolgáltatás alapú, amihez hálózaton keresztül mások is hozzáférnek, például egy folyamatosan futó inference server, adatbázis server, webalkalmazás, tárhelyszolgáltatás etc. Általában több egymástól függetlenül funkcionáló, de együttműködő komponensből épül össze, amiket egy Stack segítségével összefogunk / csoportosítunk, például egy Coderv2, Tritonserver, PostgreSQL hármast, amik egymással kommunikálnak. A horizontális skálázás általában automatikus. Mint szolgáltatást, folyamatosan futtatjuk, monitorozzuk, karbantartjuk, amíg van kreditünk. Érdeemes több konténert használni

szolgáltatásonként és komponensenként. Szoftvereket akár manuálisan is telepíthetünk a kapott gépre vagy mi választhatjuk ki azokat, ha nem szeretnénk konténereket használni. Általában egy vállalathoz vagy csoporthoz tartozik minden jogosult felhasználó.

A HPC ezzel szemben inkább felhasználókra- és kisebb csoportokra bontott, személyes használatot részesít előnyben tudományos számításokhoz. Nincs szükség hálózatépítésre, hozzáférésre vagy emberi interakcióra, futtatáskor a függőségek rendelkezésre állnak, Slurm modulukból betöltődnek vagy a konténer(ek) tartalmazzák. Feladat lehet akár egy AI tanítása, szimulációk, adatelemzés, pipeline etc. A lényeg, hogy ezek a processzek egyszer lefutnak, generálnak kimenetet, aztán leállnak. A kvóta (általában processzoróraban meghatározva) csak a compute node-okon töltött futási időért kerül elszámolásra. Kézzel vagy SBATCH script segítségével adjuk meg, hogy pontosan mennyi erőforrásra van szükségünk, az a job végéig fix, mintha egy vagy több fizikai gépet lefoglalnánk, és azon belül úgy párhuzamosítunk, ahogy akarunk. A feladott job-ok sorban állnak, a Slurm ütemező választja ki a soron következőt. Szoftvereket tekintve nincs telepítési- vagy frissítési jogosultságunk, csak a meglévő modulokat használhatjuk, erre nyújtanak megoldást a konténerek. Minden felhasználó akár teljesen különböző cégekhez is tartozhat, ezért fontos, hogy egymás adatait, konténereit, job-jait ne lássák.

Bár a felhő is használható tudományos számításokhoz, az ára ennek a HPC-hez képest csillagászati, és ha ezek a folyamatok konténerben futnak, akkor könnyű a migráció a két szolgáltató között, akár AWS-Azure, különböző HPC szolgáltatók és felhő-HPC között is. Ehhez csak egyetlen definíciós fájlt kell lefuttatni.

2.3. Fogalmak a konténerizációval és a virtualizációval kapcsolatban

Mielőtt belevágnánk a részletekbe, tudnunk kell, hogy a Singularity egy **chroot** környezet, ami egyszerűsítve annyit jelent, hogy a kernel csak átadja az erőforrásokat egy másik gyökérkönyvtárba, az ott lévő folyamatok pedig közös kernelen működnek tovább. Ennek előnye, ha a kernel nem különbözik nagyon a két rendszer között, akkor natív teljesítményt is vissza tudunk kapni (6.7 Debian 12.5 kompatibilis maradt a 4 LTS Red Hat Linux disztribúcióval

korábban említett képfeldolgozó szoftverek használata esetén), szemben egy izolált-, virtualizált környezettel. Ha a futás lassabb is a régi kernelen ugyanazon hardver- és szoftververziók mellett, a host és a konténer közötti százalékos minimális teljesítménykülönbségek nem változnak. Ezen kívül egyszerűbbé teszi a networkinget, mert képes a host portjait is használni egy szolgáltatásnál, vagy építhetünk saját hálózatot (külön gateway és egyéb konfiguráció nélkül, az adatbázisokról szóló fejezetben lesz róla szó). Például, ha egy MariaDB / MySQL adatbázis fut a 3036-os porton a konténerben, akkor azt a host rendszerén is elértem a 3036-os porton, és localhostként fog számítani (és nem kell újra feltelepíteni, jogokat- és felhasználókat beállítani etc. a host gépen). Kapcsolókkal megoldható, hogy a konténeren belüli portok átirányításra kerüljenek a host egy másik portjához vagy építhető saját belső hálózat. Hátránya, hogy nem Linux alapú rendszereken nem fut (ami még nem probléma) és bizonyos folyamatvezérlő parancsok, pl.: `systemctl` nem működik, mert nem `systemd`-ből volt bootolva a konténer. Viszont a `service` parancs működik, és nagyjából ugyanazt nyújtja nekünk. Daemon alapú konténerizációs technológiák, például Docker ezért sokkal inkább izoláltabban tud működni egy komolyabb teljesítménycsökkenésért cserébe (amit főleg mások által is mért MPI-os és más HPC-s tesztek és cikkek bizonyítanak), viszont jobb a rendszerek közötti átjárhatóság. Chroot környezet miatt a konténer parancssorából való kilépés nem terminálja a folyamatait (pl.: adatbázis-szerver), ha nem sikerül leállítani, vezérelhetjük a `'ps aux | grep NÉV'` és a `'kill FOLYAMAT AZONOSÍTÓ'` parancsokkal. A host látja a konténerek folyamatait és használt portjait, és kill-elni is tudja őket, nem szükséges külön vezérelni őket.

Singularityvel ellentétben a Docker (nagyon leegyszerűsítve) namespace-eket használ a chroot mellé, több komponensből álló fájlrendszerrel, ami egy könnyűsúlyú virtualizációnak tekinthető.

Ha egy gép virtualizáltságának szintjét egy olyan skálán képzeljük el, ami a host rendszertől indul és a teljesen izolált környezethez végződik, akkor a Singularity közelebb áll a hosthoz, míg a Docker az izolált környezethez szemléletmódot és használatot tekintve. Technikailag azt a szoftverrendszert, amit meg lehet csinálni az egyikkel, szinte biztosan meg lehet a másikkal is (megfelelő jogosultságok birtokában). Én mindkettőt használom, és a használati eset dönti el, melyiket érdemesebb inkább választani. Akár IDE-ben is programozhatunk vagy

böngészhetünk a neten NetBeans- vagy Brave- és az X-forwarding segítségével. De már készül a támogatás Waylandhoz is. Nekem a kedvenc IDE-m a GNU Nano, mert a legtöbb rendszeren elérhető és abban már akkor volt dark mode, amikor még IDE sem volt.

Tekintsük a két megoldás közötti különbségeket:

Singularity:

1. Nincs külön démon, nem igényel jogosultságkezelést, nincs 'docker' csoport.
2. Jól integrálódik a Slurm ütemezővel, különösen MPI használata esetén, rugalmas használat a compute node-ok között, kisebb erőforrásigény.
3. 7 + 1 háttértárhasználati lehetőség / képesség.
4. Egyetlen fájlrendszerkép, de futásonként opcionálisan különböző csatolható hozzá (pl.: más bemeneti adatokkal vagy szoftververziókkal). Ez bele is építhető.
5. Inkább egy felhasználónak tervezett, aki speciális szoftvereket használ a számításaihoz.
6. A letöltött képfájlokhoz, volume-khoz, felhasznált erőforrásokhoz, futási információkhoz csak az adott felhasználó és root fér hozzá.
7. Singularity konténerben nincsenek szokásos értelemben vett felhasználók, a /home könyvtárban maximum a felcsatolt host /home könyvtára lehet. Jogosultságkezelés a konténeren belül elég bináris (root vagy bárki más). Az alap .sif fájl overlay nélkül nem írható a buildelés után.
8. Natívan tud letölteni, szerkeszteni, futtatni és átalakítani Docker konténereket.

Docker:

1. Démon alapú rendszer 'docker' csoporttal, aki ehhez hozzáfér, minden futó konténerhez hozzáfér. Van rootless módja is, de a HPC egy többfelhasználós rendszer, ahol a felhasználók nem egy szervezetbe tartoznak.
2. Portainer vagy más szoftver segítségével könnyen Stack-be gyűjthető több konténer, kiépítve egy több komponensből álló szolgáltatást monitorozási lehetőségekkel.
3. Ideális egyetlen erős gép erőforrásainak felosztására. (Erre a Singularity is képes, de a Portainer egy már előre megírt jól monitorozható és konfigurálható megoldást nyújt a compose fájlok használatával.)

4. Támogat több konténerben létrehozott felhasználót, amihez akár SSH-val is csatlakozhatunk.

A `systemctl` és a `service` használata között szintén jól definiálható különbségek vannak:

Funkcionalitás (`systemctl` és `service`):

- A `systemctl` egy általános rendszerkezelő eszköz, amely támogatja a `systemd` szolgáltatások kezelését. Számos extra funkcionalitást és információt biztosít a rendszerről és a szolgáltatásokról. Modernebb, és bővebb eszközkészletű.
- A `service` egy egyszerűbb parancs, amely könnyen használható szolgáltatások indítására és leállítására a SysVinit és Upstart rendszerekkel is.

Kompatibilitás (`systemctl` és `service`):

- A `systemctl` inkább a `systemd` rendszerekre koncentrál, és nem biztosít teljes kompatibilitást más rendszerekkel. Sokan ellenszenvesei emiatt, de jelenleg ez a legjobban működő szolgáltatáskezelő rendszer.
- A `service` parancs a régebbi SysVinit és Upstart rendszerekkel is kompatibilis, így könnyen átírható a különböző rendszereken való használatra.

Elérhetőség (`systemctl` és `service`):

- A `systemctl` nem feltétlenül áll rendelkezésre minden környezetben, különösen chroot vagy konténer környezetekben, ahol a teljes `init` rendszer nem érhető el.
- A `service` és az `/etc/init.d/mysql` parancsok gyakran elérhetők különböző rendszerek esetén, beleértve a konténereket is. Ilyen az SSH is.

2.3.1. Kernelverzió fontossága (chroot előnye)

Általában az újabb kernelverziók javítják a stabilitást, biztonságot és teljesítményt, valamint támogatják az újabb hardvereket. Azonban minden új kernel bevezethet olyan változásokat, amelyek befolyásolhatják a rendszer működését. Előfordulhat, hogy bizonyos eszközök vagy driverek jobban vagy rosszabbul teljesítenek az új kernel alatt. Chroot környezetekben ez még nem szokta elrontani a rendszerhívásokat, funkcionalitásban és végeredményben ugyanazt kellene adják még akkor is, ha a két művelet más ütemezővel vagy módszerrel lett végrehajtva.

Előnye, hogy régi, pl. 16.04 Ubuntu konténer is sokkal jobban és biztonságosabban fog működni frissebb kernelen (teljesítménykülönbség a szoftver, pl.: gcc verzió miatt lehet, de kikényszeríthető rá újabb). Ezt úgy lehet elképzelni, mint a papíron a szorzást. Az egyik viszont összeadásként kezeli. Mindkettő megkapja a helyes végeredményt, csak a sebesség fog különbözni. Mivel úgy is a host kerneljét használja, így az kényszeríti, hogy az ő újabb módszerével számolja ki a szorzás eredményét, így mindig naprakész lesz.

A CFS (Completely Fair Scheduler) egy modern CPU ütemező a Linux disztribúciók operációs rendszereiben, amely a teljesítmény és a méltányosság egyensúlyát hivatott biztosítani. Ez az ütemező a processzeket egy piros-fekete fán tartja nyilván, ami hatékony keresést és beillesztést tesz lehetővé. A CFS a processzek futási idejét méri, és arra törekszik, hogy minden processz egyenlő időt kapjon a processzoron. Az ütemező támogatja a különböző prioritásokat és csoportokat, lehetővé téve a finomhangolt erőforrás-kezelést. A CFS-t Linus Torvalds vezette be a Linux kernel 2.6.23-as verziójában, 2007 októberében. Bár léteznek más CPU ütemezők is, például EEVDF, ha a jövőben CPU ütemező változna- vagy újabb verziót kapna, akkor a konténer a közös kernel miatt a host rendszer ütemezőjét fogja használni, megtartva annak előnyét, szemben egy virtuális géppel.

Összességében elmondható, ha egy új kernel jobb CPU ütemezővel vagy hibajavításokkal kijön, chroot miatt ez kihatással lesz a konténerre is, így az elavulásuk jelentősen lecsökken, teljesítményük nem romlik a kernel változásai miatt, akár nőhet is, és biztonsági szempontból is nagyobb védelmet nyújt, sokkal hamarabb fog alavulni a használt szoftver(verzió).

2.4. Az L1 szinten használt kódokról

Az L1 mérések elsődleges célja egyszerű, gyakran elemi jellegű kód sokszori futtatása annak érdekében, hogy egy adott hardvert vagy annak egy részét (pl. egy processzoron belüli gyorsítótárat) célozva teljesítménnyel kapcsolatos eredményeket kaphassunk. Fontos megjegyezni, hogy természetesen nem lehet tisztán processzor, memória, vagy háttértár összehasonlításról beszélnünk, mert általában egy adott erőforrás nem tud a többi nélkül működni. A fő erőforrás ugyanakkor legtöbbször jól azonosítható. A teszt kódok mellett más

folyamatok is futnak a rendszeren, és az operációs rendszer (bár nagyon jól optimalizált), szintén használ némi erőforrást. Emiatt az összes mérést többször és más rendszereken- és kernelen is elvégeztem, hogy csökkentsem a mérési hiba esélyét, nagyságát.

A kódok többsége fordítóprogramos optimalizálás nélkül kerül bemutatásra. Optimalizálva, a számítás- és memóriaigényes folyamatok futási ideje jelentősen csökken mind natív, mind konténeres környezetben. Megjegyzendő továbbá, hogy mind a konténer-, mind a natív rendszer pontosan ugyanazt a kernelt és fordítóprogramot használta, nem meglepő módon a lefordított C, C++ kód is ugyanaz lett (ezért fontos az azonos verzió). HPC-környezetben nem gyakori, hogy ezeket frissítik az aktuális kiadásra, így a kódunk nem biztos, hogy működhet a HPC-rendszerre telepített szoftverrel, vagy nem fordítható le, vagy nem futtatható, esetleg sokkal lassabban fut bizonyos kapcsolók nélkül (például egyszer nem engedett 'for' ciklusban változót deklarálni kapcsoló nélkül a gcc). De még ha nem is optimalizált, számítási szempontból intenzív kód egy újabb gcc vagy g++ fordítóval még mindig jelentősen gyorsabban fog futni. A konténerek használata éppen azért érdemes, mert ezek a problémák teljesen eliminálhatóak.

2.5. Az L2 szinten használt benchmark eszközök

Annak érdekében, hogy olyan eredményeket kapjunk, amelyek összehasonlíthatóak más munkák eredményeivel, meglévő benchmark eszközöket használtam a natív és a konténeres munkák teljesítményének tesztelésére. Ezeket az eszközöket nevezem hivatalos benchmark eszközöknek a L2 szinten. A következő hivatalos benchmark eszközök kerültek felhasználásra L2 szinten:

CPU benchmarkok (egy szálon futtatva): 7-Zip Compression 22.01 [4, 9, 11, 12]; XZ compression 5.2.4 [11]; CppPerformanceBenchmarks 9; LAME MP3 Encoding 3.100 [11]; FFTW 3.3.6; Geekbench 5.4.6 [7, 20]; Glibc Benchmarks; Himeno Benchmark 3.0; Timed MAFFT Alignment 7.471 [11]; ACES DGEMM 1.0; libjpeg-turbo tjbench 2.1.0

Párhuzamosítás teljesítményét mérő benchmarkok (több szál vagy több processz): pixz [18]; pbzip2 [20]; sysbench [19]; y-cruncher [20]; Intel MKL LINPACK [21]; Geekbench [7, 20]; Intel MPI Benchmarks [22]; BPMF [23]; HPCG Benchmark [16, 24]; miniFE [24, 25]

Memória benchmarkok: CacheBench [11]; MBW 2018-09-08 [9]; RAMspeed SMP 3.5.0 [9, 11, 12]; STREAM 2013-01-17 [11]; sysbench 1.0.20 [1]; Tinymembench 2018-05-28 [9]

Tárolási I/O benchmarkok: Bonnie++ 2.00 [10]; fio 3.28 [5]; FS-Mark 3.3 [6]; IOzone 3.465 [4, 11, 13]; PostMark 1.51 [9, 14]; sysbench 1.0.20 [1]

Az esetek többségében ezeket a benchmarkokat a Phoronix Test Suite (PTS) [11] által futtattam, a kivételek pedig: Geekbench, sysbench, Bonnie++, és fio. A PTS benchmarkok legalább háromszor futottak, és a bemutatott eredmények ezen futási eredmények átlaga.

