Docker Container based PaaS Cloud Computing Comprehensive Benchmarks using LAPACK

Dmitry Zaitsev[1[0000-0001-5698-7324]], Piotr Luszczek[2[0000-0002-0089-6965]]

[1]Odessa State Environmental University, 15 Lvivska Str., Odessa, 65016, Ukraine

daze@acm.org, http://daze.ho.ua

[2]University of Tennessee's Knoxville, 1122 Volunteer Blvd, Knoxville, TN 37996, USA

luszczek@icl.utk.edu

Abstract. Platform as a Service (PaaS) cloud computing model becomes widespread implemented within Docker Containers. Docker uses operating system level virtualization to deliver software in packages called containers. Containers are isolated from one another and comprise all the required software, including operating system API, libraries and configuration files. With such advantageous integrity one can doubt on Docker performance. The present paper applies packet LAPACK, which is widely used for performance benchmarks of supercomputers, to collect and compare benchmarks of Docker on Linux Ubuntu and MS Windows platforms. After a brief overview of Docker and LAPACK, a series of Docker images containing LAPACK is created and run, abundant benchmarks obtained and represented in tabular and graphical form. From the final discussion, we conclude that Docker runs with nearly the same performance on both Linux and Windows platforms, the slowdown does not exceed some ten percent. Though Docker performance in Windows is essentially limited by the amount of RAM allocated to Docker Engine.

Keywords. Cloud computing, PaaS, Docker, benchmark, LAPACK, Top500

1 Introduction

A concept of cloud computing [1], widely spread recently, means on-demand availability of computer system resources. Peter Mell and Tim Grance from National Institute of Standards and Technology (NIST) define cloud computing as a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction. Their cloud model is composed of five essential characteristics, three service models, and four deployment models. Five essential characteristics include: on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service. The service models include: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a

Service (IaaS). Deployment models are the following: Private cloud, Community cloud, Public cloud, and Hybrid cloud.

Among known implementations of PaaS service model, Docker Containers [2-4] are rapidly developing and attracting more and more customers in various application areas. Docker uses operating system level virtualization to deliver software in packages called containers. Containers are isolated from one another and contain all the required software, including operating system API, libraries and configuration files. Containers can communicate with each other through well-defined channels. All containers are run by a single operating-system kernel and are thus more lightweight than virtual machines [5-6]. The software that hosts the containers is called Docker Engine and is installed now on Linux, MS Windows, and Apple MacOS platforms. Works on Docker has started in 2010 and first released in 2013. Recently Docker wins millions of developers and customers offering a public repository of containers. It supports also parallel and distributed computing with Docker Swarm technology.

Developers of Docker claim that the application within a container runs quickly and reliably from one computing environment to another. On the one hand, it is utmost convenient tool encapsulating within a single container everything required to run an application – all dependencies including operating system API, libraries, configurations etc. On the other hand, it is efficient because it is light-weighted compared with the virtual machine concept. Moreover, it is ubiquitous, running everywhere where Docker Engine is installed and access to repository of containers is provided. Thus, it looks like a miracle that looks a bit of suspicious for IT professionals [6,7]. The design of research presented in the paper was either to prove it strictly and convincingly or to depose a legend.

In the present paper after a brief overview of Docker Container technology, we come to LAPACK software [8,9], widely applied for performance benchmarks including modern supercomputers [10,11]; then we meticulously create a series of containers to run LAPACK, compose one solid container for benchmarks, and upload it to Docker Hub repository; we run the LAPACK container on Ubuntu Linux and MS Windows 10 platforms installed on the same computer and also planning to supply additional information regarding benchmarks for Apple MacOS platform in future; finally we represent benchmarks graphically and discuss obtained results. A conclusion that Docker completely corresponds to the company claims showing rather good performance for any available platform accomplishes the paper.

2 Basics of Developing and Running Docker Containers

Docker [2-4] represents one of the most successful implementations of the PaaS Cloud Computing concept. A Docker image encapsulates an application together with its entire environment including libraries and operating system and runs on Docker Engine. Docker Engine can be started on Linix, Windows, MacOS and in future on other operating systems. Thus a certain independence of an image from operating environment is provides while it is claimed that Docker runs an image considerably faster than a virtual machine.

