POLITECNICO DI MILANO
DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
HIGH PERFORMANCE PROCESSORS AND SYSTEMS PROJECT

H-Droid Cluster

Bruzzo Paolo and Casula Dario
{paolo.bruzzo ; dario.casula}@mail.polimi.it
March - June 2014
Tutor: Ferroni Matteo
Professors: Sciuto Donatella, Santambrogio Marco D.

1 ABSTRACT

Today's smartphones are becoming more and more powerful, and the number of users around the world is constantly increasing year after year[1]. New functionalities, more storage and a higher computational power are the key-features of a successful device just brought onto the market, but how many users effectively exploit all the features that such a device offers? What about making these devices share some of their unused resources (i.e. computational power)? Here is where the idea of cluster of mobile devices comes into play.

2 INTRODUCTION

The general idea of making computers cooperate in a cluster actually comes from the '60s [2], and there has been plenty of research works on this topic. Mobile computing instead can be considered a pretty new trend, and the idea of making a cluster out of them has already been demonstrated by F. Busching, S. Schildt, L. Wolf, to be a reasonable and feasible concept that has all the basis to be a successful future topic for further studies[3]. In this report we will describe how we explored the potentials of a mobile cluster in a more realistic scenario, in which the devices are different from one another. The goal of our study is not to test the peak performance reachable by our devices, but we want to check the loss of performance experienced by adding one device after another to the cluster, and find out whether there exist

some realistic situations in which a mobile cluster can be exploited.
The report is organized as follows: in Section 3 we start introducing the technologies used nowadays to setup and benchmark (super)computer clusters, while in Section 4 we proceed explaining the technique we adopted to use the same technologies with the smartphones; in Section 5 we present our solution to setup a cluster in a mobile devices context, then in Section 6 we discuss the results obtained and we conclude with Sections 7 and 8 with our conclusions and possible future developments.

3 Technologies Overview

To make the cluster work and to study its behavior, it is necessary to have a way to make all the nodes communicate with each other and a tool to measure the performance. MPI and Linpack are two libraries that we have chosen to fulfill our needs: the first implements a communication protocol while the second is the base of a family of benchmarks.

3.1 MPI

The Message Passing Interface (MPI) is a communication protocol used to program parallel computers and it is now a standard de facto for distributed-memory systems. MPI allows to define a virtual topology of the cluster network and to distribute the computation of a program among all the nodes of the systems, sending to and gathering data from all the devices and leaving to the programmer the possibility to choose the number of processes to allocate for each single node of the cluster. Communication takes place in a "Communication World" where all the machines of the network can thus communicate with each other, both with a point-to-point connection and with a collective one. MPI uses a static network topology that has to be defined (using ip addresses in a configuration file) before a master machines initializes the communication. The protocol does not support a run-time re-configuration of the network topology. There are a lot of implementations of MPI, written in several languages and similar in the basic functions. We have chosen the MPICH2 implementation because it's a mobile implementation of MPI which supports all the features required by the HPL benchmark, it is specific for distributed-memory applications used in parallel computing and it can be easily installed in a Linux system (see section 6 and the repository [4] for further details).

3.2 HPL

Linpack (LINear algebra PACKage) is a software library which contains routines for solving problems in linear algebra. It was originally written in Fortran by Jack Dongarra from the University of Tennessee in the '70s. Linpack makes use of BLAS (Basic Linear Algebra Subprograms) for performing common vector and matrix operations. The Linpack Benchmark is based on the Linpack library and it is a measure of a system's 64bits-floating-point operations rate (MFLOPS), that is an estimation of its computing power. HPLinpack (Highly Parallel) is a version of the benchmark suitable for testing parallel computers and HPL (High Performance)

is a portable implementation of HPLinpack which measures the performance of a distributed-memory parallel system. HPL is currently used to provide the Top500, the list of most powerful computing systems (supercomputers) in the world [3].

THE ALGORITHM: HPL generates and solves a random dense linear system of equations on distributed-memory computers. It uses the BLAS library to handle linear operations and an implementation of MPI (Message Passing Interface) to distribute the workload to all the nodes of the distributed system. The algorithm solves the linear system of order N:

$$Ax = b\ ;\ A \in \mathbf{R}^{nxn}\ ;\ x, b \in \mathbf{R}^{n}$$

by computing the LU factorization of the coefficient matrix A:

$$Ax = LUx = b$$

into two matrices L (lower triangular) and U (upper triangular) using consecutive Gaussian eliminations [10,11]. The solution of the system is then obtained by solving the second equation of the new system, where P is a permutation matrix.

$$\begin{cases} Lx = Pb \\ Ux = y \end{cases}$$

Data is distributed to the nodes by partitioning the matrix into sub-blocks and assigning each of them cyclically to a PxQ grid of processes as shown in the figure, where the matrix is divided into 8x8 blocks of a fixed size and each block is assigned to one of the 6 processes of the 2x3 grid.