GPU benchmarkok: Nnunet 1.7.1, Freesurfer (2022 Ubuntu LTS kiadás), Schrodinger 2021-04.

2.6. A használt tároló I/O benchmark módszertanról

Fájlokat különböző módon lehet tárolni mind a natív, mind a konténeres környezetben. Általam használt partíciós séma GPT volt 4096 byte felosztással. Három tárolási lehetőséget vizsgáltam natív környezetben:

1. a natív (host) fájlrendszer (ext4) használata;
2. egy ext3 képfájl használata a host (ext4) fájlrendszeren belül;
3. a /dev/shm használata.

A Singularity konténerkörnyezetben a tárolási I/O standard benchmark eszközök csak az alábbi négy fájlrendszer (sudo nélküli) opcióval futtathatók:

1. a natív (host) fájlrendszer (ext4) használata
2. egy külső ext3 képfájl overlay használata
3. egy ext3 overlay beágyazása a SIF konténerképbe
4. a /dev/shm használata

A /dev/shm egy **memóriával** támogatott fájlrendszer, amelynek maximális mérete általában az összes memória felére korlátozódik, de ez manuálisan is állítható. Fanatikusok telepíthetnek oda akár játékokat is, és memóriából használhatják az egész nagyon gyors

rendszert, csak nem szabad kikapcsolni a gépet, mert a RAM elfelejti a tartalmát, ha nem kap áramot. Meg kicsit sok memória is kell hozzá.

Az összes tárolási I/O benchmarkhoz elegendő szabad hely állt rendelkezésre a tárolón ahhoz, hogy biztosítva legyen az, hogy ne jelenjen meg szignifikáns teljesítmény romlás nem elégséges szabad tárhely miatt.

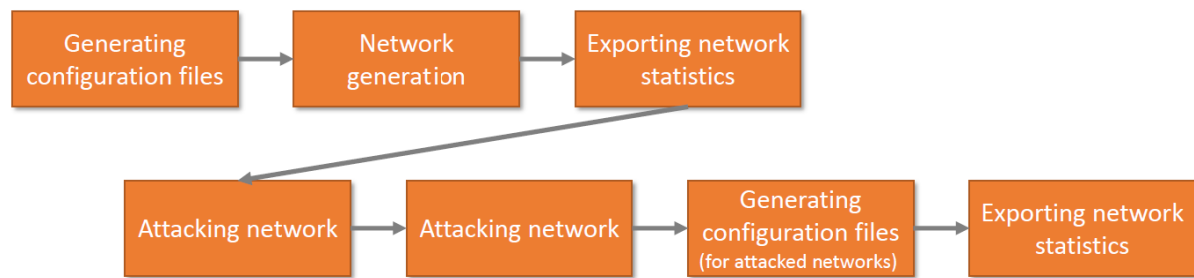
Az előbb említett fájlrendszer opciók nem az összes elérhető, vannak olyanok, amelyek kimaradtak, például a könyvtár overlay vagy a tmpfs overlay, mivel olyan technikai követelményeknek kellene megfelelniük, amelyek nem teljesülhetnek az HPC rendszereken, például root jogosultság szükségessége. Korábban említettem, képes natívan Docker konténereket, és azok fájlrendszerét futtatni és átalakítani. De ennek a célja inkább a kompatibilitás. Emellett nem vizsgáltam a sandbox típusú konténereket, mivel ezek csupán fejlesztési célokat szolgálnak.

2.7. Az L3 szintű „pipeline” kód ismertetése

Az utolsó szinten való vizsgálatokhoz egy valódi hálózattudományban használt C kód pipeline-ját használtam [8, 17], aminek absztrakt struktúráját a 2. ábrán láthatjuk. A pipeline két hasonló változatát is felhasználtam. Illetve ugyanazt a pipeline-t kétféleképp paraméterezve gyakorlatilag két külön verzióról lehet beszélni.

Bár a kód pontos célja nem fontos, ami számunkra jelenleg lényeges az, hogy a terhelés az összes fontos erőforrást igénybe veszi. Például a hálózatok generálása és „támadása” nagy mértékben terheli a CPU-t és a memóriát, és a pipeline szakaszai között a hálózat állapota kiíródnak és betöltődnek a tárolóból.

A példa pipeline csak egy lehetséges beállítása a hivatkozott kutatásokban megjelent kódoknak [8, 17], ugyanakkor fontosnak találtam ezeket a vizsgálatokat elvégezni a másik két szint mellett, hogy láthassuk, milyen kapcsolatban állnak azokkal a valódi alkalmazásokkal, amelyek kevésbé mesterséges benchmark-eredményekből származnak.



2. Ábra: A vizsgálatokhoz használt valódi számítási pipeline vázlata.

A használt kód részletes leírása: [8,17].

3. Eredmények

3.1. Egyéni mikrobenchmark tesztek (L1)

Az L1 szinten végzett vizsgálatom eredményeit az 5. táblázatban foglaltam össze. A következőkben a három fő rendszererőforrásra összpontosítok, és értelmezem a natív és a konténeres környezetekhez kapcsolódó megfigyeléseket, ahogy azt a táblázatban láthatjuk. Bár első pillantásra nyilvánvaló, hogy mindkét környezetnek megvannak az előnyei, meg kell jegyezni, hogy ezek a különbségek csekélyek.

Microbenchmark: (Eltelt idő)	Natív (s)	Konténer (s)	Hatékonyság (%)	Fő hardware erőforrás igény	fájlnév
egész számok	540,50	527,41	102,48%	CPU	egeszszamok.c
w. n.& * 20 bashscript	860,00	853,00	100,82%	CPU (több szál)	backgroundbash. bash
string formázás	473,43	472,92	100,11%	CPU, CPU CACHE	stringformazas.c
sqrt	413,95	433,72	95,44%	CPU	sqrt.c
sqrt több CPU szálon	173,91	174,31	99,77%	CPU (több szál)	sqrtmorethreads. c
sinus, cosinus	378,36	374,84	100,94%	CPU	sincos.c
sinus, cosinus több szálon	88,32	83,53	105,74%	CPU (több szál)	sincosthreads.c
természetes alapú logaritmus	421,33	421,53	99,95%	CPU	log.c
természetes alapú logaritmus több szálon	74,62	73,32	101,78%	CPU (több szál)	logmorethreads.c
Mátrix-Dijkstra algoritmus	7406,86	7418,63	99,84%	CPU, CPU CACHE	matrixutkereses.c pp

Mátrix-Dijkstra algoritmus több szálon	1839,97	1870,75	98,35%	CPU (több szál), memória	matrixutkereses.cpp
Memóriakezelés (nagy méret)	299,08	298,95	100,05%	memória	fajlolvassmemor.y.c

5. Táblázat: Az L1 szinten végzett mérések eredményeinek összesítése.

A színezett cellák 1%-tól nagyobb eltérést jelentenek.

3.1.1. Processzor intenzív benchmarkok egy szálon (L1)

Sokféle mérési módszer létezik a processzor teljesítményének mérésére, mert a sebességét több szempontból is lehet mérni, és az eredmény a mérés típusától függ. Ezért a méréseket az alábbi kategóriákra osztom:

- Processzor szálak: Egy szál / Több szál (dinamikus és statikus szálallokációval).
- Adatszerkezet szempont: Primitívekkel számolva / Komplex / Folytonos / Láncolt adatszerkezetek használatával.
- Cache szempont: 1., 2. és 3. szint.
- Optimalizáció: O2, O3 / O3+ vagy Ofast / optimalizáció nélkül

Hibahatár definiálása kihívást jelenthet, mert az operációs rendszerben más folyamatok is zajlanak vagy épp egy terheltebb, melegebb vagy kisebb sebességű processzorszálát kap meg a folyamat (ha csak kis időkre is) vagy az operációs rendszer máshogy prioritizálja épp a folyamatot. Ezekre igyekeztem figyelni és minimalizálni a hibákat. Kernelbeli különbségekről nem beszélhetünk, szoftververziók pedig ugyanazok. Néha egy saját vagy egy hivatalos benchmark (pl. Geekbench) is ~3%-al rosszabbul teljesített a konténerben, majd a következő méréskor 1%-al leelőzte a host rendszerét. Azonos mérések között 2-3% különbségek voltak legtöbb esetben, olykor kicsit több, ezért teszteltem sokat és sokszor, és átlagoltam őket. Így ~1% különbségeket nem tartok relevánsnak még átlagként sem, 2-3%-ot pedig hibahatáron belülinek fogadok el, és olyan 5+% körüli különbségnél mondható ki, hogy valós különbségek vannak a host és a konténer között.

Elsőként az egyszálú nem optimalizált binárisokra összpontosítok. A konténer jobban teljesített az egész számokkal végzett műveleteknél. Nem volt tapasztalható különbség egyszerűbb adatszerkezetek, például karaktertömbök (stringek) esetén. Egy Dijkstra-

útvonalkereső algoritmust is létrehoztam saját implementációból. Az adattároláshoz főként a harmadik szintű processzor cache-t használta, és nem használt nem egész számokat. Ebben az esetben a natív operációs rendszer kissé gyorsabbnak tűnt, habár hibahatáron belülnek tekinthető.

Az igazi különbségek a lebegőpontos számokkal jönnek. Bár végtelen mennyiségű mérés végezhető el különböző matematikai függvényekkel, a választás a négyzetgyök, a szinusz és a természetes alapú logaritmus számítására esett. A négyzetgyöknél a host jobb eredményt ért el, míg a többi esetben a kettő egyenlőnek vagy a konténer kicsit jobbnak bizonyult. Valószínűleg a (teljesen) saját nem optimalizált folyamatait előbbre veszi a rendszer (a számításigényesség miatt nem tud mindent kiszámolni a kis időközök alatt a processzor), de ha nagyon sok feladatot adtam neki, akkor a különbség 1% alá csökkent minden vizsgált matematikai függvény esetén. Újra tesztelve közepes optimalizációval (O2) a futási idők kb. 18%-al csökkentek, és kisebb bemenetekre is 1% alá csökkentek a különbségek a rendszerek között. Ennél még elég jól használhatóak a lefordított kódok más gépeken, bár gyakorlatban érdemes inkább a konténer buildelése esetén adott rendszeren lefordítani és a legnagyobb optimalizációt használni kódjaink és telepített szoftvereink esetében is. Figyelni kell arra, hogy ha saját kódban ciklusokat vagy elágazásokat írunk, és benchmarkolni szeretnénk, akkor használjuk fel valahol az adatot később, és ne mindig ugyanabba az ágba essen az if, mert a fordító optimalizáláskor lehet, hogy teljesen átírja vagy ki is veszi a kódból az elágazást vagy a ciklust.

Az eredmények alapján fontos az optimalizálás szerepe, mert az **O3 szintű optimalizáció akár negyedére is csökkentheti a lebegőpontos számítások idejét**. Megjegyzendő, hogy a futási idő különbség optimalizálásakor kevesebb, mint 1%-ra csökkent minden (L1) processzorbenchmark esetén (kernelverziótól függetlenül). Érdekeség, ha "csak" egy két évvel régebbi (2020) gcc verziót használtam, futási idő körülbelül 20%-kal romlott, továbbá lehet, hogy a kódunk csak külön kapcsolókkal vagy egyáltalán nem fordul le. (Ez gyakori, különösen akkor, ha másik gépen vagy HPC-n futtatjuk, ahol nem frissítik a fordítókat a legújabb verzióra érthető nagyvállalati okokból.) Tehát fontos, hogy a szoftvert a konténerben minél naprakészebb állapotban tartsuk és optimalizáljunk, mert az már ténylegesen jelentős hatással lehet a teljesítményre.

Az eredmények összefoglalásaként elmondható, hogy a CPU-használat eredményei maximum 5% hibahatáron belül vannak, nem feltétlen mindig a host javára, ami az L1 benchmarkok szintjén az egy szálon futó műveletek esetében, ami a konténerizáció használata számára meghatározó érvet jelenthet. Ha továbbá újabb szoftververziót telepítünk a konténerbe és optimalizálunk, akkor előny alakulhat ki, nagyságrendekkel nagyobb, mint a gazdagép és a konténer közötti elenyésző pár %-os eltérések.

3.1.2. Processzor intenzív benchmarkok több szálon (L1)

Mivel a natív operációs rendszer és a konténer közti memóriakezelése között nincs különbség (egyik mérési szinten sem) és az egy szálon futó műveleteknél is legtöbb esetben elenyészőek a különbségek, ezért a programozó elkezdhet azon gondolkodni, hogy lehetne-e egy adott feladatot gyorsabban végrehajtani. Például, ha a tesztelt rendszer 20 processzorszállal rendelkezik.

Erre egy régi bevált módszer az, hogy többszálásítunk (amennyiben lehetséges). Egy példa erre a mátrixszorzás. Ez a témakör azonban rendkívül összetett. A legfontosabb dolog, hogy tudjuk, a végrehajtás sorrendje nem feltétlenül az, amit szekvenciálisan hajtánánk végre, és ha több szál próbál egy memóriaterületre konkurensen írni, az komoly következményekkel járhat és sokszor még hibaüzenetet sem kapunk róla.

Többféle szálkiosztás között válogathatunk, ha egy adott feladatot nem szekvenciálisan szeretnénk végrehajtani. A legnépszerűbb választások ezek közül a statikus, dinamikus és „guided”. Én az első kettővel fogok foglalkozni az (L1) szinten. Mindegyik módszernek megvannak az előnyei és a hátrányai. További változó paraméter lehet, hogy egy gépnek nem minden processzormagja, ezáltal processzorszála egyenrangú (pl.: az Intelnél az effektív- és teljesítménymagok a 12.+ generációtól, egy több processzorral rendelkező szuperszámítógép esetén vagy alaplapi szinten kontrollált magonkénti overclock-beállításoknál). Tartsuk észben, hogy attól, hogy többszálásítunk, az adott szál az idejének nem feltétlen a 100%-át tölti a programunk végrehajtásával. Például, ha allokálnánk a gép összes processzorszállát, és ténylegesen csak a programunkkal foglalkozna mind, akkor az operációs rendszer lefagyna, leállna. Gondoljunk a „context switch” jelentőségére. (Ez egyébként egy szála is igaz.) Ezen túl van költsége a szálak közötti kommunikációnak is. Egy processzormagon belül gyorsabban

kommunikál két szál, mint magán a processzoron belül, ha pedig több processzorunk van, akkor ennek a költsége még magasabb. Az eltérések változhatnak attól függően, mennyi szálat adunk egy folyamatra.

A statikus kiosztáskor egy „for” ciklus esetén minden szál ugyanannyi (+1) iterációt kap. Hátránya lehet, ha egyes iterációk között exponenciális futási idő különbségek vannak, akkor megeshet, hogy adott processzorszál már befejezte a rá kiszabott összes feladatot, és tétlen marad, nem vesz át a még üres lépések közül egyet sem. Gyakorlatban pedig a szóban forgó szál általában kap az operációs rendszertől valami más feladatot, és azt kezdi el csinálni, miközben a programunk még véget sem ért.