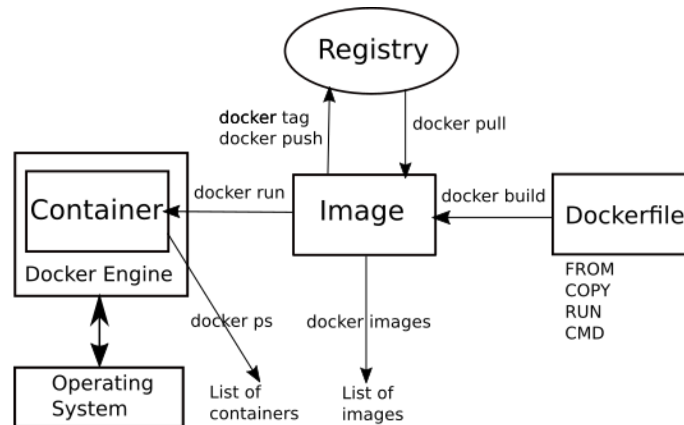

Fig. 1. Docker brief scheme of work.

Docker command line interface executes commands starting with "docker" prefix. To run an image, we use "docker run" command. For a quick start, we can either run *hello-world* image or use explicitly Linux *echo* command starting image *ubuntu*:

```
docker run hello-world
docker run ubuntu /bin/echo 'Hello world'
```

On processing *run* command, Docker creates and starts a new container from the image, downloading a new image if required. A container runs within Docker Engine which can be considered as a kind of thin virtual machine (Fig. 1); recently Docker Engine works within Linux, MS Windows, and MacOS. There are more than fifty Docker commands [2]. Among the most frequently used, we mention: "docker info" to display system-wide info; "docker images" to list images; "docker ps" to list containers (running images); "docker build" to build an image from a *Dockerfile* which represents a textual file specifying how the image should be built. For instance, we build and run image specified by the following file named *Dockerfile* and stored in the current directory:

```
cat Dockerfile
FROM ubuntu:latest
RUN /bin/bash
docker build -t u2 .
docker run -it u2
#
```

The option "-t" specifies the image name, and the option "-it" specifies an interactive mode of work; after starting, the image issues an invitation "#" waiting a bash command to process. A *Dockerfile* begins with "FROM" instruction that specifies parent image from which the current image is built. To use a file when building an image, we copy it into the image using "COPY" instruction. Instruction "RUN" speci-

fies commands which build a new layer of the current image. When an image is started, a command specified in “CMD” instruction is executed. Dockerfiles containing all the considered instructions are studied in Section 4.

On default, all the built images are stored locally within the current computer. It is rather convenient to store images in a cloud repository (registry), for instance http://hub.docker.com For this purpose we create the corresponding account and enter it using “docker login” command. Local images are identified using 6 octets represented as a hex number, we can obtain identifies via “docker images” command. To push an image to a repository we tag it with a name using “docker tag” command and then push it using “docker push” command with specified name. When working on a new computer, we can pull an image from repository using “docker pull” command. Pushing and pulling images are studied in detail in Section 4.

3 LAPACK as de-facto Standard for Performance Benchmarks

The LAPACK library contains a collection of numerical methods to solve problems arising from dense systems of linear equations assuming one of the following forms:

$$
\begin{gathered}
Ax = b, \\
Ax = \lambda x, \\
Ax = \lambda Bx, \\
Av = \sigma u.
\end{gathered}
$$