Figure 3.1: Matrix 8x8 splitted into a 2x3 grid

As shown in the example of figure 3.1, it is possible that the workload is assigned to every process in a non-balanced way: processes in the right section of the second matrix have a significantly lower load. The choice of the blocks' size is indeed one of the parameters that have to be taken into account in the tuning of the benchmark. A deeper analysis is made in the section 5.1 of this document. The LU decomposition is computed recursively factorizing a

panel of columns of the size of one block. Once the panel factorization has been computed, partial results are broadcasted and the overall factorization can proceed updating the trailing matrix, which will be decomposed in the same way in the next iteration of the loop.

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = P \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{11} & U_{12} \\ 0 & L_{22} \end{bmatrix} = P \begin{bmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22} \end{bmatrix}$$

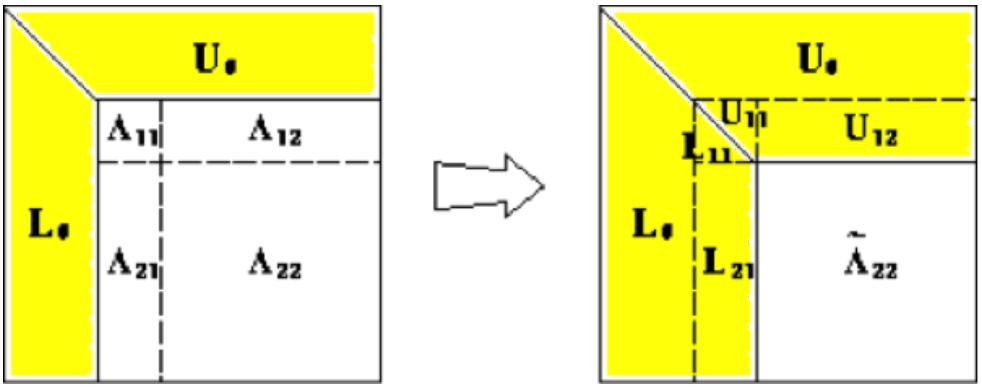

Figure 3.2: Panel factorization and trailing matrix updating

3.3 Android Cluster Structure

The cluster of our case study is heterogeneous in terms of CPUs, OS version, computational power and memory available. We have used 5 phones that belong more or less to same generation of smartphones (years 2009-2010), a Samsung Galaxy S2 and a Samsung Tab 2; a short description of the devices is available in the following table:

Model	CPU	Cores	Memory	OS
Samsung S5570	Qualcomm MSM 7227 - 832 MHz	1	384 MB	Android 2.3
LG-P500	ARM11 - 600 MHz	1	512 MB	Android 2.3.2
LG-P500	ARM11 - 600 MHz	1	512 MB	Android 2.3.3
LG-P350	ARM11 - 600 MHz	1	140 MB	Android 2.2
HTC Wildfire	Qualcomm MSM7227 - 528 MHz	1	384 MB	Android 2.3.3
Galaxy Tab 2	ARM Cortex A9 - 1 GHz	2	1 GB	Android 4.0.3
Samsung Galaxy S2	ARM Cortex A9 - 1.2 GHz	2	1 GB	Android 4.1.2

Since the phones need to communicate with each other using MPI, we had to choose a master phone responsible of initializing the connection and distributing the job to the other devices in the cluster (the slaves). The choice has been done totally random because it does not impact on the final result, and we have selected Samsung S5570. While for common MPI applications you want to use low latency links between nodes, for a mobile phone the only realistic available option is WiFi, so we set up the cluster like shown in Figure 3.3.

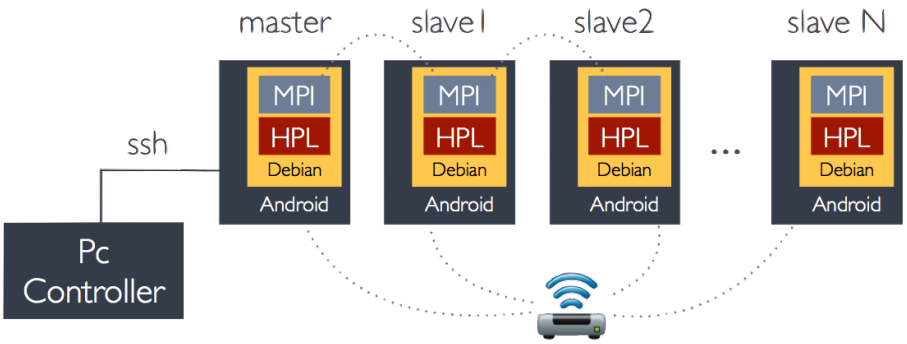

Figure 3.3: Cluster organization

4 Debian On Android

Nowadays it's not possible to run MPI on any mobile phone operative system, and there exist no benchmark in the market specifically written for mobile clusters; these are the two reasons why we installed Debian on the phones. In this way we have been able to run MPICH2 and test performances with HPL.
The first step to install Debian on Android, was to gain the root privileges on the devices, and we managed to do it with a simple program called *OneClickRoot* that is freely available on the Internet. The next step was to prepare a Debian image for ARM processors: we chose to do it with *Deboostrap*, a tool that allows to install a core Debian operating system into the subdirectory of another operating system already installed (Android in our case). The procedure has to be made in two steps:

Step 1 has been done on a personal computer with Ubuntu, and we ran the following bash script to prepare the image:

```
# installs the debootstrap tool on the local machine
apt-get install debootstrap
# creates an empty image of 1GB
dd if=/dev/zero of=debian.img seek=1024M bs=1 count=1
mke2fs -F debian.img
# mounts the image in a directory
mkdir debian
mount -o loop debian.img debian
# runs the first step of debootstrapping
debootstrap -arch armel -foreign squeeze debian http://ftp.cz.debian.org/debian
umount debian
```

Step 2 has been done on a rooted Android phone with an ARMv6 processor, Busybox installed, and the image generated at STEP 1 saved in the sdcard (the image generated is compatible with all the later ARMs CPUs) running the following bash script:

```
# export variables
export kit=/sdcard/
export mnt=/data/local/mnt
busybox mkdir -p $mnt
export PATH=/usr/bin:/usr/sbin:/bin:$PATH
export TERM=linux
export HOME=/root
# make and build a loop file
busybox mknod /dev/loop0 b 7 0
mount -o loop,noatime $kit/debian.img $mnt
# chroot into Debian
chroot $mnt /debootstrap/debootstrap --second-stage
# set the source list (e.g. to use apt-get)
echo 'deb http://ftp.cz.debian.org/debian squeeze main' > $mnt/etc/apt/sources.list
echo 'nameserver 8.8.8.8' > $mnt/etc/resolv.conf
```

Once this procedure has been done, it's possible to chroot into Debian. Chroot is a Unix operation that changes the apparent root directory for the current running processes and their children, and we use it to jump into the mounted Debian filesystem. It's important to notice a couple of things: first of all we are minimally invasive on the native OS because Debian is saved on the sdcard and once we chroot into it, the Android OS and files are inaccessible; furthermore we are not booting Linux, but we are running on the same kernel of Android, so many useful modules that usually belong to a Linux kernel are missing here.
For the sake of semplicity, we wrote a shell script that can be used to chroot into Debian:

```
##########################################
# Variables                              #
##########################################
export host=phone1
export kit=/sdcard/
export mnt=/data/local/mnt
busybox mkdir -p $mnt
export PATH=/usr/bin:/usr/sbin:/bin:$PATH
export TERM=linux
export HOME=/root

##########################################
# Mounting filesystem                    #
##########################################
echo "Mounting Loop Device..."
busybox mount -o loop,noatime $kit/debian.img $mnt
if [ $? -ne 0 ];then echo "Unable to mount the loop device!"; else echo "loop device mounted"; fi

busybox chroot $mnt mount -t devpts devpts /dev/pts
if [ $? -ne 0 ];then echo "Unable to mount /dev/pts!"; else echo "pts mounted"; fi

busybox chroot $mnt mount -t proc proc /proc
if [ $? -ne 0 ];then echo "Unable to mount /proc!"; else echo "proc mounted"; fi

busybox chroot $mnt mount -t sysfs sysfs /sys
if [ $? -ne 0 ];then echo "Unable to mount /sys!"; else echo "sys mounted"; fi

##########################################
# Setting hostname                       #
##########################################
echo -n "Setting hostname..."
echo $host > /proc/sys/kernel/hostname
```

```
echo $host > $mnt/etc/hostname
echo "Done"

########################################
# CHROOT                               #
########################################
echo "Chrooting into Debian!"
busybox chroot $mnt /bin/bash

########################################
# After exiting from Debian            #
########################################
echo -n "Umount filesystem..."
busybox chroot $mnt umount /proc /dev/pts /sys
umount -l $mnt
echo "Done"
```

Once we were into Debian after the first chroot, we installed all the services we needed:

- SSH : necessary to initialize the MPI daemon on all the nodes in the cluster, and useful to access the phones from a personal computer (writing shell commands on a 2" soft keyboard is not really comfortable)
- MPICH2 : implementation of MPI (see Section 3.1)
- HPL : Linpack benchmark (see Section 3.2)

A "ready to use" Debian image is available on the project repository [4]. Anyway we have written a complete guide (available on the repository as well) that goes step by step into the details of a correct installation of MPICH2 and HPL, since it's a pretty long process that you have to go through if you want to prepare your own image. Here we simply highlight some interesting points that may be source of errors:

- SSH daemon doesn't start automatically after the chroot, since it's supposed to start after the boot of the operating system, and it's not our case as we already explained. A solution to start it automatically after the chroot is to place the shell command into the *.bashrc* file; anyway there is a bug in this file [5], that makes some daemons not to work properly when they print something on the */dev/stdout*. We solved it by redirecting the output of the daemon initialization to */dev/null* file.
- Also the MPI daemon does not start automatically, but instead of adopting the same solution of the last point, we have decided to write a more complete script to run the daemon once on all the phones in the cluster. This script is presented in Section 5 of this report. However if wish to run an MPI program on a single node, you may want to run it manually typing *mpd –daemon* in the shell.