Dinamikus szálkiosztás esetén nincs megszabva, hogy a processzorszálak mennyi iterációt kapnak külön-külön. Ezáltal az sem megjósolható, hogy melyik fogja megkapni az adott feladatot, és mikor. Előnyös lehet, ha egy-egy lépés végrehajtásakor nagyságrendi időkülönbségek vannak vagy az adatszerkezet különböző méretű listákra épül. Hátránya viszont, ha a számítási költsége minden iterációnak nagyjából megegyezik (pl.: mátrixszorzás), akkor mivel az operációs rendszer folyamatosan újra és újra kell osztania a szálaknak a folyamatokat, így rosszabbul járunk, mint egy statikus kiosztással.

Mint ahogy az 5. táblázatban látható, volt lehetőségem többszálúsításra a legtöbb benchmarknál. Optimalizáció nem került használatra itt sem. (Bár itt is igaz, hogy a futási idő jelentősen csökken, és a különbségek is sokkal kisebbek lesznek.) Az egész számoknál végzett műveleteknél a bash scriptben elindítottunk párhuzamosan több folyamatot, a különböző matematikai függvényeknél kihasználtam az egy iterációra eső számítási költség egyenlőségét, így statikus kiosztást használtam, míg a Dijkstra-algoritmusra épülő implementációban, ahol a bemenő gráfok méretei nagyságrendekben különböznek, dinamikus szálkiosztást választottam. Érdekessége, hogy végig referenciákat, konstans referenciákat és pointereket használ, így ha szekvenciálisan futtatjuk (ahogy eredetileg), akkor az egy adott gráf betöltődik a level 3 processzor-cache-be, és el is fér benne, amíg dolgozik vele. Többszálúsításkor, amikor több gráfot néz egyszerre, az mind már nem fér bele a közös processzor-cache-be, így a memóriából kell olvasnia. Van level 1,2,3 is, attól függően, hogy a processzor mekkora részéhez tartozik (pl.: csak egy adott szálhoz, maghoz vagy az egészhez). Ha nem egy szálon

futtatunk, akkor elveszik a közös level 3 cache használatának az effektív kihasználása, a memória pedig nagyságrenddel lassabb, ez egy (további) oka, hogy a 20 szálunk esetében nem lesz 20-szor gyorsabb a programunk. A cache effektív kihasználása függ attól is, hogy az adatszerkezetünk folytonos vagy szétszórt reprezentációval ábrázolt, mert az előbbit egyszerre be tudja tölteni.

Észrevehetjük, hogy a gyökvonásnál a natív operációs rendszer legnagyobb előnye teljesen elveszett, egésze kerekítve a kettő futási ideje %-ban megegyezik, ha statikusan többszálúsítottunk, így bőven mérési hibán belül van. Hasonló esemény történt a természetes alapú logaritmus számolása esetében. Itt a konténer előnye tűnt el, viszont még mindig nem teljesített rosszabbul, mint a host. A dinamikus kiosztású Dijkstra-algoritmus esetében is néhány másodperc eltéréssel egyenlőek a hosszú futási idők, bár itt a host egy kissé gyorsabb volt. Amennyiben egész számokkal végeztünk műveleteket, akkor maradt a konténer apró előnye, viszont ez egyáltalán nem jelentős változás. Ami érdekes, hogy a szögfüggvényeket több szálon 5%-al gyorsabban számolta ki a konténer, ami egyébként tényleges és kicsit misztikus különbség, de nem számottevő. A **Linux** kernelre épülő operációs rendszerek egyébként híresek a **párhuzamos folyamatok hatékony kezeléséről** (Windows-al összehasonlítva). A különbségek eltűnésének is ez állhat a hátterében.

Beszéljünk az elhasznált processzoridőről, az eltelt valós időről és a költségekről.

- Legtöbb esetben **20** szálnál csupán **~4x** lett gyorsabb a feladat végrehajtása
- Az elhasznált processzoridőt tekintve **~75%-os** kihasználtság volt több szálon
- Az elhasznált processzoridőt tekintve **~96%-os** kihasználtság volt egy szálon

A fenti sorok azt jelentik, hogy 1 szál helyett 20-at felhasználva nem 20-szor, csak 4-szer futott le gyorsabban (valós időt tekintve) a kód, tehát volt veszteség. Viszont a feladatra elhasznált processzoridő hatékonysága többszálúsításnál 75% hatékonyságú volt. Ez azt jelenti, hogy ha 75 feladatot elvégzünk több szálon, akkor 1 szálon (szépen lassan) 100 feladatot tudunk elvégezni ugyanannyi processzoridő felhasználásával. Felvetődik a kérdés, hogy 20/4 az még mindig 5, azaz 20%-os hatékonyságot feltételez, nem 75%-öt. A processzor valóban 75%-os hatásfokkal teljesítette több szálon ugyanazt a feladatot a job-ra szánt processzoridőt tekintve, viszont a szálkiosztás és operációs rendszerbeli folyamatok, szálak közötti

kommunikáció miatt a processzor nem csak azzal az egy programmal foglalkozott. (Lehet, hogy csak alacsony órajelen „pihent”, míg nem kapott új iterációt.) És ne feledjük, hogy minél több magot használunk ki, annál jobb hűtésre lesz szükség, amennyiben ez megoldhatatlan, az operációs rendszer visszaveszi a számítási egység teljesítményét, így áramfogyasztását, de ezt más okokból kifolyólag is megteheti. (Tehát csak a processzorra vetítve és csak az adott egy programra, az energiafelhasználás és elvégzett munka több szálon 75%-os volt, ha egy szállal hasonlítom össze.)

Költségeket tekintve megnyugodhatunk, hogy a villanyszámla nem fog annyira megugrani, ha többszálúsítunk. A gépben amúgy sem feltétlenül a processzor fogyasztja a legtöbb áramot. Ha felhőben üzemeltetünk, és mi az elhasznált processzoridőért fizetünk, akkor is 75% hatékonysággal tudunk többszálúsítani. Ha viszont HPC környezetben szeretnénk ezt, kicsit bajban vagyunk a 20% hatásfokkal, mert ott a processzorszálakat mi lefoglaljuk előre, és amennyi tetszik, annyit felhasználunk belőle (tehát a valós időért fizetünk, ami lefoglalt processzorszálak száma * eltelt valós idő). Az, hogy nem ketyeg 100%-on a szál, az nem számít. Ha felhőben vagy helyi gépen szeretnénk futtatni, akkor egyértelműen megéri többszálúsítani. HPC-n a kérdés inkább az, hogy megéri-e többet fizetni ugyanazért a feladatért, cserébe gyorsabban megkapjuk rá az eredményt. A válasz általában igen, mert nincs ezer évünk, de ezer dollárunk lehet. Persze ez most nem azt akarja jelenteni, hogy bizonyos felhőszolgáltatónál jobban megéri futtatni, mert a felhőben mindenért fizetni kell, és sokszor a processzorórát drágábban adja, mintha HPC-n rossz kihasználtsággal lefoglalnánk szálakat, és ott (többször) semmi másért nem kell fizetni.

A végső konklúzió a (L1) többszálúsításhoz, hogy igen, ha tudjuk mit csinálunk, akkor megéri többszálúsítani némi többletköltségért az időnk javára.

3.1.3. Memória benchmarkok (L1)

Mivel a DDR5 memória olyan gyors (harmadával jobb, mint elődje, a DDR4), hogy még ha sok van belőle, akkor is nagyon gyorsan felülírja önmagát. Ezért rengeteg mennyiségű nagy memóriefoglalás, olvasás, másolás, felülírás és felszabadítási utasításokat végeztem el. Gyakorlatilag 256-szor került sor egy nagy rész memóriára történő foglalására, másolására, felülírására és felszabadítására. A 32 GB RAM-ból majdnem az egész ki lett használva 256

alkalommal 5 perc alatt, ami rengeteg adat. A cél az volt, hogy az adatmennyiség ne férjen bele a processzor gyorsítótárába, de ne is kelljen a memóriát visszaírni az adathordozóra („swap” megakadályozása).

Persze processzor nélkül memória sem tud működni, de szerencsére processzorhasználatban a különbségek elenyészőek voltak. Ráadásul a memória amúgy is nagyságrenddel lassabb, mint a processzor, tehát ha (CPU) különbség is lenne, se számítana sokat. Az memóriateszt eredményében alig volt 0,05%-os különbség, az előnyére a konténernek. A következtetés az, hogy még a legrosszabb nézőpontból is a konténer ugyanolyan jól kezeli és használja a memóriát, mint a natív operációs rendszer az L1 benchmark szintjén.

3.1.4. Kibővített háttértárhasználati benchmarkok (L1)

Háttértárat is sokféleképpen lehet mérni. Az L1 benchmark szintjén a szekvenciális írást, olvasást és másolást vizsgáljuk. A konténer használhatja a gazda ext4 (GPT 4096) fájlrendszerét vagy saját egyéni ext3 (4096) overlay fájlját különválasztható .img kiterjesztéssel. Ezt az overlay fájlt a gazdagép is használhatja, ha létrehoz egy mappát és beállítja a csatlakoztatást. Az overlay hasznos lehet, mert például lehetővé teszi a programok telepítését vagy bemeneti adatokat, amelyeket a konténer futási időben használhat. Ellenkező esetben a konténer egyszerre használhatja a host és a saját fájlrendszerét is, akár beépített overlay mellett. Hiba nélkül megy a cp, mv és más hasonló parancsok. Az ezekhez a vizsgálatokhoz tartozó eredményeket a 6. és a 7. táblázatokban foglaltam össze.

Fontos megjegyezni, hogy a méréseket negyedik generációs SSD meghajtóval végeztem, ami nagyságrenddel gyorsabb, mint egy szabványos M2-SATA III SSD vagy bármely merevlemez. Szekvenciális olvasást, írást és másolást hajtottam végre. (Természetesen nincs rendszeresség a fájl tartalmakban.) A fájl méret körülbelül 18 GB volt. Általánosságban elmondható, hogy:

- **Írás:** a konténer kb. 2-5% lassabb, mint a host
- **Olvasás:** a konténer kb. 3-11% gyorsabb, mint a gazdagép
- **Másolás:** a konténer kb. 2% gyorsabb, mint a host

Háttértár benchmarkok: (Eltelt idő)	Natív (s)	Konténer (s)	Hatékonyság (%)	E. (%) vs. host r/w
--	-----------	--------------	-----------------	---------------------

file írás (ext4 host)	105,73	107,03	98,78%	98,78%
file olvasás (ext4 host)	26,45	23,80	111,13%	111,13%
file másolás (ext4 host)	22,43	22,02	101,86%	101,86%
file írás (ext3 overlay)	124,07	130,59	95,01%	80,96%
file olvasás (ext3 overlay)	26,75	25,94	103,09%	86,46%
file másolás (ext3 overlay)	33,20	22,31	148,79%	100,53%
/dev/shm írás (Memory file system)	98,70	99,38	99,32%	106,39%
file olvasás (4,4 GB) (ext4 host)	7,22	6,07	119,06%	119,06%
/dev/shm olvasás (4,4 GB)	6,08	7,27	83,69%	99,38%
file másolás (4,4GB) (ext4 host)	3,81	3,72	102,42%	102,42%
/dev/shm másolás (4,4GB)	1,34	1,28	104,10%	296,66%

6. Táblázat: Kibővített háttértár benchmarking eredmények natív és konténeres környezetben

Háttértár benchmarkok (csak konténerben)	Írás (s) (ha lehetséges)	Olvasás (s) (ha lehetséges)	E. (%) vs. host r/w	
embedded overlay	104,52	25,86	101,16%	102,31%
A SIF fájlrendszer részeként	nem elvégezhető	67,68	nem releváns	39,09%

7. Táblázat: Csak konténeres környezetben futtatható háttértár benchmarking eredmények

Amikor a gazdagép az overlay-re akart másolni, lényegesen lassabb volt, mint amikor a konténer saját fájlján végezte a műveletet. A konténer 22,31 másodpercet vett igénybe az ext3 fájlrendszer másolására, míg a host 22,43 másodpercet az ext4 fájlrendszer másolására. Ez csak 2% különbség. Ennek az eltérésnek az oka az, hogy a gazdagép lassabban másol az ext3 konténernek készített fájlrendszerbe, mint az ext4-be. Ezen kívül az írási művelet kissé lassabb,

amikor az ext3 .img overlayt használjuk. Az ext4 fájlrendszer hatékonyabbnak bizonyult, mint az ext3, de a konténer amúgy is tudja natívan használni a host fájlrendszerét.

Természetesen sokféle módszer létezik az olvasás/írás/fájlmásolás elvégzésére. Több fájl is másolható egyszerre, vagy sok kis fájlt lehet egymás után másolni, vagy sok nagy fájlt egyszerre stb. Azonban nálam csak az írás teljesített rosszabbul az L1 benchmarkoknál a konténerek esetében. Az összes benchmark közül a háttértár eredményezte a legnagyobb különbségeket, de ezek a különbségek még mindig kicsik, nem több nagyságrenddel eltérőek. Az eltérések inkább a fájlrendszer használatbeli különbségekből és nem a host / konténer teljesítményéből fakadnak. Szerencsére használhatják ketten ugyanazt a fájlrendszert, ezért van értelme a fenti összehasonlításnak. Az in-memory fájlrendszerek is befolyásolhatják az eredményeket. Emellett 7+1 módszer létezik a fájlok kezelésére a konténerek esetében (ezek közül sok bemutatásra kerül a 3.2.3-as szakaszban), így a felhasználásnak megfelelő legmegfelelőbb módszert lehet választani. Ahogy korábban említésre került, a konténer meglehetősen szerteágazó módokon tudja használni a háttértárat, köztük úgy is, ahogy a natív host rendszer nem is képes, ebből pár csak „sudo” jogosultsággal lehetséges, de azokkal nem foglalkozok. Az összehasonlításban ezért két esetet különíték el:

1. A natív host és a konténer tesztelése ugyanazon fájlstruktúrán (ha ez lehetséges).
2. A konténer adott rendszeren való teljesítményének összevetése a host ext4 fájlrendszerével.

A második lehetőség célja, hogy képet kapjunk, hogy pl.: mennyivel gyorsabban másol a konténer a /dev/shm memóriabeli fájlrendszeren a gazdagép natív ext4-én történő műveletéhez képest. Az (L1) mérésekben a szekvenciális írásra, olvasásra és másolásra koncentráltam nagy (~18GB) méretű fájlokkal. Néhány esetben a memória korlátossága miatt (pl.: a /dev/shm-be való másolásakor) vissza kellett venni a fájl méretét. A szövegfájl tartalma többnyire számokból áll, és teljesen szabálytalan.

A /dev/shm lényege, hogy kikerüljük a háttértár használatát, és minden kizárólag a memóriában menjen végbe. Az eredményeket tekintve meglepő lehet az, hogy mi a mai leggyorsabb SSD-t használtam, aminek a sebessége bár nem éri el, de megközelíti a tényleges RAM sebességét, további befolyásoló tényező lehet, hogy én a DDR4 helyett DDR5

memóriamodulokat használtam dual-channel módban. Amikor az adott mappában dolgozunk, akkor a művelet a memóriában hajtódik végre, és abba is íródik, olvasódik be (tehát két műveletet végez, még a háttértár csak egyet.).