Less conceptually, the properties of matrices A and B dictate the specific of the algorithmic approach taken by the LAPACK solver. The overarching methodology involves the decomposition approach [12] that splits the algorithmic work into two distinct phases: factorization and substitution. The former is commonly the computationally-intensive step that consumes majority of execution time. The latter uses the output of the factorization to deliver the solution of the original problem at much lower cost than the first step. Moreover, the factorization results may be reused multiple times with different right-hand sides for much reduced computation time in many practical situations. Furthermore, numerical stability of using the factors is superior to using explicit inverse of the system matrix that may fail with overflows or division-by-zero even when the solve with the factors would still succeed to deliver a few digits of accuracy in the solution. The factorizations implemented in LAPACK include the LU, Cholesky, and QR factorizations that are often called one-sided factorizations because they apply numerical transformations to only one side of the original matrix and hence do not preserve the spectral properties. In contrast, the two-sided transformations do preserve the original matrix spectrum and are used in LAPACK’s Schur decomposition of a matrix. These two types of transformations present a different challenge to the modern hardware and thus may be used for a wide range of evaluations.

From the benchmarking standpoint, LAPACK offers a wide range of routines that reveal performance of characteristics of the tested hardware platform. At the basic

level there are BLAS subroutines that constitute the portable performance layer above the hardware that allows the rest of LAPACK to express more complex algorithms in an efficient manner. Using BLAS for benchmarking is an easy option to obtain trivial hardware metrics such as bandwidth and latency of the main memory by running some of the Level 1 and Level 2 BLAS subroutines. Moving up to Level 3 BLAS allows the user to test compute capability of the hardware platform both in terms of single-core performance as well as its scaling across all available sockets and cores contained therein. In practice, more complex measures of performance are obtained from LAPACK subroutines. One common example is the High Performance LINPACK benchmark [13] commonly referred to as HPL. The benchmark measure the time taken to solve a system of linear equations with a dense system matrix of an arbitrary size by using LU factorization with partial pivoting. This can be easily implemented by a single call to LAPACK's routine DGESV() and stands to show how LAPACK is an important performance tool. In fact, HPL has been used for decades for measuring efficiency and comparison the largest supercomputing machines in the world as is recorder by the TOP500 list [10]. Prior to proliferation of distributed memory machines, the shared memory supercomputers were dominating the list and, at that time, LAPACK's implementation of DGESV() was a perfect software implementation of HPL eligible for running and submitting TOP500 results. Later on, a distributed memory implementation of DGESV() was required and it had been provided by the ScaLAPACK library in the form of PDGESV() routine that not only computes locally on each of the distributed nodes but also coordinates the solution between the processes through an interconnect fabric, including Ethernet or InfiniBand, and a message passing library such as MPI. However, the detailed use and analysis of ScaLAPACK is outside of scope of this work due its complex software stack requirements and hardware complexities associated with running a modern distributed-memory cluster. These complex hardware-software interactions complicate evaluation of containers and might prevent from drawing proper conclusions about the associated overheads.

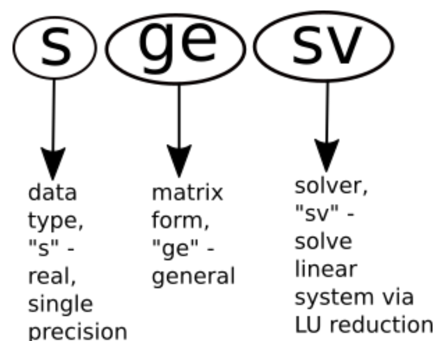

Fig. 2. Scheme of LAPACK routine names.

LAPACK packet is downloaded freely from its website [8] and installed according to the installation guide in some quarter of an hour, in its work it uses BLAS library [9,13]. For instance, to solve a given system, we can use the following small program that initializes matrix A and vector of the right side b statically and then calls routine LAPACKE_sgesv to solve the corresponding system:

```
#include <stdio.h>
#include <lapacke.h>
#define size 3            // dimension of matrix A
int main()
{
  lapack_int n=size, nrhs=1, lda=size, ldb=1, info,
ipiv[size];
  float A[size*size] = {2.0,5.0,8.0,
                        3.0,-6.0,9.0,
                        4.0,7.0,-1.0};
  float b[size] =      {13.0,25.0,-7.0};
  int i,j;
  info = LAPACKE_sgesv(LAPACK_ROW_MAJOR, n, nrhs, A, lda,
ipiv, b, ldb);
  for (i=0; i<n; i++) printf("%f\n", b[i]); // print vec-
tor x
}
```