5 Cluster Setup

Unfortunately MPI does not support the addition of new nodes at runtime. This issue requires a manual setup of static IP addresses of the nodes, and it's exactly the solution adopted in

many case studies of big computer clusters that are basically never moved from their location. On the other hand it's evident that this process gets to be boring very soon when we are dealing with mobile devices that connect to different networks many times a day. For this reason we decided to write a C program that allows the phones to get a dynamic IP from a common router that uses DHCP, and the master node asks for all the IPs that have to be included in the cluster just before initializing the MPI daemon, and sends them through an SSH tunnel all the necessary configuration files needed to run the benchmark. The program also distributes the master's public SSH key to every slave, since MPI requires an initialization through SSH without authentication (to notice the huge limitation for a "user friendly" cluster configuration that could be brought into the real world). This program's source code is already included in the "ready to use" Debian image under the */root* directory and can be compiled directly on the phones with the GCC compiler. However it is also available on the project repository too.

5.1 Benchmark Tuning

Tuning the benchmark is extremely important in order to reach the best performance possible. It is made by modifying the HPL.dat input file that is located in the same folder of the xhpl executable of the benchmark. There are a lot of possible settings, but two parameters are the most important ones and need a non-obvious study for choosing the right value: the order of the matrix (N) and the size of the blocks (Nb).

Matrix Order Choice: The order of the matrix is an extremely important choice because it determines the amount of memory used by the nodes for computing the solution of the system. The amount of memory used by HPL is essentially the size of the coefficient matrix ($N * N * 8$ bytes, double precision) [7]. Best performance can be obtained using the largest problem size, filling the memory as much as possible. A first guess, since the OS needs some extra memory to work it, is about the 80% of the total system memory. Total amount of memory in the heterogeneous cluster has been calculated making the sum of the available amount of memory in bytes of each node of the cluster. In an homogeneous one, where nodes are all perfectly equal, the matrix order is simply the inverse of the formula that provides the percentage of memory used:

$$N = \sqrt{\frac{\#nodes * memory * 0.80}{8}}$$

Each HPL test requires many hours to complete, but after few tests it can be noticed that the marginal performance gain begins to drop with the increase of the matrix order, while the time spent continues to grow with a $O(n^3)$ complexity. A study is needed to choose a acceptable value for the size of the matrix. A graph helps us to fix a reasonable amount of memory used by the computation. The figure 5.1 shows the results of the study for the android smartphone Samsung S5570:

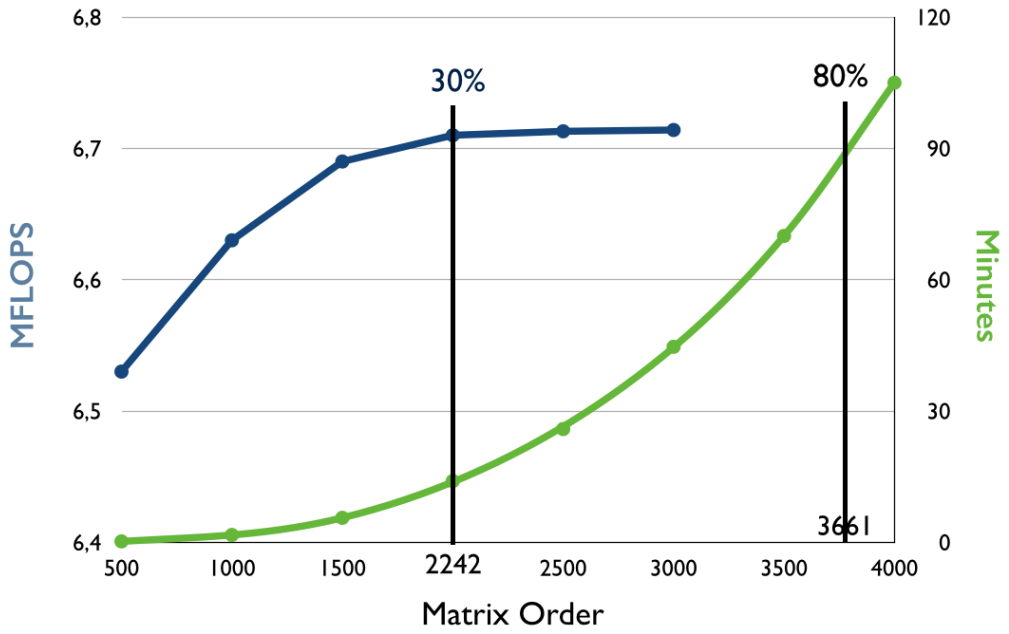

Figure 5.1: Samsung S5570 Matrix Size choice

As shown in the figure, at 30% of memory the marginal gain becomes smaller with the growth of the matrix size, giving an angle in the curve. The "elbow" method can be applied to set the parameter at the order value of the angle. We have thus calculated the order of the matrix using the same formula above, but adapted to the heterogeneous cluster:

$$N = \sqrt{\frac{\#nodes * memory * 0.30}{8}}$$

Similar results have been obtained with other devices available in the laboratory and used for composing the cluster studied. The same criterion has been used for all the tests made during the study, for all the configurations from one to n devices. Note that with this setting, we fill a percentage of the total available memory of the cluster, and not a portion of the available memory of each single device. Since the level of memory occupied in each node is relatively low, we are sure that adding devices to the cluster (and so increasing the problem size) does not make any device swap data onto the mass memory, which would reduce performance by a large factor. This obviously does not apply to clusters in which you want to test the peak performance, in that case an MIT study has shown that the problem size has be tuned on the machine with the least available memory [8].