A memóriabeli fájlrendszerbe való íráskor kissé a host teljesített jobban, míg olvasásnál a gazdagép sokkal nagyobb előnyre tett szert. Elmondható volt, hogy az olvasás nem igazán különbözött attól, mintha csak az ext4-en hajtánánk végre, az írás viszont valamennyivel gyorsabb volt. Másolási műveletet tekintve a konténer minden esetben valamivel jobb volt, viszont itt már kiugróan magasan, majdnem 3x gyorsabban végrehajtódott a másolás a konténerben (és hasonló aránnyal a hoston is), mintha ext4-en csinálta volna a gazdagép. Bár a memória közös, a konténer összefüggően használ jóval kevesebb területet, ez befolyásolhatta az eredményt. Ezek az eredmények azért is érdekesek, mert megmutatják, hogy egy mai korszerű SSD meghajtó sebessége nagyságrendileg felér azzal, mintha a memóriában olvasnánk vagy íránk a háttértár megkerülésével, viszont a másolás művelet még mindig nagyságrenddel lassabb. Viszont tartsuk szem előtt, hogy a gen3 m.2 SSD-k, SATA III SSD-k és HDD-k máig népszerűek, és nagyságrenddel lassabbak, mint a gen4, így nem csak a memóriában végzett másolás lenne nagyságrenddel gyorsabb, hanem az írás és az olvasás is, ezen kívül az SSD-k élettartama véges, sokkal inkább, mint egy HDD-é vagy a RAM-é és még jóval drágábbak is, mint elődeik.

A konténer `.sif` fájlkiterjesztése / fájlrendszere nem írható, csak olvasható, ezért szokta valamilyen módon használni a host fájlrendszerét vagy a neki kialakított overlay fájlkat (vagy akár úgynevezett mappaooverlay-t is). Van azonban még két mód a háttértár használatára a konténernek. Első az, hogy magát a `.sif` fájlt úgy hozzuk létre, hogy a szükséges adatok, amiket kizárólag olvasni szeretnénk már benne legyenek. A második lehetőség, hogy egy overlay `.img` kiterjesztésű fájlt beleágyazunk a `.sif`-be, viszont ez a művelet nem visszavonható. Mint látható, a beágyazott overlay ugyanúgy, sőt néhány %-al kicsit jobban teljesített, mint ha a natív operációs rendszer ext4 fájlrendszerén végeztem volna a műveleteket, viszont, ha magából a `.sif`-ből olvastam (írásról itt nem beszélhetünk), akkor 60+% teljesítményromlást kaptunk a gazdagépen végzett művelethez képest, ezért érdemes inkább a beágyazott overlay-t használni, ha csak egyetlen fájlként szeretnénk kezelni a konténert. (Viszont, ha csak

gcc-vel szeretnénk lefordítani egy fájlt, nem tapasztalunk különbséget, mivel nincs sok olvasás, tehát ezeket nyugodtan fel lehet továbbra is létrehozáskor tenni.)

Végső konklúzió az, hogy a konténer már az előző benchmarktípusokhoz képest jobban végezte el a szekvenciális olvasást és másolást, viszont valamivel alulteljesített az írásban, viszont az eltérések nem nagyon voltak nagyobbak, mint 10% (nem feltétlen egy újonnan telepített natív vas javára), így kimondhatjuk, hogy itt sincs jelentős teljesítményjavulás- vagy csökkenés. A tényleges háttértár helyett /dev/shm memóriabeli fájlrendszer viszont még mindig külön figyelmet érdemel, és érdemes használni, különösen, ha rendelkezésre áll elég RAM vagy nem a legkorszerűbb SSD meghajtók állnak rendelkezésünkre.

3.1.5. Adatbázis mérések és webserver tesztelések eredményei

Az adatbázis szerves része szinte minden rendszer működésének, így nem hagyhatjuk el a konténerizációs használata, megoldásai és teljesítménymutatói megvizsgálását. Egy régebbi kisebb csoport-projectemet használom ehhez, ami megtalálható a <https://github.com/javajedik/mnist> oldalon. Az adatbázist és a backendet én írtam benne, de sokat tettem hozzá a frontendhez is. A használt szoftverek a Debian 12.4-6.1 MariaDB 11, Spring-boot 3, Maven 3.8.7, Openjdk-17, React voltak. Hitelesítésre JWT tokeneket, adatbázis-kapcsolatra JDBC-t használtam natív SQL parancsokkal. A MariaDB egy egyre szélesebb körnek örvendő MySQL fork, ami garantálja nekünk, hogy a project nyílt forráskódú és ingyenes marad, és többnyire a MySQL fejlesztői dolgoznak tovább rajta. A parancsok, syntax, működés teljesen megegyezik vele és kompatibilisek egymással. Előnye, hogy ingyenesen több szálon végez műveleteket, nincsenek korlátok adatbázisokra és jól integrálódik Linux rendszerekkel. Indítása systemctl híján jogosultságok beállítása után 'mysqld_safe &' paranccsal működik.

Külön tesztelve SQL parancsokat, 3 tizedesjegyre kerekítve nem tudtam olyan join-t vagy függvényt írni, ami különbséget okozott volna a háttértár-benchmarkokhoz képest a két rendszer között (tesztmappákban és az src-ben van jópár SQL parancs) különböző Linux disztribúciók között sem (Ubuntu 22.04,- 23.10, Debian 12.4). Próbáltam nagyobb, néhány GB méretű adatbázisokon is kereséseket és más műveleteket végrehajtani. Itt jelentősen hatékonyabbnak bizonyultak a keresések, ha jól definiált indexeket és kódoptimalizációt alkalmaztunk. (Az indexekkel ellátott attribútumokat BTREE-be (nem binary tree-be!) rendezi,

ami logaritmikus időre csökkenti a háttértár lapozásainak számát. Fontos, hogy ugyanolyan fájlrendszeren teszteljünk (ext3 vagy ext4), mert ez képek feltöltésénél okozhat jelentős különbségeket. Azt ajánlom, hogy mentse ki a .sql create script-jét az adatbázisnak, generálja le, és a táblákat az ext4-en tárolják (ehhez nem kell telepíteni host-ra semmit). Ha az adatbázis kisebb méretű, használható még a /dev/shm memóriabeli fájlrendszer is.

A webalkalmazáson keresztül teszteltem a futási időket a konténerben és a host-on is. Azonosításhoz JWT tokeneket használt és a bejelentkezéshez és különböző "játék" session-ök leírására és hitelesítésére is ezt használtam. Elég egyszerű, 46 tábla, be kell lépni, játékot indítani, és a képről eldönteni, hogy micsoda. (Az adatszerkezet az adatbázisban elég bonyolult, sok táblával és join-nal definiált, hogy egy képre milyen válaszgűjtemény, típus stb. adható meg, mi a válasz rá meg hasonló metainformációk.) Regisztráltam új játékosokat, beléptem, képeket töltöttem fel, játszottam vendégként (ez nem azt jelenti, hogy nincs hitelesítve), regisztrált játékosként, beléptem jelszóval (só+hash módszerek alkalmazásával, nem nyílt jelszótárolásként). Feltöltöttem egyszerre 1-5-10-100 képet, az eredmények ugyanolyanok lettek, mint a háttértárbenchmarknál, így ezt nem részletezem, ha host ext4-t használt volt a legjobb mindkét esetben (eltekintve a dev/shm-től). Különbségek kereséseknél és beszúrásoknál nem voltak mérhetőek (<0,001 sec eltérések). A játék, amikor elindítjuk, hitelesíti a felhasználót, aztán keres képeket az adatbázisban, ami megfelelő adott játéktípusnak / válasznak, megpróbálja felépíteni a képet, és ha jó, egy speciális adatszerkezetben BASE64-ben elküldi, amit a React feldolgoz. 5 darab 4k, 4MB természetképnél is mindkettőt egy másodperc tört része alatt megcsinálta, és feldolgozásra átadta a React-nek, ami megjeleníti. A képek az adatbázisban 'LONGBLOB' típusban külön táblában vannak tárolva natív bináris (és nem BASE64 kódolásban!). Lehetne a háttértáron is, de arra már van benchmark.

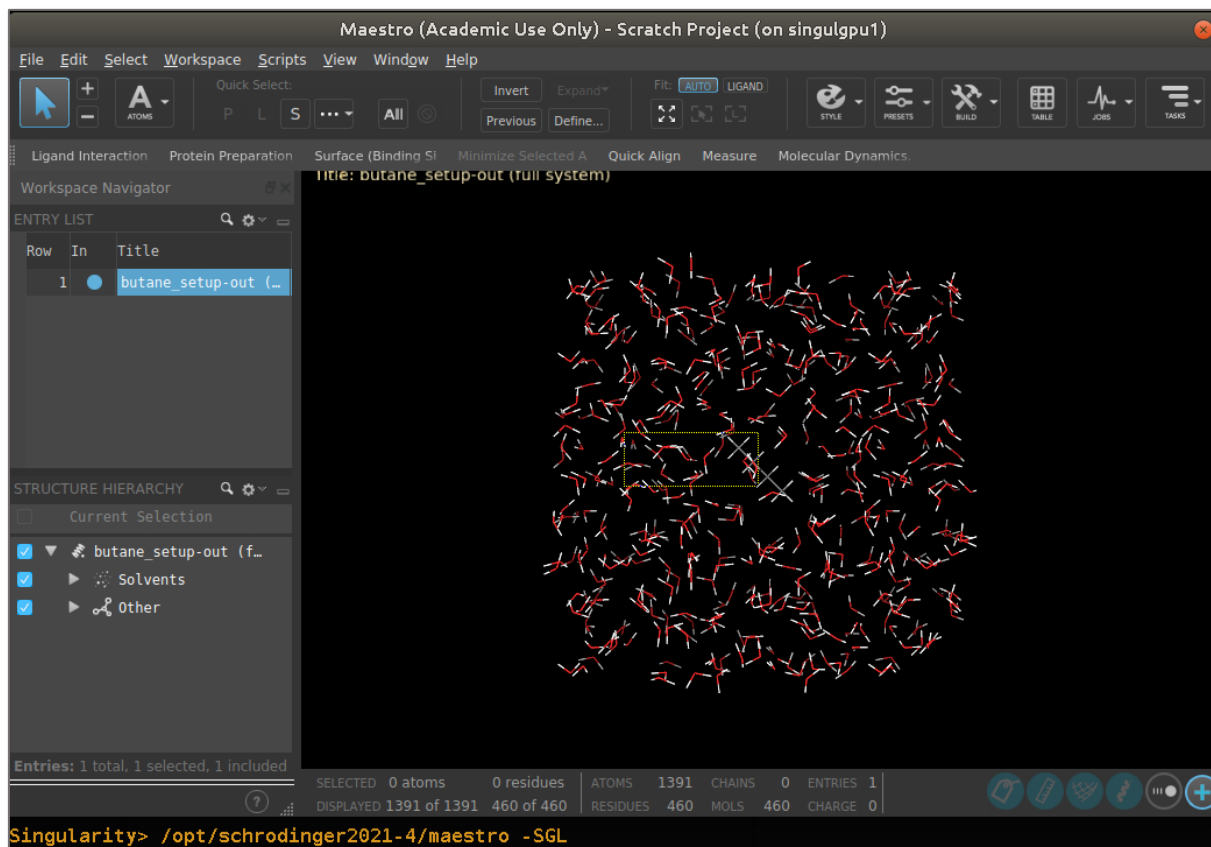
Miután szereztünk némi gyakorlatot az adatbázis kezeléséről és konfigurációjáról, nagyon jól használhatókká és teljesítményorientálttá válnak Singularity konténerekben, amit máshol is érdemes lehet használni, nem csak tudományos számításokhoz a HPC-n.

3.1.6. Videókártyát használó szoftverek a konténerben

Singularity és a Docker is támogatják a videókártya használatát például egy tanításhoz. Jelen állás (2024) szerint a Cuda-t használó API-k mint a Torch kizárólag Nvidia kártyákon képesek működni, remélhetőleg ez a jövőben változik. Singularity erre a `-nv`, Docker pedig a `-gpu` kapcsolót használja, ha több videókártya elérhető, n darabot oda tudunk neki adni, alapértelmezetten mindet felcsatolja. **A konténerbe nem kell telepíteni semmilyen Nvidia illesztőprogramot vagy nvcc-t. Ez közvetlenül a host illesztőprogramjait és Nvidia szoftvereit használja**, figyeljünk a beállított környezeti változókra. Onnantól kezdve funkcionalításban nincs különbség a host és a konténer között, mivel közvetlen erőforrás-átadásról beszélhetünk.

A Képképző Klinika több képfragmentációs és más képfeldolgozó szoftvereket használ, amihez szükség van (akár több) videókártyára is. Ilyenek például az Nnnet, Freesurfer, Moose, Torch, Desmond etc. (Ezeket a szoftvereket nem kizárólagosan erre lehet használni.)

Említettem, hogy a konténer képes a grafikus kimenetet a hostra továbbítani, a lenti 3. ábrán a Desmond grafikus felületét láthatjuk, amikor egy Azure T4-GPU gépen SSH-n keresztül megnyílik nekünk. A hostra nincs telepítve az alkalmazás.



3. Ábra: A Desmond grafikus kimenete a konténerből SSH -X forwardinggal a távoli gépről átirányítva a helyi gépre.

Teljesítménymutatókat tekintve éles adatokon vannak már használva már egy éve, kompatibilitási és számítási hibák nélkül. Ugyanazon az Azure gépen futtatva konténerizációval és anélkül 1-2% teljesítménybeli csökkenést vettek észre, aminek oka a minimális overhead, viszont a natív erőforrások használata. HPC-n a 40 GB A100-on ~2x gyorsabban futottak a szoftverek a konténerben, mint az Azure gépeken a T4-GPU használata mellett ugyanúgy 128 GB rammal konténer használata nélkül. HPC-n natívan nem lehet tesztelni, mert nem telepítenek fel szoftvereket, ami érthető is. Ma (2024) csak konténerizálva használják már ezeket a gyors root nélküli telepítés, hordozhatóság és a kompatibilitás miatt. Szemléltetés képpen ha egy job egy óráig fut, ez kevesebb, mint egy percet jelent.

3.2. Hivatalos benchmark eszközök eredményei (L2)

Az L2 szinten az eredményeimet a 2.3-as fejezetben bemutatott jól ismert benchmark eszközökkel értem el. Mivel nincs elegendő hely, hogy az összes eredményt figyelembe vegyük, itt csak a legérdekesebb, de még mindig részletes eredményeket mutatom be.

3.2.1. CPU-intenzív benchmarkok egy szálon (L2)

Azoknak a CPU benchmarkoknak az aránya, ahol a különbség a natív és a konténeres környezet eredményei között 1% vagy annál kisebb, 82.5% (ezen összefüggésben a különbség azt jelenti, hogy arányosan mennyivel volt jobb az egyik a másikhoz képest). Ezeket a különbségeket hibahatáron belülnek tekintjük, és a benchmarkok 42,42%-ában a konténer nyert. Ahol a natív és a konténer eredmények közötti különbség 1% -nál nagyobb volt, a következő sorrendben vannak a különbségek növekvő sorrendben: 1,33%, 1,33%, 2,08%, 2,17%, 3,43%, 4,54% és 14,33%. Látható, hogy ezek a különbségek 5% -nál kisebbek, kivéve egy esetet, amely 14,33%. Az ezeket az eltéréseket produkáló benchmarkokat kétszer futtattam újra. Ezek az újrafuttatások a legtöbb esetben kevesebb, mint 1% -os különbséget eredményeztek a környezetek között, és a használt benchmarkok nem egyértelműen ítélték meg egy adott környezettípust gyorsabbnak. Az egyetlen kivétel a 7-Zip tömörítési benchmarkja volt, amely minden esetben azt mutatta, hogy a natív környezet teljesít jobban, de csak 0,68% és 1,25% -kal (az eredeti futtatás 3,43% volt).

Ezért a következtetés az, hogy az összes különbség ezekben a benchmark eredményekben hibahatáron belül van, és a benchmarkok nem tudták szignifikánsan megkülönböztetni a két környezetet.