Finally it prints the obtained vector of solutions which replaces the right side vector *b*. The program is written in C language and uses C interface of LAPACK called LAPACKE, all the corresponding routines have prefix “LAPACKE_”. LAPACKE only translates the call into the call of the corresponding LAPACK routine, sgesv in the considered example (a scheme of LAPACK routine names abbreviation is explained by Fig. 2). The program can be built and run using the following *Makefile*:

```
include ../../make.inc
all: lp_ex1
LIBRARIES = ../../$(LAPACKELIB) ../../$(LAPACKLIB)
$(BLASLIB)
lp_ex1: lp_ex1.o $(LIBRARIES)
  $(LOADER) $(LOADOPTS) -o $@ $^
  ./$@
.c.o:
  $(CC) $(CFLAGS) -I. -I../include -c -o $@ $<
```

We assume that out example directory is situated at the same level as the standard LAPACKE example directory /lapack-3.8.0/LAPACKE/example. The computed vector of solutions can be checked by substitution into equations of the system:

```
daze@lion:~/lapack-3.8.0/LAPACKE/example$ make
gcc -O3 -I. -I../include -c -o lp_ex1.o lp_ex1.c
```

```
gfortran -o lp_ex1 lp_ex1.o ../../liblapacke.a
../../liblapack.a ../../librefblas.a
./lp_ex1
-1.313167e+00
-1.000000e-01
1.800000e+00
daze@lion:~/lapack-3.8.0/LAPACKE/example$
```

Standard set of LAPACKE tests includes routines which solve a given linear system or implement computations by the least square method. At first, a system of a given size is generated with random elements and then it is solved, the obtained solutions printed. It uses two options: “-n” to specify the system size and “-nhrs” to specify the number of right-hand sides; for instance:

```
daze@lion:~/lapack-3.8.0/LAPACKE/example$
./xexample_DGESV_rowmajor -n 3
 Entry Matrix A
   0.34  -0.11   0.28
   0.30   0.41  -0.30
  -0.16   0.27  -0.22
 Right Rand Side b
   0.05
  -0.02
   0.13
LAPACKE_dgesv (row-major, high-level) Example Program Re-
sults
 Solution
  -0.48
   1.19
   1.22
 Details of LU factorization
   0.34  -0.11   0.28
   0.88   0.50  -0.55
  -0.48   0.43   0.15
 Pivot indices
      1      2      3
daze@lion:~/lapack-3.8.0/LAPACKE/example$
```

One can check the obtained results and calculate the error, such intermediate results as LU-factorization and pivot indices are printed as well, though in the present study we are interested mainly in LAPACK running time to use it for benchmarks.

4 Install and Run LAPACK in Docker Containers

In the present section we create a Docker image to run LAPACK tests and also an additional image to solve the example equation from the previous section. We are going to compare multi-layer and solid images as well that is why we create a series of images having the following structure of directories and files of the top directory *lpd*:

```
daze@lion:~$ ls -R lpd
lpd:
myubu1  myubu2  myubu3  myubu4  ubuntu-lapack
lpd/myubu1: Dockerfile
lpd/myubu2: Dockerfile  lapack-3.8.0.tar.gz
lpd/myubu3: Dockerfile  dt_example_DGESV_rowmajor.c
Makefile
lpd/myubu4: Dockerfile  lp_ex1.c  Makefile
lpd/ubuntu-lapack: Dockerfile lapack-3.8.0z.tar.gz
daze@lion:~$
```

We use a separate directory for a separate Docke image; besides the corresponding *Dockerfile* that specifies how to build the image, each directory contains the required software or other files which will be embedded into the image. We create the following directories and build the following images which we can use separately:

- myubu1 — recent Linux Ubuntu and essential developer tools;
- myubu2 — adds to *myubu1* LAPACK installed;
- myubu3 — adds to *myubu2* BLAS, LAPACKE, and LAPACKE examples installed;
- myubu4 — runs a program that solves the example linear system using LAPACK;
- ubuntu-lapack — a solid Docker image corresponding to *myubu3*.