BLOCK SIZE CHOICE: There is not a precise rule for setting the size [bytes] of the blocks in which the matrix has to be partitioned. The smaller the block size, the better the workload balance among nodes because blocks can be dealt better onto the grid of processes (see section 3.2), assigning to each node about the same amount of data to handle. On the other

hand, smaller blocks might limit the performance because no data reuse would occur in the highest level of he memory hierarchy [7]. In addition, the number of messages passed would also increase because of the smaller size of each panel to factorize. Good block sizes are almost always in the 32..256 bytes interval and depend on the specific system. One should try some tests to find out the best size for the cluster to study. We have tested some typical values (64, 128, 160, 192) on different devices alone, and on different configurations of the system. We have noticed that for our cluster (heterogeneous system - few number of nodes) the differences among results of different tests, with different blocks' sizes on the same configuration, are negligible due to a very low standard deviation. Besides, our goal is to study the behavior of our heterogeneous cluster gradually increasing the number of devices involved in the computation, and measuring the performances given by the benchmark. Moreover, we want to compare the results of the different configurations in order to understand how (and if) the performance increase with the number of devices. For these reasons, the choice of only one intermediate value of block size (128), without stressing the system to maximum performance possible, seems reasonable and all our tests, for all the configurations, use this block size value. The table below shows the results of the tests made on the Android smartphone HTC Wildfire and Samsung Galaxy SII, both in a single-nodes configuration and in a two-nodes configuration. As one could expect, the standard deviation increases with the number of devices in the system.

	Blocks Size [byte]				
Device	**64**	**128**	**160**	**192**	**St. dev**
HTC	6,046	6,046	6,079	6,075	0,016
Galaxy S2	18,82	18,87	18,74	18,82	0,054
HTC + S2	11,67	11,38	12,02	11,77	0,147

Table 5.1: Block Size Choice: results given in MFLOPS

HPL.DAT: HPL.dat is the input file to tune the benchmarks in order to fit as much as possible the cluster configuration to study. It has to be located in the same directory of the xhpl benchmark executable. The file is 31 lines long and each of them has a precise meaning. All the parameters are printed in the output generated by the executable. Detailed instructions about the meaning of each line can be found on the Netlib website[6,7]. Note that P and Q are the numbers of rows and columns of the grid of processes in which blocks of the matrix are dealt onto. The total number of processes involved into the benchmark, and launched from the command line to start the computation, has to be equal to the value PxQ. Below is reported the HPL.dat file used for all the tests described in this report (except for parameters to be tuned accordingly to the cluster configuration, marked with "-").

```
netlib.org/benchmark/hpl/tuning.html
Innovative Computing Laboratory, University of Tennessee
HPL.out      output file name (if any)
8            device out (6=stdout,7=stderr,file)
-            # of problems sizes (N)
-            Ns
```

```
-            # of NBs
-            NBs
0            PMAP process mapping (0=Row-,1=Column-major)
-            # of process grids (P x Q)
-            Ps
-            Qs
16.0         threshold
1            # of panel fact
2            PFACTs (0=left, 1=Crout, 2=Right)
1            # of recursive stopping criterium
4            NBMINs (>= 1)
1            # of panels in recursion
2            NDIVs
1            # of recursive panel fact.
1            RFACTs (0=left, 1=Crout, 2=Right)
1            # of broadcast
1            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1            # of lookahead depth
0            DEPTHs (>=0)
2            SWAP (0=bin-exch,1=long,2=mix)
64           swapping threshold
0            L1 in (0=transposed,1=no-transposed) form
0            U  in (0=transposed,1=no-transposed) form
1            Equilibration (0=no,1=yes)
8            memory alignment in double (> 0)
```

6 Experimental Results

To sum up before proceeding, the two steps we have made to be ready to run the benchmark are:

- chroot each phone into debian (using the script presented in Section 4)
- boot the MPI daemon on every device (running on the master node the script presented in Section 5)

Now the cluster should be ready to use (a further check can be done by typing *mpdtrace* on the master node). To run the benchmark, the command we typed in our environment is:

```
root@phone1# mpirun -recvtimeout 100 -n 7 /usr/local/hpl-2.1/bin/squeeze/xhpl | tee out.txt
```

The meaning of this command is:

- mpirun : executes the program using the MPI protocol
- recvtimeout 100 : sets a larger timeout (default is 20 seconds). It's not necessary, but is a good practice especially when you are running a program on a high latency link as WiFi.
- n 7 : indicates the number of processes to run in the cluster, and it must match with the sum of the processes allocated on each phone that are asked by the script described in section 5. In this case we ran 7 process on 7 different nodes.
- /../xhpl : is the path (common to each node) of the benchmark executable file.

- tee out.txt : just a good practice to save the output into a txt file.

We first started testing each single device running the benchmark locally. The results that we have obtained are listed in Table 6.1.