3.2.2. CPU-intenzív benchmarkok több szálon (L2)

Az (L2) szinten ismét a hivatalos CPU-benchmarkokat tekintjük, hogy megtudjuk, másokhoz képest mi milyen eredményeket kapunk. Ahol lehetséges volt, megnéztem az eredményeket 1, 5, 10, 20 szála is. A „alacsony, gépi szintű” többszálúsításról az (L1) részben hosszabban lehet olvasni. Ennek a szintnek a lényege viszont már nem az, hogy operációs rendszer szinten megmondjuk a gépnek, hogy milyen elemi műveleteket végezzen el és pontosan hogyan ossza

ki a szálakat a folyamatok között, hanem konkrét, „értelmes”, összetettebb feladatokat hajtson végre.

Például Bayes-féle valószínűségi mátrixfaktorizáció algoritmus (bpmf) párhuzamosítása, bonyolult matematikai műveletek végrehajtása (Intel MKL tesztek), a π első 5 milliárd (igen, annyi, ehhez ~30 giga memória is kell) kiszámítása (y-cruncher, amelynek van stressztesztje is), párhuzamos tömörítő algoritmusok futtatása (pixz és a pbzip2), a standardnak tekinthető stressztesztek és mérések, amik a memóriát is figyelembe veszik (sysbench, geekbench, y-cruncher), és végül a HPC-re tervezett processzorkommunikációs MPI benchmarkok (IMB, MiniFE, HPCG), amik többek között mérik, hogyan, mekkora csomagmérettel és milyen gyorsan tudnak kommunikálni különböző szálak egymással akár különböző magokon (egy magban több szál is lehet) vagy akár eltérő (fizikai) processzorokon (többprocesszorral rendelkező szuperszámítógépeknél) is, ezen kívül a memóriát is tesztelik.

Az egész gép tesztelésére még jól használható a UserBenchmark is, de ez inkább magát a hardver állapotát teszteli és veti össze másokéval, és inkább Windows-os alternatíva.

Tömörítő algoritmusok

A pixz és a pbzip2 is párhuzamos tömörítő algoritmusok. Tesztelésekből kiderült, hogy nagyobb méretű fájlnál a host és a konténer között nem állapítható meg különbség, hol az egyik volt a gyorsabb, hol a másik, a különbség nem érte el az 1%-ot. Kis fájlra vetítve pedig mindkettő emberi érzékel nem mérhető idő alatt tömörített, a futási időkülönbségek viszont arányaiban nagyok voltak, de egyik összességében itt sem volt egyik jobb a másiknál, a különbségeket inkább az operációs rendszer folyamatai okozhatták.

A π értékét kiszámoló y-cruncher

Option	Decimal Digits	Approx. Memory Needed
1	25,000,000	738 MiB
2	50,000,000	757 MiB
3	100,000,000	913 MiB
4	250,000,000	1.30 GiB
5	500,000,000	2.43 GiB
6	1,000,000,000	4.79 GiB
7	2,500,000,000	11.2 GiB
8	5,000,000,000	23.1 GiB
9	10,000,000,000	46.2 GiB
10	25,000,000,000	114 GiB
11	50,000,000,000	229 GiB
12	100,000,000,000	458 GiB
13	250,000,000,000	1.12 TiB
14	500,000,000,000	2.24 TiB
15	1,000,000,000,000	4.48 TiB
16	2,500,000,000,000	11.3 TiB

4. Ábra: A π első 5 000 000 000 számjegyét kiszámoló y-cruncher

A y-cruncher konzolalkalmazás képes kiszámolni a π -nek akár a 2.5 trilliárdadik számjegyét is, ha van elég számítási kapacitásunk és memóriánk. Mi ebből az első 5 milliárdot számoltattuk ki 1 és 20 szál alkalmazásával is (lásd 4. Ábra). Ne felejtjük el, hogy a CPU-szál nem feltétlen egy feladattal foglalkozik kizárólag. Az eredmények egy szálon az 8. Táblázatban olvashatóak voltak a host és a konténer esetében.

y-cruncher (1 Szál)	Natív	Konténer	Hatékonyság (%)
Teljes számítási idő	2402,392s	2391,293s	100,46%
CPU használat	95,00%	95,00%	100,00%
Kernel overhead	0,00%	0,00%	100,00%

8. Táblázat: π 5 milliárd jegyének számítása

Ahogy a 8. Táblázatból olvasható, ha csak egyetlen szálon számoltuk ki a π -t, akkor sem a processzoridő-kihasználásban, sem a kernel overheadben nincs különbség (2 tizedesjegyre kerekítve). A konténer hibahatáron belül, de gyorsabbnak bizonyult.

Nézzük meg ugyanezt a feladatot, ha többszálúsítunk 20 szálra. Tekintsük a 9. Táblázatot.

y-cruncher (20 Szál)	Natív	Konténer	Hatékonyság (%)
Total Computation Time	543,498s	545,620s	99,61%
CPU Utilization	1472,04%	1488,11%	101,09%
Kernel overhead	0,43%	0,45%	95,56%
Multi-core Efficiency:	73,60%	74,41%	101,10%
Kernel overhead	0,02%	0,02%	100,00%

9. Táblázat: π 5 milliárd jegyének számítása 20 szálon

Láthatólag itt sincsenek futási időbeli különbségek, ami viszont érdekes, hogy a konténer még a nagyobb kernel-overhead-del együtt is kicsit hatékonyabban tudta használni a feladatra kapott processzoridejét. (A processzoridő és a valós idő két külön dolog, amiről az (L1) szinten már beszéltünk a többszálúsításnál.)

Ezzel a méréssel igazolódni látszik az (L1) szinten felvetett teóriák, mik szerint az 1 szála vetített szekvenciális végrehajtáskor ~95-96% processzoridő használati hatékonyság van, ami párhuzamosításkor szálonkénti ~70-75%-os hatékonysági mutatóra csökken (ami olyan, mintha 20 szálból ~15 dolgozna 100% hatékonysággal szekvenciálisan), valamint az is, mivel a szálak nem csak az adott egy folyamatot szolgálják ki, így a futási idő 20 szál esetében nagyságrendileg csak négyszer lesz gyorsabb. Ennek előnyeiről és hátrányairól már (L1) szinten beszéltünk.

Stresszteszt benchmarkok

Az első esetben nézzük meg az előző y-cruncher benchmark stressztesztjét. Ezekben minden processzorszál (20) leallokál magának 1,38 GB memóriát, és egymás után 6 különböző matematikai mérést futtatnak újra és újra. Az eredményeket itt foglaltam össze:

NATÍV	KONTÉNER
Iter.: 0 Total Time: 6.326s (0.105m) BKT: Passed Test Time: 120.001s (2.000m) BBP: Passed Test Time: 143.967s (2.399m) SFT: Passed Test Time: 120.001s (2.000m) N64: Passed Test Time: 167.721s (2.795m) HNT: Passed Test Time: 126.221s (2.104m) VST: Passed Test Time: 149.027s (2.484m)	Iter.: 0 Total Time: 7.030s (0.117m) BKT: Passed Test Time: 120.092s (2.002m) BBP: Passed Test Time: 129.280s (2.155m) SFT: Passed Test Time: 120.036s (2.001m) N64: Passed Test Time: 134.443s (2.241m) HNT: Passed Test Time: 129.529s (2.159m) VST: Passed Test Time: 132.561s (2.209m)
Iter.: 1 Total Time: 833.264s (13.888m) BKT: Passed Test Time: 120.001s (2.000m) BBP: Passed Test Time: 143.924s (2.399m) SFT: Passed Test Time: 120.000s (2.000m) N64: Passed Test Time: 165.411s (2.757m) HNT: Passed Test Time: 144.101s (2.402m) VST: Passed Test Time: 148.697s (2.478m)	Iter.: 1 Total Time: 772.972s (12.883m) BKT: Passed Test Time: 120.059s (2.001m) BBP: Passed Test Time: 126.977s (2.116m) SFT: Passed Test Time: 120.015s (2.000m) N64: Passed Test Time: 138.363s (2.306m) HNT: Passed Test Time: 132.340s (2.206m) VST: Passed Test Time: 134.032s (2.234m)
Iter.: 2 Total Time: 1675.398s (27.923m) BKT: Passed Test Time: 120.001s (2.000m) BBP: Passed Test Time: 143.931s (2.399m) SFT: Passed Test Time: 120.001s (2.000m) N64: Passed Test Time: 166.847s (2.781m) HNT: Passed Test Time: 139.021s (2.317m) VST: Passed Test Time: 148.716s (2.479m)	Iter.: 2 Total Time: 1544.757s (25.746m) BKT: Passed Test Time: 120.053s (2.001m) BBP: Passed Test Time: 125.877s (2.098m) SFT: Passed Test Time: 120.063s (2.001m) N64: Passed Test Time: 135.968s (2.266m) HNT: Passed Test Time: 132.174s (2.203m) VST: Passed Test Time: 121.229s (2.020m)
Iter.: 3 Total Time: 2513.915s (41.899m) BKT: Passed Test Time: 120.001s (2.000m) BBP: Passed Test Time: 143.915s (2.399m) SFT: Passed Test Time: 120.001s (2.000m) N64: Passed Test Time: 166.100s (2.768m) HNT: Passed Test Time: 141.904s (2.365m) VST: Passed Test Time: 148.120s (2.469m)	Iter.: 3 Total Time: 2300.121s (38.335m) BKT: Passed Test Time: 120.009s (2.000m) BBP: Passed Test Time: 126.817s (2.114m) SFT: Passed Test Time: 120.015s (2.000m) N64: Passed Test Time: 133.494s (2.225m) HNT: Passed Test Time: 133.324s (2.222m) VST: Passed Test Time: 132.052s (2.201m)
Iter.: 4 Total Time: 3353.956s (55.899m) BKT: Passed Test Time: 120.001s (2.000m) BBP: Passed Test Time: 143.903s (2.398m) SFT: Passed Test Time: 120.001s (2.000m) N64: Passed Test Time: 166.429s (2.774m) HNT: Passed Test Time: 139.446s (2.324m) VST: Passed Test Time: 148.725s (2.479m)	Iter.: 4 Total Time: 3065.832s (51.097m) BKT: Passed Test Time: 120.030s (2.000m) BBP: Passed Test Time: 127.104s (2.118m) SFT: Passed Test Time: 120.068s (2.001m) N64: Passed Test Time: 138.826s (2.314m) HNT: Passed Test Time: 131.028s (2.184m) VST: Passed Test Time: 133.726s (2.229m)
Iter.: 5 Total Time: 4192.460s (69.874m) BKT: Passed Test Time: 120.001s (2.000m) BBP: Passed Test Time: 143.950s (2.399m) SFT: Passed Test Time: 120.001s (2.000m) N64: Passed Test Time: 167.614s (2.794m) HNT: Passed Test Time: 126.244s (2.104m) VST: Passed Test Time: 149.008s (2.483m)	Iter.: 5 Total Time: 3836.614s (63.944m) BKT: Passed Test Time: 120.024s (2.000m) BBP: Passed Test Time: 127.116s (2.119m) SFT: Passed Test Time: 120.028s (2.000m) N64: Passed Test Time: 139.608s (2.327m) HNT: Passed Test Time: 128.402s (2.140m) VST: Passed Test Time: 129.716s (2.162m)

Röviden a mérésekről:

- BKT : Basecase algoritmus alapú szorzás plusz Karatsuba algoritmus alapú szorzás
- BBP : A Bailey–Borwein–Plouffe formulán alapuló számjegy kinyerő algoritmus
- SFT : Kicsi, cache-en belüli gyors Fourier transzformáció (FFT)
- FFT : gyors Fourier transzformáció (A Fourier-analízis a jelet az eredeti tartományból (gyakran idő vagy tér) a frekvenciatartományban történő ábrázolásra alakítja át, és fordítva.)
- N32 : 32 bites számelméleti transzformáció (Number-Theoretic Transform (NTT)) ami 31 bites prímekeket használ
- N64 : 64 bites számelméleti transzformáció (Number-Theoretic Transform (NTT), a lebegőpontos FFT egy alternatívája) ami 63 bites prímekeket használ
- HNT : Hybrid NTT. A hibrid NTT 2008-ban jött létre, amikor megpróbáltak egy olyan algoritmust találni, amely középutat jelent az FFT és az NTT közötti szörnyű tér-idő kompromisszumban.
- VST : Vector-Scalable Transform (VST), azaz Vektor-Skálázható Transzformáció. A VST algoritmus egy ostoba kísérlet volt a SIMD-vel 2011-ben, amely végül egy félig tisztességes lemezcsereelő algoritmussá nőtte ki magát a swap üzemmódú számításokhoz.
- C17 : A C17 algoritmus egy névtelen algoritmus, amelyet 2016 elején fejlesztettek ki és adtak hozzá az y-cruncher v0.7.1-hez.

A BKT és SFT benchmarkok stabilan ugyanazt az eredményt hozták mindkét esetben, a HNT-nél viszont néhány % fölényt szerzett a konténer, a maradék háromnál viszont elsőprő előnnyel veszített a host. Persze, ez csak egy benchmark a sok közül, és lehet, hogy máshol a konténer fog jelentősen rosszabbul teljesíteni, ezért érdemes többet is megnézni, mielőtt döntést hozunk.

Másodikként nézzük meg a sysbench-et. A program lényege, hogy párhuzamosítva keres 1 és 100 millió között prímekeket. Hogy árnyaltabb képet kapjunk az eredményekről, ezt lefuttatásra került 1, 5, 10 és 20 processzorszálon is. Fontos, hogy a skálázás úgy történik, hogy annyiszor

keresi meg a prímeket, amennyi a bemeneti szálak száma. (Tehát, 20 szálon 20-szor annyi az elvégzendő feladat is.) Ezért jól össze lehet hasonlítani, hogyan változik a teljesítmény, ahogy egyre több szálát használunk. Elég nagyok az eltérések, ezért sokszor lettek újratestelve. Vessünk egy pillantást a 10. Táblázatra.

sysbench (primes up to 100 million)	Natív (s)	Konténer (s)	Hatékonyság (%)	Futásidő hatékonyság (%)
Szálak és job-ok száma: 1	122,8089	122,774	100,03%	100,03%
Szálak és job-ok száma: 5	179,7156	159,7047	112,53%	76,90%
Szálak és job-ok száma: 10	198,2941	206,3414	96,10%	59,52%
Szálak és job-ok száma: 20	316,4789	313,1945	101,05%	39,21%

100. Táblázat A sysbench eszközzel kapott eredmények 1, 5, 10 és 20 szálra

Mint látható, ha 1 szálon futtattunk, akkor nem volt különbség a teljesítményben, ha 5-ön, akkor a konténer szerzett magának 12,53% előnyt, amit el is veszített, ha már 10 szálát használtunk, majd újra kiegyenlítődték az erőviszonyok, ha már 20 szál volt alkalmazva. Ennek egyik mondanivalója az, hogy a teljesítmény eltérő lehet attól függően, mennyi szálát használunk, viszont ahogy egyre több kerül használatra, az ezekből származó különbségek egyre jobban eltűnnek. Másik érdekesség, hogy a szálak növelésével romlottak a valós időben vett futási idők arányai az 1 szálon való futtatáshoz képest (mert a feladat mennyisége is lineárisan skálázva lett), ami ennél a mérésnél azt jelenti, hogy minél több processzorszálát használunk fel, annál többet kellett fizetnünk a párhuzamosításért. Ténylegesen felhasznált processzoridőt ez a benchmark nem mért. Tehát, ha 5 job 5 szálon 1 perc alatt fut le, akkor 100 job 100 szálon egyáltalán nem biztos, hogy szintén 1 perc alatt végez, tapasztalat szerint több (valós) időre lesz szükség. A többi benchmarkból kiderült, hogy a tényleges processzoridőt tekintve konstans a 70-75% hatékonysági mutató szálak számától függetlenül, de a CPU más folyamatokkal is foglalkozik.

A geekbench egy hardvert felmérő összetettebb benchmark, elég sok mindent mér. A kapott eredményeket a 5. Ábrán mutatom be.