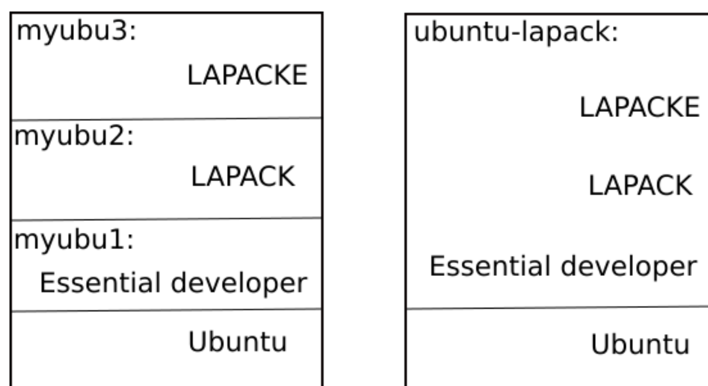

Fig. 3. Scheme of Docker images for benchmarks.

Let us consider in detail how we build and try each of mentioned images. To create image *myubu1*, which contains *ubuntu* and essential developer tools, we use the following *Dockerfile*:

```
FROM ubuntu:latest
RUN apt-get update && apt-get install -y \
     build-essential \
     gfortran \
     python
RUN /bin/bash
```

The image starts from the latest image of *ubuntu* and adds such developer tools as basic compilers and *make* utility. We can use *myubu1* separately to develop programs in C, C++, Gfortran, and Python. The following commands allow us to build and try *myubu1*:

```
docker image build -t myubu1 .
docker run -it myubu1
```

To create image *myubu2*, which installs LAPACK on *myubu1*, we use the following *Dockerfile*:

```
FROM myubu1:latest
COPY lapack-3.8.0.tar.gz .
RUN tar -zxvf lapack-3.8.0.tar.gz
RUN cp /lapack-3.8.0/make.inc.example /lapack-
3.8.0/make.inc
RUN ulimit -s unlimited && cd /lapack-3.8.0 && make
RUN /bin/bash
```

The image starts from the *myubu1* image and installs LAPACK on it. We can use *myubu2* separately to develop programs in Gfortran which call LAPACK functions or run LAPACK tests written in Gfortran. The following commands allow us to build and try *myubu2*:

```
docker image build -t myubu2 .
docker run -it myubu2
```

To create image *myubu3*, which installs BLAS, LAPACKE, and LAPACKE examples on *myubu2*, we use the following *Dockerfile*:

```
FROM myubu2:latest
COPY dt_example_DGESV_rowmajor.c /lapack-
3.8.0/LAPACKE/example
COPY Makefile /lapack-3.8.0/LAPACKE/example
```

```
RUN cd /lapack-3.8.0/CBLAS && make && cd /lapack-
3.8.0/LAPACKE && make && cd /lapack-3.8.0/LAPACKE/example
&& make
RUN /bin/bash
```

The image starts from the *myubu2* image and installs BLAS, LAPACKE, and LAPACKE examples on it. We can use *myubu3* separately to develop programs in C which call LAPACKE functions or run LAPACKE tests written in C. The following commands allow us to build and try *myubu3*:

```
docker image build -t myubu3 .
docker run -it myubu3
```

To create image *myubu4*, which runs a program that solves the example linear system using LAPACK, we use the following *Dockerfile*:

```
FROM myubu3:latest
RUN mkdir /lapack-3.8.0/LAPACKE/myex
COPY lp_ex1.c /lapack-3.8.0/LAPACKE/myex
COPY Makefile /lapack-3.8.0/LAPACKE/myex
RUN cd /lapack-3.8.0/LAPACKE/myex && make
CMD /lapack-3.8.0/LAPACKE/myex/lp_ex1
```

The image starts from the *myubu3* image and installs the example program *lp_ex1.c* and the corresponding *Makefile*. We can use *myubu4* to solve the example linear system within Docker. The following commands allow us to build and try *myubu4*:

```
docker image build -t myubu4 .
docker run myubu4
```