N	Device	MFLOPS
1	Samsung S5570	6,71
2	HTC Wildfire	5,99
3	LG-P350	6,71
4	LG-P500	6,56
5	Galaxy Tab 2	15,94
6	LG-P500	6,56
7	Samsung Galaxy S2	19,03

Table 6.1: Local performance of the devices

For the sake of completeness, we report here an example of a result given by the benchmark on SamsungS5570. All the other results are available on the project repository[4]:

```
================================================================================
HPLinpack 2.1  --  High-Performance Linpack benchmark  --   October 26, 2012
Written by A. Petitet and R. Clint Whaley,  Innovative Computing Laboratory, UTK
Modified by Piotr Luszczek, Innovative Computing Laboratory, UTK
Modified by Julien Langou, University of Colorado Denver
================================================================================

--------------------------------------------------------------------------------

- The matrix A is randomly generated for each test.
- The following scaled residual check will be computed:
      ||Ax-b||_oo / ( eps * ( || x ||_oo * || A ||_oo + || b ||_oo ) * N )
- The relative machine precision (eps) is taken to be               1.110223e-16
- Computational tests pass if scaled residuals are less than                16.0

================================================================================
T/V                N    NB     P     Q               Time                 Gflops
--------------------------------------------------------------------------------
WR01C2R4        2242   128     1     1            1120.02              6.715e-03
HPL_pdgesv() start time Wed Jun 11 09:04:59 2014

HPL_pdgesv() end time   Wed Jun 11 09:23:39 2014

--------------------------------------------------------------------------------
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)=        0.0061722 ...... PASSED
================================================================================

Finished      1 tests with the following results:
              1 tests completed and passed residual checks,
              0 tests completed and failed residual checks,
              0 tests skipped because of illegal input values.
--------------------------------------------------------------------------------

End of Tests.
================================================================================
```

6.1 One process per device

After testing each single device, we tested a 2 nodes cluster by joining the first phone with the second one; a 3 nodes cluster by joining the first, the second and the third, and so on...than we repeated the entire procedure a few times, just to be sure that the results we obtained were reliable enough. Actually we noticed that the benchmark is very reliable in that sense: repeating the same test on the same cluster configuration gives results with a really low standard deviation (running 5 tests on the first 2 phones, we obtained $stdev = 0,010\ MFLOPS$).
The ideal upper bound performance we could obtain from such a cluster, is trivially the sum of the MFLOPS of each single device obtained by allocating one process for each phone. For example the ideal performance of all the 7 devices in our cluster is given by $6,71+5,99+6,71+6,56+15,94+6,56+19,03 = 67,50\ MFLOPS$. Of course what we have to expect is something lower than that, since we are dealing with latencies of the network and of the communication protocol itself; but how lower is it?

RESULT 1: The result we have obtained is shown in the graph below, and really deserves some comments:

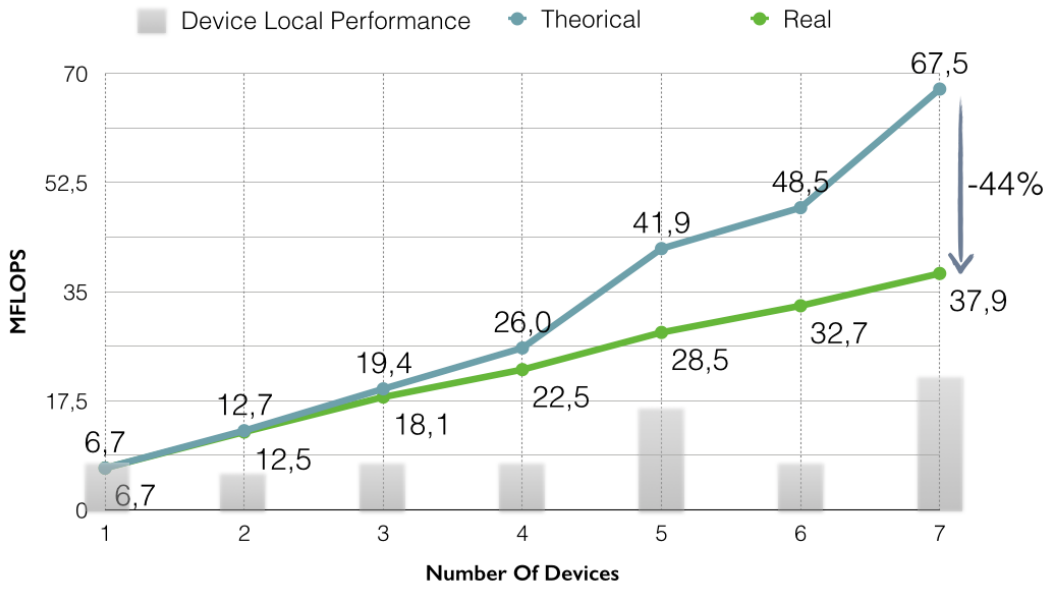

Figure 6.1: Result 1

Up to phone number 4, the result seemes to be reasonable, since we have only some percentage points of performance loss as we expected. The curious point is what happens adding the 5th device (Galaxt tab 2) which is about 3 times more performing (always in terms of FLOPS) of the previous 4 ones: it behaves exactly like its performance was comparable to their one. It's making the cluster loose about the 20% of performance with respect to the