Single-Core Performance	Native	Container	Multi-Core Performance	Native	Container
Single-Core Score	1548	1541	Multi-Core Score	12837	12529
Crypto Score	3529	3524	Crypto Score	10865	11199
Integer Score	1363	1362	Integer Score	12816	12575
Floating Point Score	1617	1599	Floating Point Score	13211	12650
AES-XTS	3529 6.02 GB/sec	3524 6.01 GB/sec	AES-XTS	10865 18.5 GB/sec	11199 19.1 GB/sec
Text Compression	1223 6.19 MB/sec	1219 6.17 MB/sec	Text Compression	11889 60.1 MB/sec	12081 61.1 MB/sec
Image Compression	1405 66.4 Mpixels/sec	1393 65.9 Mpixels/sec	Image Compression	14678 694.4 Mpixels/sec	14277 675.4 Mpixels/sec
Navigation	1264 3.56 MTE/sec	1269 3.58 MTE/sec	Navigation	7477 21.1 MTE/sec	7346 20.7 MTE/sec
HTML5	1550 1.82 MElements/sec	1554 1.83 MElements/sec	HTML5	15401 18.1 MElements/sec	15689 18.4 MElements/sec
SQLite	1350 422.9 Krows/sec	1352 423.4 Krows/sec	SQLite	16473 5.16 Mrows/sec	13474 4.22 Mrows/sec
PDF Rendering	1391 75.5 Mpixels/sec	1389 75.4 Mpixels/sec	PDF Rendering	15197 824.7 Mpixels/sec	14881 807.6 Mpixels/sec
Text Rendering	1411 449.6 KB/sec	1410 449.2 KB/sec	Text Rendering	11080 3.45 MB/sec	10675 3.32 MB/sec
Clang	1392 10.8 Klines/sec	1393 10.9 Klines/sec	Clang	14004 109.1 Klines/sec	14801 115.3 Klines/sec
Camera	1311 15.2 images/sec	1303 15.1 images/sec	Camera	11954 138.6 images/sec	12483 144.7 images/sec
N-Body Physics	1538 1.92 Mpairs/sec	1545 1.93 Mpairs/sec	N-Body Physics	11801 14.8 Mpairs/sec	13038 16.3 Mpairs/sec
Rigid Body Physics	1436 8896.6 FPS	1437 8902.4 FPS	Rigid Body Physics	16611 102909.2 FPS	15028 93101.3 FPS
Gaussian Blur	1299 71.4 Mpixels/sec	1280 70.4 Mpixels/sec	Gaussian Blur	13169 723.9 Mpixels/sec	13382 735.6 Mpixels/sec
Face Detection	1328 10.2 images/sec	1310 10.1 images/sec	Face Detection	14578 112.2 images/sec	13604 104.7 images/sec
Horizon Detection	1188 29.3 Mpixels/sec	1235 30.4 Mpixels/sec	Horizon Detection	12887 317.7 Mpixels/sec	13324 328.4 Mpixels/sec
Image Inpainting	2999 147.1 Mpixels/sec	3009 147.6 Mpixels/sec	Image Inpainting	22043 1.08 Gpixels/sec	21284 1.04 Gpixels/sec
HDR	2964 40.4 Mpixels/sec	2930 39.9 Mpixels/sec	HDR	28496 388.3 Mpixels/sec	30225 411.9 Mpixels/sec
Ray Tracing	2095 1.68 Mpixels/sec	2091 1.68 Mpixels/sec	Ray Tracing	21743 17.5 Mpixels/sec	22740 18.3 Mpixels/sec
Structure from Motion	1363 12.2 Kpixels/sec	1210 10.8 Kpixels/sec	Structure from Motion	13219 118.4 Kpixels/sec	9679 86.7 Kpixels/sec
Speech Recognition	1409 45.0 Words/sec	1402 44.8 Words/sec	Speech Recognition	4920 157.3 Words/sec	5130 164.0 Words/sec
Machine Learning	1223 47.3 images/sec	1219 47.1 images/sec	Machine Learning	4966 191.9 images/sec	3844 148.5 images/sec

5. Ábra: A geekbench eszközzel kapott eredmények

Ami számunkra érdekesek lehetnek, az az első négy sor, pontosítva az egy szálon végzett műveletekre kapott pontszám, a több szálon végzett műveletekre kapott pontszám, az egész számokkal végzett műveletek hatékonysága, illetve a lebegőpontos számokra kapott értékelés. A tesztet többször le lett futtatva, hasonló eredményeket adott.

Az egy szálon végzett műveletekre a natív operációs rendszer majdnem teljesen megegyező értéket kapott a konténerrel (+-10 pont). Többször lefuttatva a tesztet, alkalmanként konténer meg is előzte ezt az értéket. Az egész számokkal végzett műveletek hatékonysága megegyező lett, míg a lebegőpontos számokkal végrehajtottak már 1% körüli lemaradást mutattak a konténernél. Ha több szálon próbáltam ki a tesztelést, a host nagyjából 2,5%-kal gyorsabb volt. Összegezve, különbség csak többszálúsításnál volt tapasztalható, ott is csak 2,5%, ami bár a host számára kedvez, de ez túl kevés ahhoz, hogy számottevő teljesítménymúlásról beszéljünk. A sub-benchmarkok eredményeinek az elemzését az olvasóra bízom.

Intel MKL teszt

A LINPACK az Intel MKL-hez egy szoftverkönyvtár numerikus lineáris algebra végrehajtására. Ennek a bemeneti méretei lettek skálázva, illetve a tesztek többször megismételve, ahogy a 11. Táblázatban látható.

Konténer

Size	LDA	Align.	Average	Maximal
1000	1000	4	191.9486	207.9786
2000	2000	4	276.1136	284.0663
5000	5008	4	399.2120	405.7920
10000	10000	4	415.3681	423.9207
15000	15000	4	288.2683	402.7503
18000	18008	4	203.9897	204.9863
20000	20016	4	208.3200	208.5610
22000	22008	4	211.4553	211.8339
25000	25000	4	215.4127	216.0814
26000	26000	4	215.4725	216.0296
27000	27000	4	217.2249	217.8598
30000	30000	1	218.2924	218.2924
35000	35000	1	216.9466	216.9466
40000	40000	1	221.5451	221.5451
45000	45000	1	222.7384	222.7384

Natív

Size	LDA	Align.	Average	Maximal
1000	1000	4	194.8350	214.6645
2000	2000	4	277.9042	299.8725
5000	5008	4	413.3630	424.3255
10000	10000	4	430.2958	441.7517
15000	15000	4	289.3199	408.4344
18000	18008	4	202.2485	202.6580
20000	20016	4	208.2240	208.8331
22000	22008	4	211.1864	212.4889
25000	25000	4	214.9464	215.8881
26000	26000	4	215.9950	216.5673
27000	27000	4	215.2409	218.0042
30000	30000	1	218.7809	218.7809
35000	35000	1	217.5006	217.5006
40000	40000	1	220.7107	220.7107
45000	45000	1	222.5293	222.5293

111. Táblázat: Intel MKL LINPACK eszközzel kapott eredmények

A GFlops az a másodpercenként végrehajtott lebegőpontos számításokat jelenti milliárdos nagyságrendben. A tesztekben egyértelműen látszik, hogy a natív operációs rendszer kisebb bemenetekre valamivel jobban teljesített, viszont, ahogy nőtt a számításigény, ez az előny egyre jobban kiegyenlítődött a két rendszer közötti teljesítmény között. Ahogy többször felmértem az alábbi LINPACK benchmarkot, akkor nagyobb bemenetekre felváltva volt első a host és a konténer, mindig 1% hibahatáron belül, ami abból a szempontból megnyugtató lehet, ha hosszas számításokat szeretnénk végezni, mert a kisebbek így is hamar lefutnak.

Bpmf Bayes-féle valószínűségi mátrixfaktorizáció

A bpmf az Julia és C++ implementációk a Bayes-féle valószínűségi mátrixfaktorizációra Markov-lánc Monte Carlo segítségével. A BPMF egy ajánló módszer, amely lehetővé teszi például a filmek értékelésének előrejelzését. Ezek közül a „chembl” bizonyult a leghosszabbnak futási idő szempontjából, itt a konténer 5%-al gyorsabb volt a natív teljesítménnyel összevetve. Viszont „movielens” és a „tiny” bár rövid ideig futottak, harmadával hatékonyabbnak bizonyultak, ha a host futtatta őket.

Intel MPI benchmarkok

Az Intel MPI Benchmarkok egy sor MPI teljesítménymérést végeznek pont-pont és globális kommunikációs műveletekre különböző méretű üzenetekkel. Ez elsősorban inkább HPC tesztelésére, ahol több mag és fizikai processzor van, és nem asztali gép benchmarkolására szolgál.

A MiniFE egy proxy alkalmazás strukturálatlan implicit végeeselemes kódokhoz. Az ezzel lefuttatott benchmark 1%-os teljesítményjavulást jelentett a konténer javára.

A High Performance Conjugate Gradients (HPCG) Benchmark projekt célja egy új mérőszám létrehozása a HPC rendszerek rangsorolásához. A HPCG a TOP500 számítási rendszerek rangsorolására jelenleg használt High Performance LINPACK (HPL) benchmark kiegészítésére szolgál. Az ezzel elvégzett tesztünk eredménye azt mutatta, hogy nincs különbség a konténer és a gazdagép között.

Az IMP a processzek közötti kommunikáció sebességét méri, ami inkább akkor érdekes, ha több processzormagunk van, de mielőtt belemennénk, ismerjük meg az aggregált és a nem aggregált mód közötti különbségeket.

Az MPI (Message Passing Interface) benchmark tesztek során a módok közötti különbség az üzenetküldési protokollban rejlik.

Az aggregált mód során az összes MPI folyamat (processz) a tesztelés során közvetlenül kommunikál az egyes másik folyamatokkal. Ez azt jelenti, hogy ha van 10 folyamat és az egyik folyamatnak üzenetet kell küldenie a másik folyamatnak, akkor az üzenet közvetlenül megy a célhoz vezető úton.

A nem aggregált mód esetén az üzenetek átvitele közvetett módon történik, azaz az üzenetek egy központi helyen kerülnek összegyűjtésre és onnan továbbítják őket a célhoz vezető folyamatoknak. Ez azért hasznos, mert így csökkenthető az üzenetek átvitelének a késleltetése, ami hatékonyabbá teszi a kommunikációt nagy számú folyamat között.

Az aggregált mód használata hatékonyabb lehet kisebb számú folyamatok esetén, míg a nem aggregált mód használata javasolt nagy számú folyamatok esetén. Az optimális üzenetátviteli stratégia azonban függ a tesztelt alkalmazás jellegétől és a rendszer konfigurációjától.

A nem aggregált „Accumulate” és mintkettő „Unidir_Put” eseteiben nem volt teljesítménybeli különbség, viszont az aggregált „Accumulate”, „Window”, aggregált és nem aggregált „Bidir_Put”- és „Bidir_Get” mérések jelentős (10-30%) különbséggel a konténer volt a gyorsabb.

Összegezve elmondható, hogy ezeket a processzkommunikációs méréseket inkább HPC-ra tervezték, nem asztali egy processzoros gépekre, viszont úgy néz ki, egy olyan környezetre és célra kifejlesztett konténer jól teljesít asztali szinten.

3.2.3. MPI Slurm klaszteren 2 compute node-al

A laptopot és az asztali gépet összekötve és 23.02 Slurm klaszterbe húzva lefuttattam újra a korábban említett benchmarkeszközöket, és néhány újjal kiegészítettem azokat a HPCG, OSU eszközökkel. Ilyenkor mindkét rendszer az akkori legújabb 23.10 Ubuntu volt.

Az eredményeket képes illusztrációkkal is meg lehet találni a [27] forrásban (külön köszönet Polgár Péter témavezetőmnek az összegyűjtésében és elemzésében).

Több, mint 2000 fájlt tartalmaz, néhány ábrát (5., 6., 7.) ide is mellékelek.

A Singularity bind és hybrid architektúrát használtam az MPI mérésekhez, a hivatalos dokumentáció a különbséget jól össze is foglalja [26]:

Bind modell:

Előnyök:

1. Integráció a resource managerekkel, például a Slurm-mel.
2. A konténerképek kisebb méretűek, mivel nincs szükség az MPI hozzáadására a konténerekhez.

Hátrányok:

3. A konténerben futó alkalmazás MPI-ja kompatibilisnek kell lennie a host rendszeren lévő MPI verziójával.
4. A felhasználónak tudnia kell, hol van telepítve a host MPI-ja.
5. A felhasználónak biztosítani kell, hogy a könyvtár megkötése lehetséges, ahol a host MPI-ja telepítve van.
6. A felhasználónak biztosítani kell, hogy a host MPI-ja kompatibilis legyen az alkalmazás telepítésére és futtatására szolgáló konténerben használt MPI-vel.

Hibrid modell:

Előnyök:

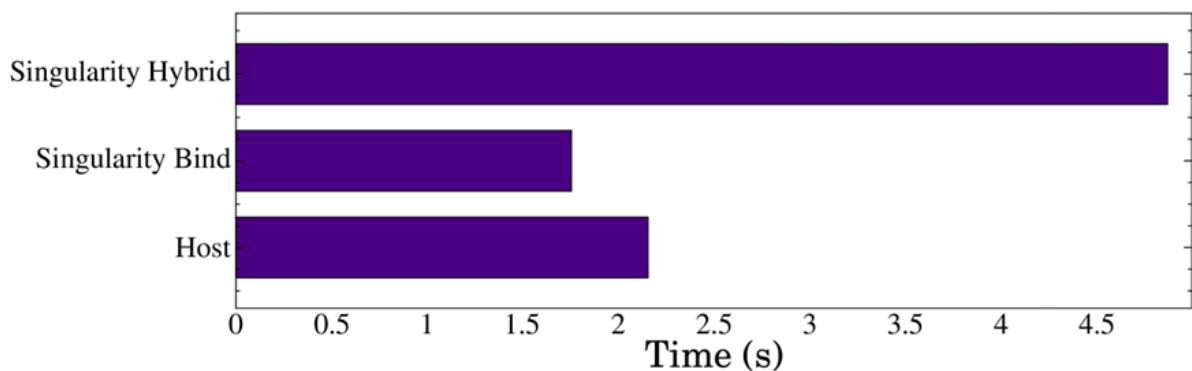
7. Integráció a resource managerekkel, például a Slurm-mal.
8. Egyszerűség, mivel hasonló a natively running MPI alkalmazásokhoz.