The obtained results coincide with the results obtained using LAPACK directly. To create a solid image *ubuntu-lapack* that corresponds to *myubu3* and will be run on various platforms for the benchmarks, we use the following *Dockerfile*:

```
FROM ubuntu:latest
COPY lapack-3.8.0z.tar.gz .
RUN apt-get update && apt-get install -y apt-utils build-
essential gfortran python && \
     tar -zxvf lapack-3.8.0z.tar.gz && \
     cp /lapack-3.8.0/make.inc.example /lapack-
3.8.0/make.inc && \
     ulimit -s unlimited && cd /lapack-3.8.0 && make && \
     cd /lapack-3.8.0/CBLAS && make && cd /lapack-
3.8.0/LAPACKE && make && cd /lapack-3.8.0/LAPACKE/example
&& make
RUN /bin/bash
```

We push the final image *ubuntu-lapack* to our repository at *http://hub.docker.com* to make it public and use in Docker for Windows benchmarks:

```
docker tag ubuntu-lapack zsoftua/ubuntu-lapack
docker push zsoftua/ubuntu-lapack
```

When required, we can pull it from the repository:

```
docker pull zsoftua/ubuntu-lapack
```

We assume that in both cases above we are logged to a repository otherwise we can add a prefix with repository address to the image name to the left.

5 LAPACK-Docker Benchmarks in Linux, Windows, and MacOS

For the performance benchmarks, big systems are solved using LAPACKE, the output is redirected to NULL device. In the simple case, we can measure the running time using system utility time as follows:

```
daze@lion:~/lapack-3.8.0/LAPACKE/example$ time
./xexample_DGESV_rowmajor -n 5000 > /dev/null
real  0m38,057s
user  0m37,964s
sys 0m0,092s
daze@lion:~/lapack-3.8.0/LAPACKE/example$
```

Thus, the program running time for system of size 5000 is about 38,057 seconds. For more precise evaluation, test program *xexample_DGESV_rowmajor* is modified by adding code for measuring time and commenting all printing operators save error messages, we call the resulting program *dt_ xexample_DGESV_rowmajor*:

```
daze@lion:~/lapack-3.8.0/LAPACKE/example$ time
./dt_xexample_DGESV_rowmajor -n 5000 > /dev/null
1 5000  25.868977s
real  0m26,654s
user  0m26,090s
sys 0m0,080s
daze@lion:~/lapack-3.8.0/LAPACKE/example$
```

The times obtained inside the program and by system utility *time* are very close with the difference less than 1 second. The essential difference with the previous listing is explained by the fact, the printing is commented in program *dt_ xexample_DGESV_rowmajor*, thus we conclude that printing consumes about one third of time for a system of size 5000. Further we will use benchmark tests without printing results. As for the time measuring code, the following sketch program illustrates it:

```
#include <time.h>
#include <bits/time.h>
#include <sys/time.h>
double magma_wtime( void )
{
  struct timeval t;
  gettimeofday( &t, NULL );
  return t.tv_sec + t.tv_usec*1e-6;
}
…
double t1,t2;
t1=magma_wtime();
info = LAPACKE_dgesv(...);
t2=magma_wtime();
…
fprintf( stderr, "%d\t%d\t%#fs\n", nrhs, n, t2-t1 );
```

To organize tests in a sequence, we compose the following tiny script:

```
for n in <list of time instants>;
do
  ./dt_xexample_DGESV_rowmajor -n $n;
done
```

For our benchmark tests, we use the same desktop computer Hare as in [14]: Intel Core i5 3.2GHz, 4 cores, RAM 8Gb. We compare the performance obtained directly in Linux with the performance obtained in Docker environment which runs both on Linux and MS Windows 10, besides we compare solid and many-layered images.

