Setting up an ARM-based micro-cluster and running the WRF weather model.

Introduction:

In this paper, we describe the motivation, mechanics, and results of attempting to build a very small cluster based on commercial off-the-shelf ARM technology, and then "porting " and running the WRF weather model in a variety of modes. Our target system consists of several ARM-based computers, with each node consisting of two or more ARM cores, interconnected by some sort of standard network. Power consumption of the ARM systems is of particular interest.

The paper is organized as follows. Section 1 describes the motivation for this activity. Section 2 details the hardware devices chosen as a basis for the work. Section 3 describes the effort to set up a functioning system consisting of Operating System and Compilers, and lists the version of WRF chosen for the work. Section 4 describes the effort required to port and build the WRF weather code. Section 5 describes the results obtained, comparing the performance to a similarly configured x86_64 "Atom" cluster. Section 6 provides some conclusions and discusses future work.

Section 1) Motivation

The current High Performance Computing (HPC) market is dominated by x86_64-type based compute devices. These devices have their roots in the so-called "attack of the killer micros" which, starting in the early to mid 1990s, supplanted and effectively removed from the marketplace, the previous generation of "vector"-based computers, such as those marketed and sold by Cray Research, NEC, Fujitsu, Hitachi, and others. This was achieved through large clusters of relatively low power, and inexpensive, x86 devices, which evolved over the last twenty years in to today's typical x86_64 chip. This was accompanied by the rise of software to take advantage of these clusters of devices, such as MPI, and standards such as OpenMP, to take advantage of the increasing core count of today's x86_64 offerings.

The current generation of x86_64 devices has moved far from their roots as "swarms of killer micros," with the transistor counts of some devices reaching into the billions. They are relatively expensive (> $2,000 per processor) power hungry (> 150 Watts per processor) and complex to program effectively. Note that these were the identical charges made against the previous generation of Vector machines, albeit at a different quantum level of cost and power consumption.

The current popular GPGPU devices being investigated for HPC have similar cost and power issues, while the complexity of programming these devices surpasses anything previously encountered.

The author speculates whether the current reign of high power, high cost, complex x86_64/gpu devices is now subject again to another "attack of the killer micros".

During their ascendancy, the x86_64 devices have adapted and co-opted an architecture similar in some respects to the old generation of Vector machines. X86_64 devices support "vectorization" of code. On the face of it, this appears the same as that on the previous generation of Vector machines. In fact, Compiler issues, and restrictions regarding ability to vectorize a code are basically identical. There is, however, a crucial difference. The SIMD employed in the typical 86_64 device does vectors differently from the previous Vector generation. Modern vectorization is achieved through the use of extremely wide "vector registers", mated with floating point functional units that exactly match this width. These wide registers are currently going through an explosion, going from 128 bits (i.e. Intel "Harpertown"), to 256 bits (i.e. Intel "Ivy Bridge"), to 512 bits (i.e. Intel "Xeon PHI"), and, one can speculate, as far as 1024 bits in the near future.

There are two disadvantages with this approach. The first is, if the code cannot be vectorized, then increasingly large fractions of available floating point performance are entirely unavailable to the application in question. In the case of Intel Ivy Bridge, this fraction is 64/256 = 1/4 for double precision, and 32/256 = 1/8 for single precision. That's only 25% or 12.5% of the rated peak speed of the device. The second disadvantage is that the ability of the memory subsystems to deliver data from system memory to the floating point registers/units does not begin to approach the huge bandwidths that would be required of typical codes. In the case of the Intel Xeon PHI, for example, simple vectorization is not sufficient to get performance. For this, it is also required that a given algorithm be "re-factored" to greatly reduce the memory bandwidth required. For many codes, this may be simply impossible, regardless if the expertise is available. Even if the expertise does exist, it's an expensive proposition to embark on a code re-write where the potential benefits (if any) are impossible to predict beforehand.

The previous generation of Vector machines avoided these two pitfalls by having modest floating-point unit widths, so that the penalty for non-vector code was not so severe. An additional, and expensive, feature was the use of "banked" memory, where successive memory banks, able to be addressed in successive cpu clock cycles, held successive memory words, allowing for ultra-high-speed memory bandwidth, capable of supplying data to the floating point functional units in a manner matching their full capacity requirements. The memory systems of today's x86_64 systems, while several orders of magnitude larger, remain relatively limited in this bandwidth aspect by their rooting in the cheap consumer-commodity class of devices. The compute capability of today's x86_64 devices so far outstrips the ability of the overall system to supply data, that they must be considered fundamentally imbalanced. As mentioned above, the trend is increasing.

A result of all these x86_64 device limitations is that typical user codes achieve effective performance in the sub 1% range of the peak theoretical speeds of the devices. For example, an Intel Ivy Bridge processor may have a peak speed in the hundreds of Giga-flops, while it is not at all uncommon for codes to be lucky to

achieve performance in the mere hundreds of Mega-flops. There is increasingly less expertise at the programming level to be able to achieve vectorization, let alone the ability to refactor codes (if at all possible) to reduce memory bandwidth requirements.