Hátrányok:

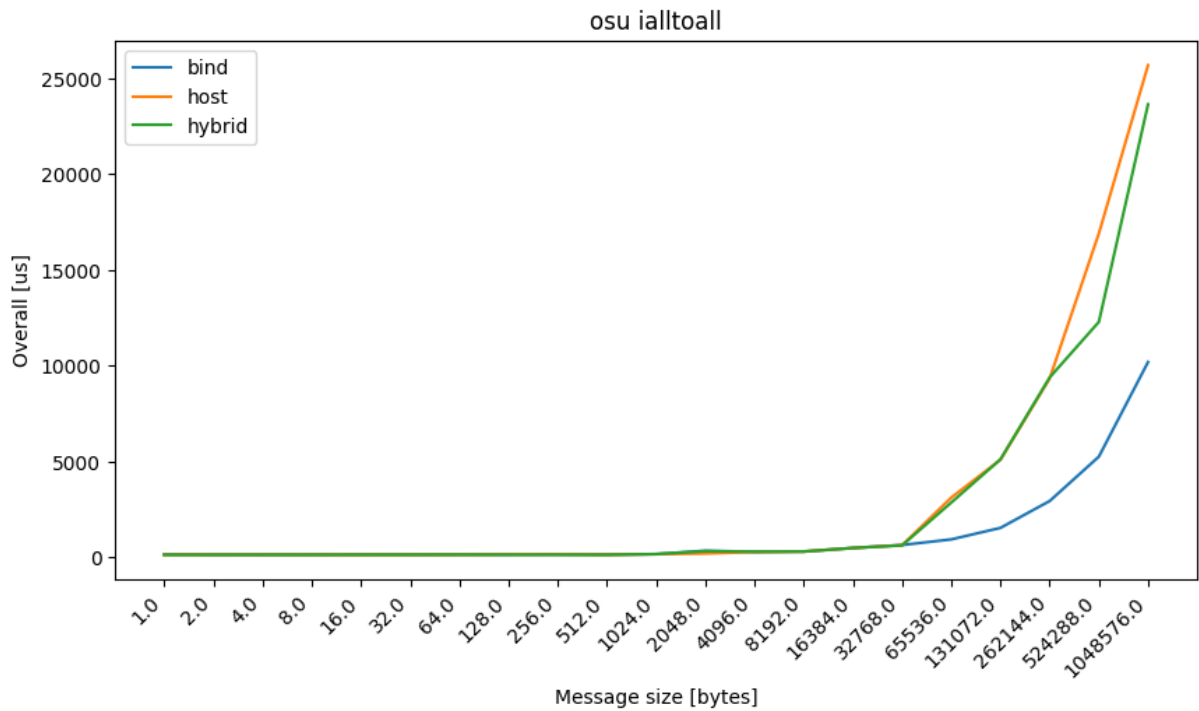
9. A konténerben futó alkalmazás MPI-ja kompatibilisnek kell lennie a host rendszeren lévő MPI verziójával.
10. A konténerben lévő MPI implementációt gondosan be kell állítani az eszközök optimális használatához, ha a teljesítmény kritikus szempont.

Az HPCG benchmark teljesítménymetrikákat szolgáltat az MPI-re vonatkozóan, egyetlen szempont alapján, konkrétan az MPI_Allreduce műveletre. A benchmarkot az összes három MPI modell esetében az alapértelmezett argumentumokkal futtattam. Átlagos értékeket figyelembe véve az MPI_Allreduce művelet a Singularity bind modellnél mutatta a legjobb teljesítményt. Összehasonlításképpen, a gép natív konfigurációja kissé lassabb teljesítményt mutatott, és a Singularity hibrid modell jelentősen lassabb teljesítményt mutatott, mint a másik két konfiguráció, ahogy azt az 6. ábra mutatja. Ezek az átlagos MPI_Allreduce eredmények egybeesnek az HPCG benchmark futási idejével és GFLOP/s értékelésével, jelezve, hogy a Singularity bind modell érte el a legkisebb futási időt és a legmagasabb GFLOP/s értékelést. Arra, hogy a bind modell miért lett gyorsabb egy kevéssel, mint a natív host, nem találtam forrást. Valószínűleg a konténerben mivel kevesebb program van, jóval kevesebb processz fut.

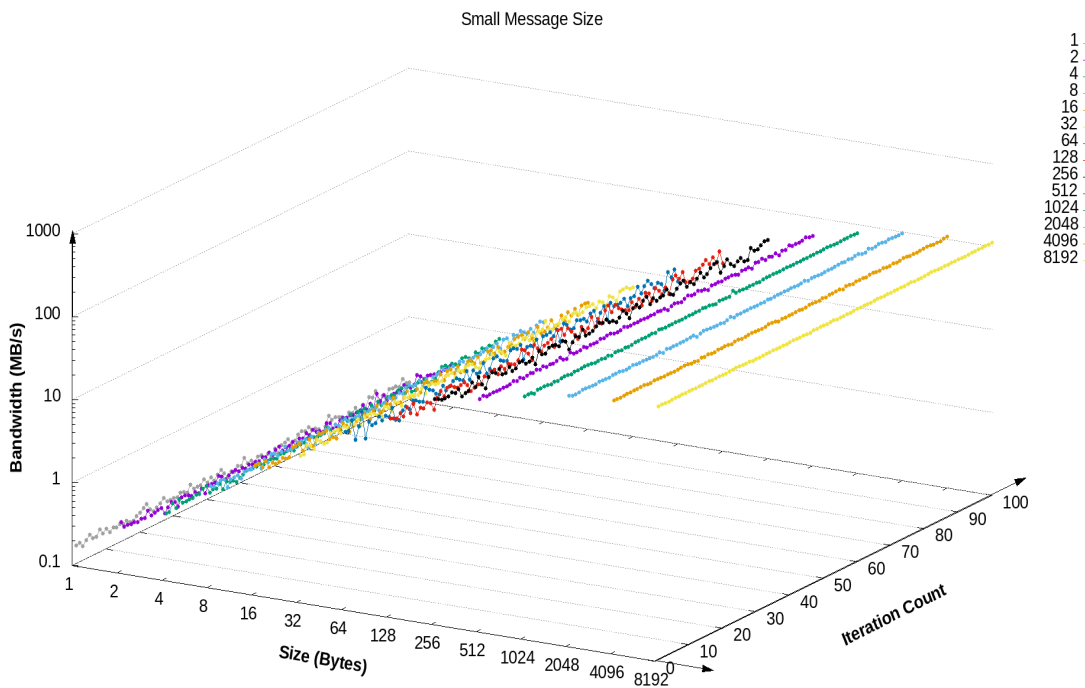
OSU esetében három kategóriájú benchmark futott: collective, point-to-point, és startup. Minden kategóriában az összes benchmark teszt az MPI-vel kapcsolatos argumentumok alapértelmezett értékeivel lett végrehajtva. Minden benchmark egyszer lett végrehajtva minden MPI konfigurációhoz. Ebből rengeteg ábra készült, a mellékletben megtalálhatóak [27]. Két jellemző mérési eredményt mutat be a 7. és a 8. ábra.



6. Ábra: A HPCG futási ideje a három jellemző HPC argumentum mellett.



7. Ábra: Az osu benchmarking futási idői az üzenet méret függvényében a három jellemző HPC argumentum esetén Az "ialltoall" egy MPI (Message Passing Interface) művelet, amely az összes folyamat között adatokat cserél. Ez a művelet arra szolgál, hogy minden résztvevő folyamat elküldje az adatait minden más résztvevő folyamatnak [P2].



8. Ábra: Az Osu BW teszt a hálózati sávszélességet méri, és azt vizsgálja, hogy milyen gyorsan tudnak kommunikálni a rendszerben lévő csomópontok egymással [P2].

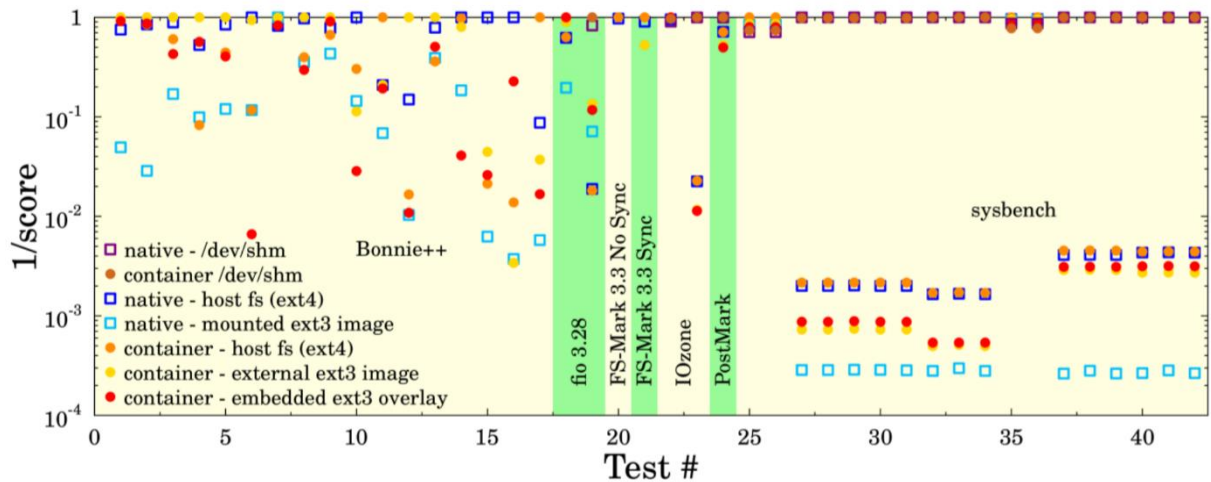
Bár az OSU MPI benchmarkjai az adott méréstől függenek, a legtöbbnél ugyanazokat a függvényeket kapjuk vissza, amit fent láthatunk. Hasonlóan a HPCG-hez, a bind architektúra szinte mindig jelentős különbséggel jobb volt, míg a hybrid architektúra vagy egyenértékű volt a host rendszerrel vagy lassabb. További ábrákért érdemes megtekinteni a [27] forrást.

3.2.4. Memória benchmarkok (L2)

Azon memória benchmarkok aránya, ahol a különbség a natív és a konténer eredményei között 1% vagy annál kisebb, 86,36% (ebben az összefüggésben a különbség azt jelenti, hogy arányosan mennyivel volt jobb az egyik a másikhoz képest). Ezeket a különbségeket hibahatáron belülnek tekintjük, és az esetek 52,63%-ában a konténer nyert. Ahol a natív és konténer eredmények közötti különbség 1%-nál nagyobb volt, az eltérések a következő sorrendben növekednek: 1,17%, 1,21% és 1,45%. Látható, hogy ezek a különbségek kevesebbek mint 2%. A benchmarkok, melyek ezeket a különbségeket okozták, kétszer futtattam újra. Az újrafuttatások két benchmarkban kevesebb mint 1% -os különbséget eredményeztek, és a használt benchmarkok nem egyértelműen döntötték el, hogy melyik környezet típusa a gyorsabb.

Az eredmények különbségei tehát mind hibahatáron belül maradtak, és a konténer győzelmének aránya hasonló volt a natív győzelmének arányához.

3.2.5. Tárolási I/O benchmarkok (L2)



9. Ábra: hivatalos benchmarkok eredményei

Mérési eredményeim alapján az alábbi következtetések állapíthatók meg (lásd 9. ábra):

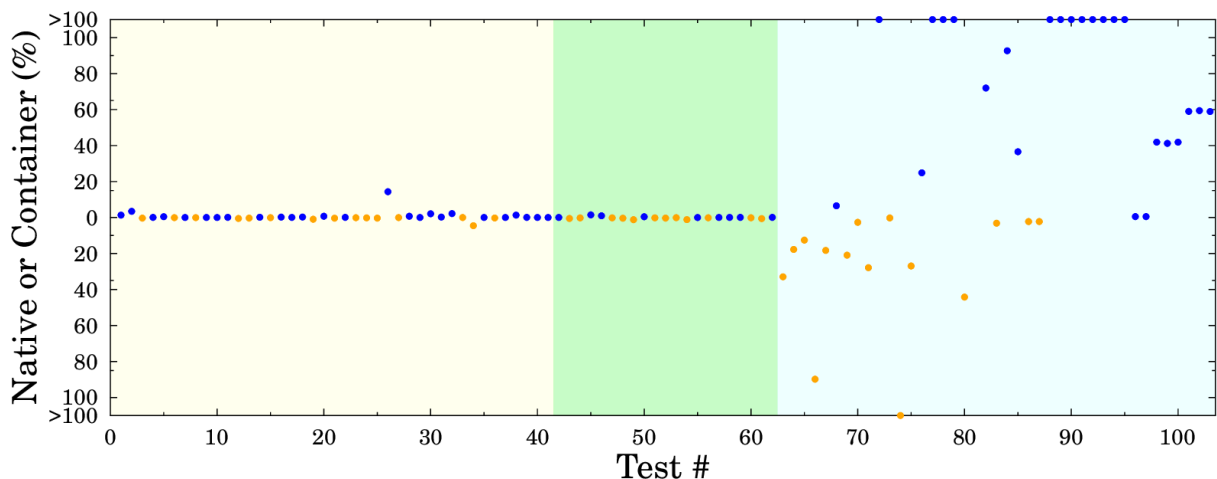
1. A Bonnie++ nem megbízható az egyes rendszerkörnyezetek összehasonlítására.
2. A Fio aszinkron nem pufferelt olvasásánál az ext3 konténer overlay (mind külső, mind beágyazott) nyer az ext4 host és az ext3 natív mountolt image fájlrendszeren alapuló környezetek felett.
3. A Fio fájlvisztaolvasásánál az ext3 overlay (mind a konténer, mind a natív) győz az ext4 host felett.
4. A Fio fájlvisztaolvasását kivéve, mindenben a natív ext4 sokkal jobb volt, mint a natív ext3 képfájl fájlrendszer.
5. A beágyazott és a külső ext3 overlayt összehasonlítva, sysbench random és szekvenciális olvasásban a külső ext3 a sokkal jobb, míg az összes többi sysbench tesztnél a beágyazott ext3 a sokkal jobb.
6. A host ext4 fájlrendszer konténer általi használata FS-Mark esetén, és sysbench-nél több esetben sokkal jobb volt a natív használathoz képest, egyébként nincs jelentős eltérés az eredmények között.
7. A host ext4 a külső és beágyazott ext3 overlayhez képest a legtöbb sysbench, PostMark, egy IOzone és FS-Mark benchmarkok esetében bizonyult sokkal

jobb, míg a konténeres ext3 overlay (külső és beágyazott egyaránt) a Fio tesztekben bizonyult sokkal jobb.

8. /dev/shm natív és konténer általi használata összehasonlítva: a legtöbb esetben marginális volt az eredmények közötti különbség, nagyobb különbség a natív javára a szekvenciális olvasásnál volt, míg a konténer a random olvasás és a fájl visszafelé olvasás esetén volt számottevően jobb.

Mivel mind a natív, mind a konténer környezetben sokféle fájlrendszer lehetőség van, és mindegyiknek megvannak a saját előnyei, további kutatásokra van szükség annak meghatározására, hogy mely natív vagy konténer fájlrendszer lehetőségek vagy ezek kombinációi a legjobbak a leggyakoribb felhasználási esetekhez, különösen HPC környezetekben.

3.2.6. Következtetések (L2)



10. Ábra: Az L2 szinten végzett mérések eredményeinek összesítése.

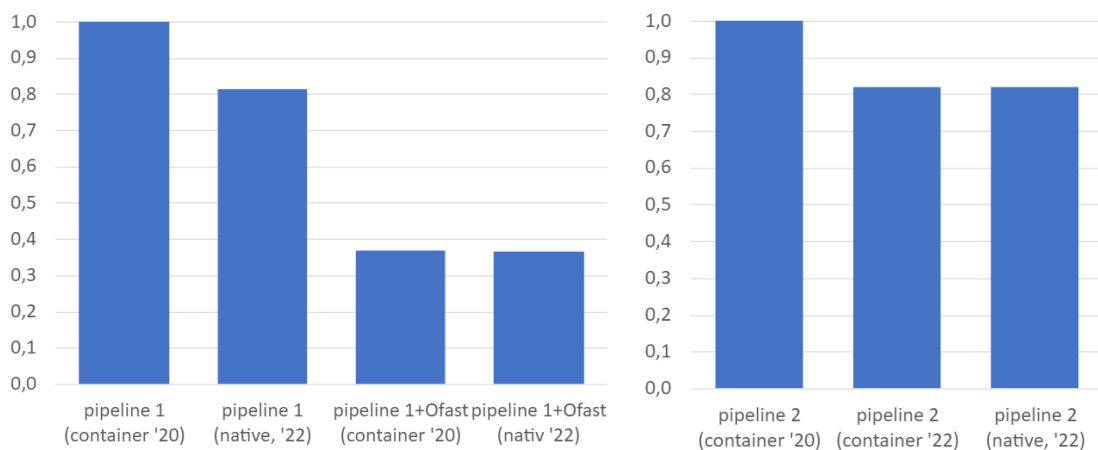
Kék szín jelöli a natív futtatások, míg narancssárga a konténerizált folyamatok előnyét.