For instance, we run tests directly in Linux with:

```
daze@lion:~/lapack-3.8.0/LAPACKE/example$ for n in 1000
2000 3000 4000 5000 6000 7000 8000 9000 10000 11000 12000
13000 14000 15000; do ./dt_xexample_DGESV_rowmajor -n $n;
done
```

and we run tests in Docker with:

```
daze@lion:~/lpd/ubuntu-lapack$ docker run -it myubu3
root@24e7e0f0f7c0:/# cd /lapack-3.8.0/LAPACKE/example
root@24e7e0f0f7c0:/lapack-3.8.0/LAPACKE/example# for n in
1000 2000 3000 4000 5000 6000 7000 8000 9000 10000 11000
12000 13000 14000 15000; do ./dt_xexample_DGESV_rowmajor
-n $n; done
```

Basic obtained benchmarks are represented in Table 1. Matrix sizes from 1000 to 15000 are considered. We have four columns comparing a direct run of LAPACK on Ubuntu with running it within a Docker container either as multilayer or solid image,

and finally, with running the solid Docker container within MS Windows 10. A brief vivid comparison is shown in Fig. 4.

Table 1. Comparing benchmarks of Docker using LAPACK.

Matrix size	LAPACK on Ubuntu (s)	Docker multi-layer image in Ubuntu (s)	Docker solid image in Ubuntu (s)	Docker solid image in MS Windows (s)
1000	0.230743	0.231323	0.233148	0.248542
2000	1.731192	1.729169	1.731458	1.814025
3000	5.706029	5.724043	5.785270	5.987914
4000	13.272057	13.306907	13.324464	13.912880
5000	25.860373	26.286293	25.965536	27.209460
6000	44.290466	44.523421	44.483824	46.647734
7000	70.136287	70.723309	70.605154	73.950816
8000	104.262183	105.543180	105.438383	110.346505
9000	148.151378	150.613408	150.461304	157.327095
10000	202.458930	206.776427	207.061478	217.058503
11000	271.838308	277.929413	278.195353	293.229381
12000	357.815169	361.973850	362.008640	387.046685
13000	462.599758	464.142452	463.041850	499.041814
14000	586.326644	586.678216	587.664414	631.014162
15000	729.811495	733.366776	731.759509	786.755366

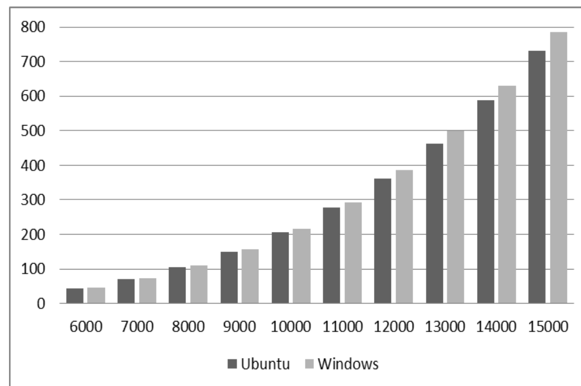

Fig. 4. Comparing Docker performance in Linux (Ubuntu) and Windows.

As for the chosen range of the system size, it is limited by the RAM size when LAPACK goes out of memory. And as for Docker for Windows, the range depends on the amount of RAM allotted to Docker Engine. When Docker starts using virtual memory, the performance slows down considerably (Fig. 5).

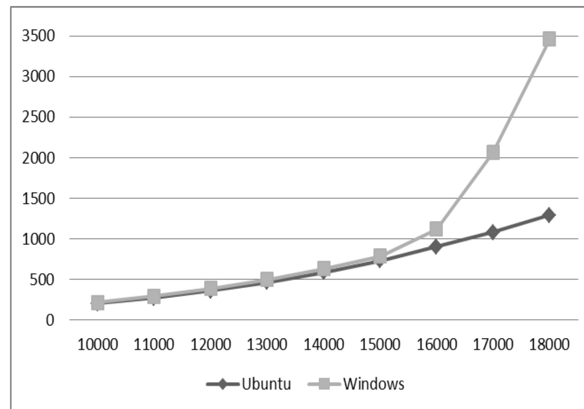

Fig. 5. Docker on Windows performance fall at exceeding RAM allocated to Docker Engine.