ideal curve, instead of the 3-4% we were loosing up to now. Inspecting further, we see that the 6th device (LG-P500) behaves like it should, making the cluster loose again just a few percentage points of performance. Adding the 7th phone instead (Samsung Galaxy S2), which is comparable to the Tablet, the cluster looses again a lot of MFLOPS. It seems like the less performing devices are the bottleneck of the ring, and that's totally reasonable since HPL algorithm is CPU-bound, and the most performing devices have to wait for the lower ones while working on the LU factorization. Remember that the phones communicate during the computation to get intermediate results from the others.
The confirmation that the bottleneck is not any other factor (i.e. the network), is exposed in Section 6.2.

6.2 One process per core

From the previous analysis, we noted that the improvement in MFLOPS adding two more powerful devices results in a significant loss of performance with respect to the ideal value. The idea is that the least performing node limits the computation of each device, but, since the first 4 smartphones are comparable in terms of CPU, the figure 6.1 does not show explicitly this issue. Up to now we have tested the cluster allocating the same workload on each phone; now we allocate one process per core, assigning in this way two processes to the Galaxy Tab and to the Galaxy S2, both dual-core machines. Making their cores work in parallel, their local performance also grows:

N	**Device**	**MFLOPS**
5	Galaxy Tab 2	30,35
7	Samsung Galaxy S2	28,25

Table 6.2: Performances exploiting 2 cores

Result 2: The overall performance of the cluster increases, but again, we clearly see in Figure 6.2 that the loss of MFLOPS is huge:

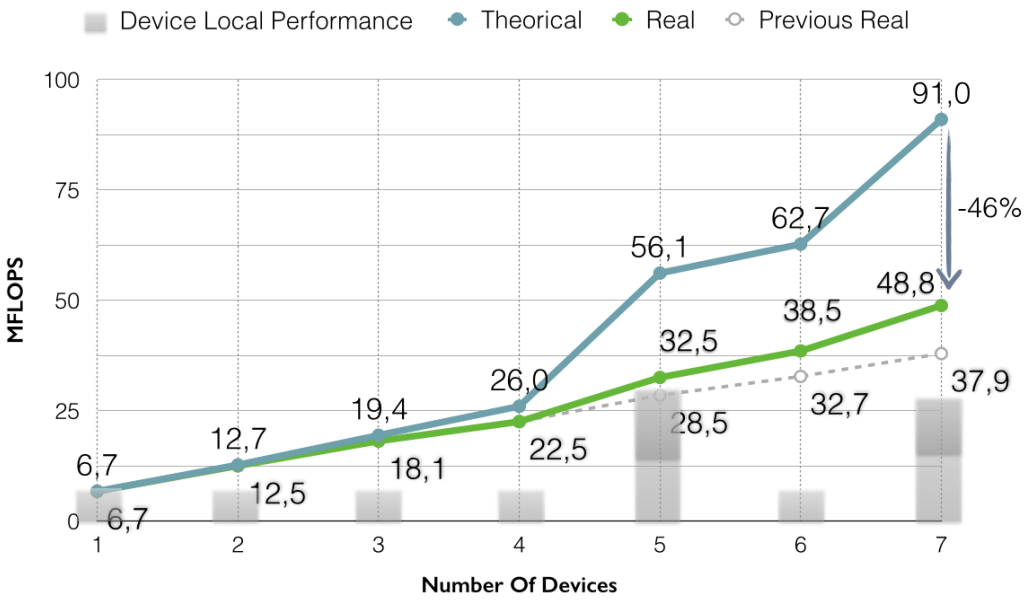

Figure 6.2: Result 2

The least powerful device (single-core, HTC) forces all cores of all the devices to work at its speed and the cluster behaves as a system composed by 9 HTC Wildfire smartphones only (1 per each core that is now computing). To stress our hypothesis we have decided to test a cluster with only these two devices, without a less powerful smartphone, to understand if they can express all their computational power. As expected, when not limited, they reach a value of 51,26 MFLOPS, more than the performance of the cluster made up of all 7 devices, included the tablet and the Galaxy S2 themselves. To provide a further proof of the theory, we have added the HTC and tested again this last configuration. The result is exactly the one expected: the HTC limits the computation of all the cores and the value of performance is poor. The figure 6.3 shows the performance of the two dual-core devices, also after the insertion of the HTC (smartphone that strongly limits the computational power of every core). The final value is indeed similar to the one ideally reachable by a cluster composed only by devices with performance similar to the HTC ones:

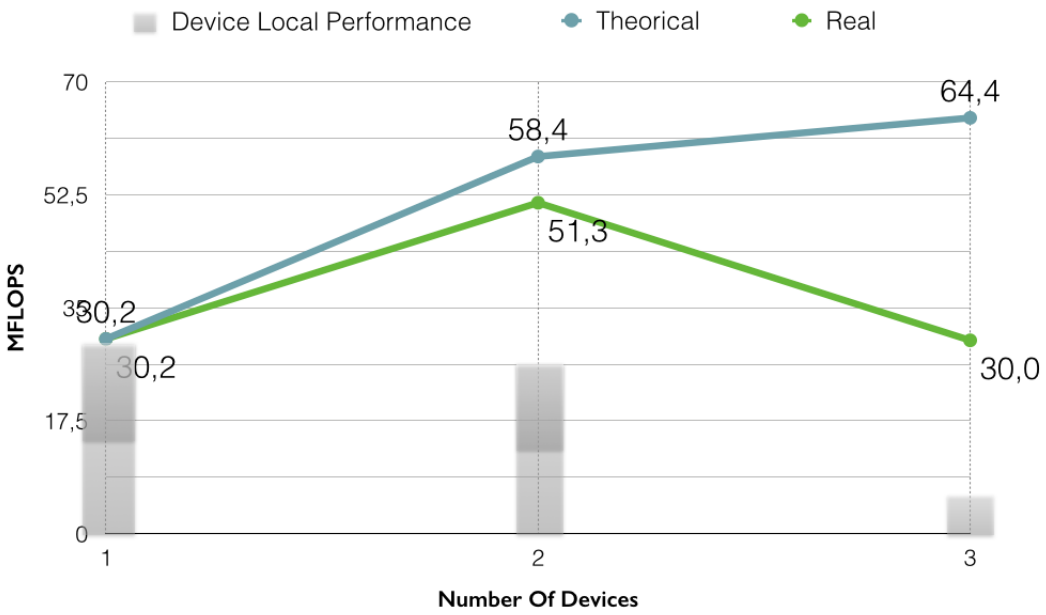

Figure 6.3: Result 3

7 Conclusion

H-Droid Cluster study shows that an heterogeneous cluster of Android devices is certainly feasible, and that there is a significant gain in performance incrementing the number of devices, even if results are strongly bounded to the least powerful machines. However, a mobile cluster is still not ready for an efficient implementation in a real-life application, even if the scripts for booting the Debian distribution and setting up the communication allow us to build a cluster easily (i.e. adding devices replicating a ready-to-use Debian image on sd-cards). We may in any case imagine future ambitious works in this direction.

8 Future Works

H-Droid Cluster opens up new and interesting perspectives of future studies and real-life applications. Since Android devices are spread all over the world, a cluster offers a potentially infinite computational power that can be exploited in huge scientific CPU-bound calculations. This idea has already been proved to be effective by IBM with the World Community Grid and the University of California at Berkeley with the BOINC platform, on which the volunteer grid-computing infrastructure SETI@home [9] is based.
Modern homogeneous supercomputers machines are usually connected through high-speed large-bandwidth interconnections like InfiniBand, while volunteer grid infrastructures are linked with common wi-fi connections or wired ones. Benchmarking can be a valid option for studying everyday communication protocols, like USB, by stressing the network with a huge

number of devices, which can reveal critical limitations and bottlenecks.
A further interesting development, of a cluster composed by Android devices, exploits to the limit the characteristic of large diffusion of such mobile technology, it would be possible to analyze data generated directly by the devices. Think about a match in a stadium or at any kind of public event, where almost every person is taking pictures and videos, or consider an environment full of people like a train: all that amount of data from devices' cameras and sensors could be collected to generate a tri-dimensional panorama of the whole stadium or to guarantee the safety of the crowd or also to map the temperature in every zone of the carriage, resulting in a perfect regulation of the heating or of the air conditioning, and in a consequent high level of power efficiency.

Figure 8.1: Real Life Scenario

REFERENCES

[1] Jonh, Heggestuen (15-Dec-2013). One In Every 5 People In The World Own A Smartphone, One In Every 17 Own A Tablet. On *BusinessInsider-Tech Section (Dec-2013)*

[2] Pfister, Gregory (1998). In Search of Clusters (2nd ed.). Upper Saddle River, NJ: Prentice Hall PTR. p. 36

[3] Felix Busching, Sebastian Schildtf, Lars Wolf (2012). DroidCluster: Towards Smartphone Cluster Computing - The Streets are Paved with Potential Computer Clusters. At the *32nd International Conference on Distributed Computing Systems Workshops*

[4] Repository: https://bitbucket.org/necst/casula-bruzzo-hpps2014-mcluster-mpi

[5] Ref: http://bugs.x2go.org/cgi-bin/bugreport.cgi?bug=327

[6] HPL Tuning How To: http://www.netlib.org/benchmark/hpl/tuning.html

[7] HPL Tuning Useful Tips: http://www.netlib.org/utk/people/JackDongarra/faq-linpack.html

[8] Daniel Loreto, Erik Nordlander, Adam Oliner Benchmarking a Large-Scale Heterogeneous Cluster, MIT, 2005

[9] David P. Anderson Public Computing: Reconnecting People to Science,2003

[10] J. Dongarra, P. Luszczek, A. Petitet, The linpack benchmark: Past, present and future, 2003

[11] HPL Algorithm: http://www.netlib.org/benchmark/hpl/algorithm.html