Vizsgálatom során 25 különböző eszközt futtattam és 103 különböző eredményt kaptam (minden tárhely futtatással együtt valójában 447-et, eltekintve az MPI tesztektől). Az áttekintés érdekében az összes teszt eredményét ábrázoltam a 10. ábrán. Az ábra CPU részében csak az 1 szálon végrehajtott (szekvenciális) mérések szerepelnek. Az ábrán minden teszt esetén egy százalékos érték látható, amely jelzi, hogy a konténerizált vagy a natív futtatás teljesített jobban. Figyelembe véve, hogy néhány esetben az alacsonyabb, míg más esetekben a nagyobb mérési érték számít jobbnak, a jobban teljesítések értéke nem egyszerűen írható le

a natív és a konténerizált futtatások eredményeinek törtjeként. Az ábrán nincs figyelembe véve a pontokhoz tartozó benchmarkok pontos típusai, célom csak az, hogy lássuk, hogyan viselkednek általában a natív és a konténerizált futtatások. MPI esetében a bind architektúra volt a leghatékonyabb, felülmúlva a host teljesítményét is.

3.3. Pipeline (L3)

Az alkalmazott L3 pipeline többnyire egymás után hívott C programokból áll, amelyek egymás bemeneteit és kimeneteit használják a számításokhoz. Ezek főként CPU-intenzív műveletek, de memória- és tárolási I/O-t is használnak, például ki- és beírásra, illetve a lépések eredményeinek visszaolvasásához. Itt az optimalizálatlan pipeline binárisok natív és konténer teljesítményének összehasonlítása mellett összehasonlítottam az optimalizált binárisokkal, valamint a GCC régi és új verzióinak használatával. Az eredményeket a 11. ábra foglalja össze.



11. Ábra: Egy valódi tudományos pipeline futási arányai.
 (balra) a kód fordítás során történő optimalizálásából származó nyilvánvaló nyereség.
 (jobbra) a legújabb operációs rendszer használata befolyásolja a teljesítményt.

Itt a konténerben egy 2020-ban kiadott gcc verzió van, míg a hostnak a 2022-es verziója van. Figyeljük meg, hogy a kód tovább futott a régebbi konténerben, amikor nem volt optimalizálva, de az optimalizálás szinte teljesen megszüntette a különbségeket (6. Ábra (balra)). Ez azt jelenti, hogy érdemes minél több naprakész szoftvert telepíteni a konténerbe, amennyire csak lehet, hogy a futási idő kellően rövid legyen, és optimalizáljuk a kódot, mivel ez nagyságrendekkel csökkentheti a kód futási idejét és szinte teljesen kiküszöbölni a nem egyező gcc verziók közötti különbségeket. A 11. ábrán (jobbra) a pipeline 2 programot futtattam a legkorszerűbb eszközökkel mindkét esetben, de az összehasonlítás kedvéért egy régebbi szoftverösszeállítással ('20) is lefuttattam. Amikor a régi gcc-t használtam, a futási idő 22%-kal nőtt. Tehát ismét a szoftver naprakészsége nagyobb hatással van a futási időre, mint az, hogy a konténerizált vagy a natív környezetet használjuk.

4. Összefoglalás

Vizsgálataim eredményeképpen (ismét) bebizonyosodott, hogy a Singularity konténerek használata egységesített környezetet biztosít a tudományos számításokhoz, ami nem befolyásolja megfizethetetlenül negatívan a számítások futási idejét. Egyes esetekben a konténerek használata felülmúlhatja a natív szoftverkörnyezeteket. Lefuttattam méréseket a komplexitás három szintjén. Az L1 és L2 szinteken külön-külön végeztem tesztek a CPU, a memória és a lemezes I/O intenzív esetekre. Megállapításaim eredményeként kimutattam, hogy L1 és L2 esetén szinte nincs különbség a natív és konténeres futtatások teljesítménye között a memóriaintenzív feladatok szempontjából. A CPU-intenzív futtatások esetében találtam némi különbséget, de a legtöbb esetben a két környezet 5%-nál kisebb teljesítménykülönbséget mutatott. A tárolási I/O intenzív futások eredményei bizonyultak a legkülönbözőbb eredményeknek, ezért nagyon óvatossá kell lenni egy adott feladat erőforrás-követelmény típusának meghatározásánál, mielőtt I/O-intenzív feladatokat futtatunk egy konténerben. Ha szükséges, használjuk a host fájlrendszerét vagy a „/dev/shm”-et. A többszálúsítás is egy összetett témakör, de megállapítható volt, hogy a konténer általánosságban jól használja fel a CPU-időt, néha felül is múlhatja a natív környezetet, viszont némi szórás volt benne. MPI benchmarkok esetén a bind architektúra

felül tudta múlni a host teljesítménymutatóit is. L3 szinten pedig semmilyen különbség nem volt tapasztalható, a különbség csak a fordítóprogram verziójából fakadt.

Végző soron pedig megemlítem, hogy összehasonlítás alapjául szinte mindig ugyanazt a legfrissebb szoftververziót vettem. Nem életszerű, hogy bárhol a nekünk éppen szükséges aktuálisan legjobb programok vannak feltelepítve HPC-n vagy távoli asztal környezetben, és ha konténerben ezeket naprakészen tartjuk, a futási idő különbségek nagyságrendekkel (10-20% vagy akár konstansszoros eltéréssel) nagyobbak lehetnek, illetve optimalizáljuk a kódokat, ezek sokkal inkább befolyással vannak a teljesítményre, mint a szintetikus elvégzett tesztekben kiderülő apró különbségek. Ha tehetjük, a szoftvereket és a kódokat is forrásból bulidéljük a célhardverre ahelyett, hogy előkészített binárisból telepítenénk, és lehetőség szerint használjuk a host fájlrendszerét vagy a /dev/shm memóriabeli fájlrendszert. Tehát kimondható a végére, hogy a hordozhatóság ára nem a teljesítmény lesz, ha konténerizációról van szó, sőt újabb kernellel, szoftverrel gyorsulás is elérhető.

Köszönetnyilvánítás

Köszönetet mondok Polgár Péter külső témavezetőmnek minden segítségért, de különösképpen az (L2) mérések összeválogatásában és objektív kielemezésében nyújtott segítségéért. További köszönet Dr. Kocsis Gergely belső témavezetőmnek a dolgozat elkészítése során adott iránymutatásokért és minden további segítségért.

Kapcsolódó saját publikációk

[C1] Péter Polgár, Tamás Menyhárt, Csanád Bátor Baksay, Gergely Kocsis, Tibor Tajti, Zoltán Gál; Three Level Benchmarking of Singularity Containers for Scientific Calculations; The 12th International Conference on Applied Informatics, March 2-4, 2023 - Eger, Hungary.

[C2] Tamás Menyhárt, Péter Polgár, Csanád Bátor Baksay, Tibor Tajti, Gergely Kocsis, Zoltán Gál; Performance of Parallel and Storage Intensive Jobs on Clustered Singularity Containers; IEEE 21st International Symposium on Intelligent Systems and Informatics, Pula, Croatia, September 21-23 2023.

[P1] Péter Polgár, Tamás Menyhárt, Csanád Bátor Baksay, Gergely Kocsis, Tibor Gábor Tajti, Zoltán Gál; Three level benchmarking of Singularity containers for scientific calculations; Annales Mathematicae et Informaticae, Vol. 58, pp. 133-146, (2023).

[P2] T. Menyhárt, P. Polgár, C. B. Baksay, T. G. Tajti, G. Kocsis and Z. Gál, "Performance of Parallel and Storage Intensive Jobs on Clustered Singularity Containers," 2023 IEEE 21st Jubilee International Symposium on Intelligent Systems and Informatics (SISY), Pula, Croatia, 2023, pp. 000515-000520, doi: 10.1109/SISY60376.2023.10417911.

Hivatkozások

- [1] S. Abraham, A. K. Paul, R. I. S. Khan, A. R. Butt: On the Use of Containers in High Performance Computing Environments, in: 2020 IEEE 13th International Conference on Cloud Computing (CLOUD), 2020, pp. 284–293, doi: 10.1109/CLOUD49709.2020.00048.
- [2] C. Arango, R. Dernat, J. Sanabria: Performance Evaluation of Container-based Virtualization for High Performance Computing Environments, Microsoft Research WA 98052 (2005).
- [3] N. G. Bachiega, P. S. L. Souza, S. M. Bruschi, S. d. R. S. de Souza: Container-Based Performance Evaluation: A Survey and Challenges, in: 2018 IEEE International Conference on Cloud Engineering (IC2E), 2018, pp. 398–403, doi: 10.1109/IC2E.2018.00075.
- [4] R. K. Barik, R. K. Lenka, K. R. Rao, D. Ghose: Performance analysis of virtual machines and containers in cloud computing, in: 2016 International Conference on Computing, Communication and Automation (ICCCA), 2016, pp. 1204–1210, doi: 10.1109/CCAA.2016.7813925.
- [5] J. Bhimani, J. Yang, Z. Yang, N. Mi, Q. Xu, M. Awasthi, R. Pandurangan, V. Balakrishnan: Understanding performance of I/O intensive containerized applications for NVMe SSDs, in: 2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC), 2016, pp. 1–8, doi: 10.1109/PCCC.2016.7820650.
- [6] T. A. Bodhanya: Comparing Cloud Orchestrated Container Platforms : Under the lenses of Performance, Cost, Ease-of-Use, and Reliability, MA thesis, Uppsala University, Department of Information Technology, 2022, p. 31.
- [7] J. Hadley, Y. Elkhatab, G. Blair, U. Roedig: MultiBox: Lightweight Containers for Vendor-Independent Multi-cloud Deployments, in: Embracing Global Computing in Emerging Economies, ed. by R. Horne, Cham: Springer International Publishing, 2015, pp. 79–90, isbn: 978-3-319-25043-4.
- [8] G. Kocsis, I. Varga: Investigating the effectiveness of advertising on declining social networks, *Creative Mathematics and Informatics* 23.5 (2014).

- [9] M. Lindström: Containers & Virtual machines : A performance, resource & power consumption comparison, 2022.
- [10] P. E. N, F. J. P. Mulerickal, B. Paul, Y. Sastri: Evaluation of Docker containers based on hardware utilization, in: 2015 International Conference on Control Communication & Computing India (ICCC), 2015, pp. 697–700, doi: 10.1109/ICCC.2015.7432984.
- [11] Y. Pan, I. Chen, F. Brasileiro, G. Jayaputera, R. Sinnott: A Performance Comparison of Cloud-Based Container Orchestration Tools, in: 2019 IEEE International Conference on Big Knowledge (ICBK), 2019, pp. 191–198, doi: 10.1109/ICBK.2019.00033.
- [12] A. M. Potdar, N. D G, S. Kengond, M. M. Mulla: Performance Evaluation of Docker Container and Virtual Machine, *Procedia Computer Science* 171 (2020), Third International Conference on Computing and Network Communications (CoCoNet'19), pp. 1419–1428, issn: 1877-0509, doi: <https://doi.org/10.1016/j.procs.2020.04.152>, url: <https://www.sciencedirect.com/science/article/pii/S1877050920311315>.
- [13] A. Putri, R. Munadi, R. Negara: Performance analysis of multi services on container Docker, LXC, and LXD, *Bulletin of Electrical Engineering and Informatics* 9.5 (2020), pp. 2008–2011, issn: 2302-9285, doi: 10.11591/eei.v9i5.1953, url: <https://beei.org/index.php/EEI/article/view/1953>.
- [14] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, L. Peterson: Container-Based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors, *SIGOPS Oper. Syst. Rev.* 41.3 (Mar. 2007), pp. 275–287, issn: 0163-5980, doi: 10.1145/1272998.1273025, url: <https://doi.org/10.1145/1272998.1273025>.
- [15] E. Strohmaier, J. Dongarra, H. Simon, M. Meuer: The 500 most powerful commercially available computer systems, 2022, url: <https://www.top500.org/>.
- [16] A. Torrez, T. Randles, R. Priedhorsky: HPC Container Runtimes have Minimal or No Performance Impact, in: 2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC), 2019, pp. 37–42, doi: 10.1109/CANOPIE-HPC49598.2019.00010.

- [17] I. Varga: Weighted multiplex network of air transportation, *European Physical Journal B* 89.6 (2016).
- [18] C. G. Kominos, N. Seyvet and K. Vandikas, "Bare-metal, virtual machines and containers in OpenStack," 2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN), Paris, France, 2017, pp. 36-43, doi: 10.1109/ICIN.2017.7899247.
- [19] Á. Kovács, "Comparison of different Linux containers," 2017 40th International Conference on Telecommunications and Signal Processing (TSP), Barcelona, Spain, 2017, pp. 47-51, doi: 10.1109/TSP.2017.8075934.
- [20] Zhanibek Kozhirbayev, Richard O. Sinnott, A performance comparison of container-based technologies for the Cloud, *Future Generation Computer Systems*, Volume 68, 2017, Pages 175-182, ISSN 0167-739X, <https://doi.org/10.1016/j.future.2016.08.025>.
- [21] I. Mavridis and H. Karatza, "Performance and Overhead Study of Containers Running on Top of Virtual Machines," 2017 IEEE 19th Conference on Business Informatics (CBI), Thessaloniki, Greece, 2017, pp. 32-38, doi: 10.1109/CBI.2017.69.
- [22] Emily Le and David Paz. 2017. Performance Analysis of Applications using Singularity Container on SDSC Comet. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact (PEARC17)*. Association for Computing Machinery, New York, NY, USA, Article 66, 1–4.
<https://doi.org/10.1145/3093338.3106737>
- [23] Zhou, N., Georgiou, Y., Pospieszny, M. et al. Container orchestration on HPC systems through Kubernetes. *J Cloud Comp* 10, 16 (2021). <https://doi.org/10.1186/s13677-021-00231-z>
- [24] Pankaj Saha, Angel Beltré, Piotr Uminski, and Madhusudhan Govindaraju. 2018. Evaluation of Docker Containers for Scientific Workloads in the Cloud. In *Proceedings of the Practice and Experience on Advanced Research Computing (PEARC '18)*. Association for Computing Machinery, New York, NY, USA, Article 11, 1–8.
<https://doi.org/10.1145/3219104.3229280>

[25] A. M. Beltre, P. Saha, M. Govindaraju, A. Younge and R. E. Grant, "Enabling HPC Workloads on Cloud Infrastructure Using Kubernetes Container Orchestration Mechanisms," 2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC), Denver, CO, USA, 2019, pp. 11-20, doi: 10.1109/CANOPIE-HPC49598.2019.00007.

[26] SingularityCE User Guide, SingularityCE and MPI applications, <https://docs.sylabs.io/guides/latest/user-guide/mpi.html#>, (utolsó látogatás: 2024.05.23).

[27] Polgár Péter, Menyhárt Tamás: alátámasztó dokumentumok és ábrák. https://drive.google.com/drive/folders/1XlnHK6m-zCL_uQT3MnCOUeWNkfFQrPJ?usp=drive_link, (utolsó látogatás: 2024.05.23).

Saját munka leírása

A TDK munka során minden feladatot meghatározóan én végeztem el, beleértve az összes (L1) C és C++ kód megírását, a programok futtatását és a kapott eredmények elemzését is. Ugyanakkor több résznél is segítettek a témavezetőim, mind a formai követelményeknek megfelelő megfogalmazásban, táblázataim elsősorban jelen dolgozatba való migrálásában és néhány elkészítésében az adataimból, ötletek adásában, hogy mit lenne érdemes mérni, különféle Linux operációs rendszerbeli és kompatibilitási hibák elhárításában, és annak ellenőrzésében, hogy egy-egy megállapításom minél kevésbé legyen szubjektív.