6 Final Discussions and Conclusions

Thus, using Docker yields very little slowdown (about one percent) in Ubuntu and little slowdown (about ten percent) in Windows that acknowledges that Docker platform is a perfect solution from performance point of view as well.

We should mention that Docker performance in Windows is considerably limited by the amount of RAM allocated to Docker Engine, a slowdown observed when active swapping within virtual memory starts.

Note that, the benchmarks have been collected for computations over real numbers. Recently computations over integer numbers become more significant for manifold applications to discrete event systems [15], fuzzy logic [16], cybersecurity, and artificial intelligence domains. For benchmarks over integer numbers, we can apply packet ParAd [14,17] recently developed based on clans composition theory [18].

7 References

1. Cloud Computing: Principles, Systems and Applications, Antonopoulos, Nick, Gillam, Lee (Eds.), Springer, 2017.
2. Docker https://www.docker.com/
3. A. Ahmed and G. Pierre, "Docker Image Sharing in Distributed Fog Infrastructures," 2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Sydney, Australia, 2019, pp. 135-142. doi: 10.1109/CloudCom.2019.00030

4. Rajkumar Buyya; Satish Narayana Srirama, "A Lightweight Container Middleware for Edge Cloud Architectures," in Fog and Edge Computing: Principles and Paradigm , Wiley, 2019, pp.145-170.
5. P. Zhang, M. Zhou and X. Wang, "An Intelligent Optimization Method for Optimal Virtual Machine Allocation in Cloud Data Centers," in IEEE Transactions on Automation Science and Engineering. doi: 10.1109/TASE.2020.2975225
6. A. Lingayat, R. R. Badre and A. Kumar Gupta, "Performance Evaluation for Deploying Docker Containers On Baremetal and Virtual Machine," 2018 3rd International Conference on Communication and Electronics Systems (ICCES), Coimbatore, India, 2018, pp. 1019-1023.
7. Emiliano Casalicchio and Vanessa Perciballi. 2017. Measuring Docker Performance: What a Mess!!! In Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion (ICPE вЂ™17 Companion). Association for Computing Machinery, New York, NY, USA, 11вЂ“16.
8. LAPACK http://www.netlib.org/lapack/
9. E. Angerson et al., "LAPACK: A portable linear algebra library for high-performance computers," Supercomputing '90:Proceedings of the 1990 ACM/IEEE Conference on Supercomputing, New York, NY, USA, 1990, pp. 2-11.
10. Top500 http://top500.org
11. G. Xie and Y. Xiao, "How to Benchmark Supercomputers," 2015 14th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES), Guiyang, 2015, pp. 364-367.
12. G. W. Stewart. The decompositional approach to matrix computation. Computing in Science & Engineering, 2(1):50-59, Jan/Feb 2000.
13. Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet, The LINPACK Benchmark: Past, Present, and Future, Concurrency and Computation: Practice and Experience, 15(9):803-820, August 10, 2003.
14. Dmitry Zaitsev, Stanimire Tomov, Jack Dongarra. Solving Linear Diophantine Systems on Parallel Architectures, IEEE Transactions on Parallel and Distributed Systems, 30(5), 2019, 1158–1169. DOI: 10.1109/TPDS.2018.2873354
15. Zaitsev D.A. Verification of Computing Grids with Special Edge Conditions by Infinite Petri Nets, Automatic Control and Computer Sciences, 2013, Vol. 47, No. 7, pp. 403–412. DOI: 10.3103/S0146411613070262
16. Zaitsev D.A., Sarbei V.G., Sleptsov A.I., Synthesis of continuous-valued logic functions defined in tabular form, Cybernetics and Systems Analysis, Volume 34, Number 2 (1998), 190-195. DOI: 10.1007/BF02742068
17. ParAd http://github.com/dazeorgacm/ParAd
18. Zaitsev D.A. Sequential composition of linear systems' clans, Information Sciences, Vol. 363, 292–307. Online 12 February 2016. DOI: 10.1016/j.ins.2016.02.016