In light of all the above, the question obviously rises of why continue in this direction at all? Why not embrace fully the real meaning of "attack of the killer micros" and re-engage the revolution that initiated the current levels of achievement in the HPC market?

What would be the shape of new devices to re-energize HPC and bring effective performance levels back to the hands of the mass of today's applications and users? They would be low-transistor-count devices. They would have single width floating-units. They would be low power. They would be able to be networked. They would be readily available and cheap.

The device that best fits the above description today is the ARM processor design. Note that the ARM is not a hardware device itself. It is a design for a processor device. In fact, it is many designs, all serving particular markets. ARM is ubiquitous in today's market, and one can envision quite easily, a vendor building a system with a network of ARM design-based devices appropriate for HPC. Some companies are making small steps in this direction.

The motivation for this work is to remove some of the uncertainty of this approach. To do this, we have chosen to port and run, not a suite of small benchmark codes, but a full-blown application, in wide use. To make the work as open and free to discussion as possible, we need a complex application that is completely open-source. The WRF weather model: http://www.mmm.ucar.edu/wrf/users/downloads.html meets all the requirements for this work. We chose version 3.5.1 for the arbitrary reason that it was the most recent version at the time of this work. (fall, 2013)

We want to explore: how hard is it to setup an ARM system? How hard is to port code? Does the code work at all? What sort of performance can we get? How does this compare to x86_64? What does the power usage look like?

Section 2) Choice of Hardware Platform

Our criteria for choosing an ARM platform is that we want at least two nodes, four would be preferable, it should have a two-core ARM processor on each board, be able to be networked, and be cheap.

We chose the Pandaboard ES:

www.pandaboard.org

It is based on the Texas Instrument OMAP 4460 ARM-based device, with 2 ARM cores. Note that the Pandaboard ES and the OMAP 4460 device have many capabilities of no interest to us in the context of this work. We configure the systems "headless", with no access via keyboards, video, or mouse. The only items of interest to us are the dual core processor, the 1 Gbyte of memory, network interface, and minimal USB support for a serial console. In production this Serial/USB connection is not used.

We purchased four Pandaboard ES systems for the work here.

The basic layout of the Pandaboard ES is as follows, as shown in the file:

panda-es-b-manual.pdf

available at www.pandaboard.org

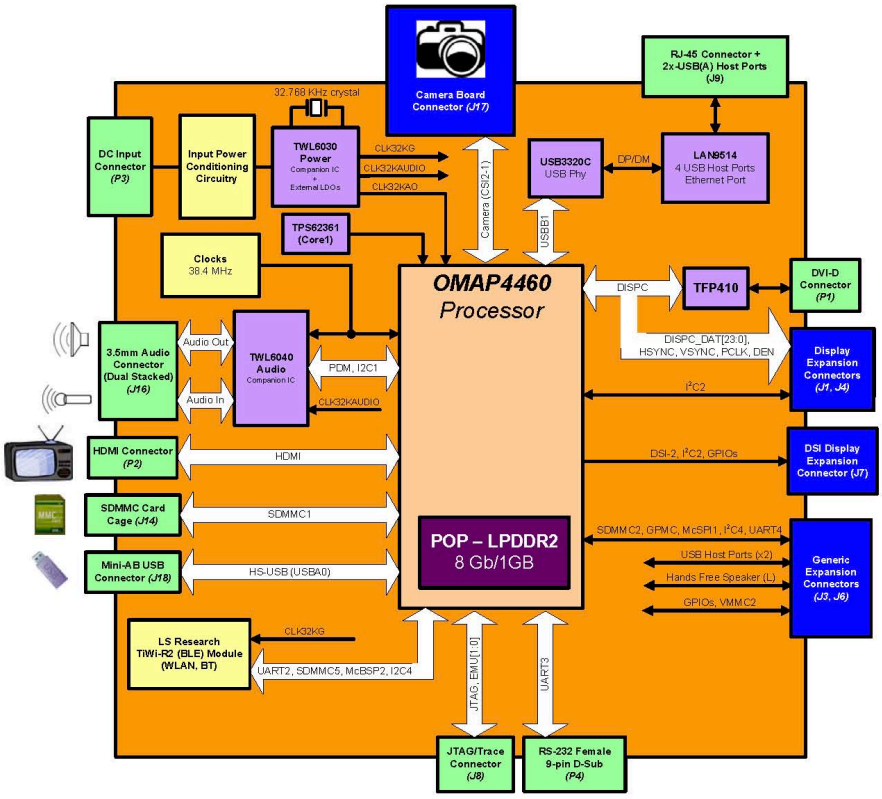
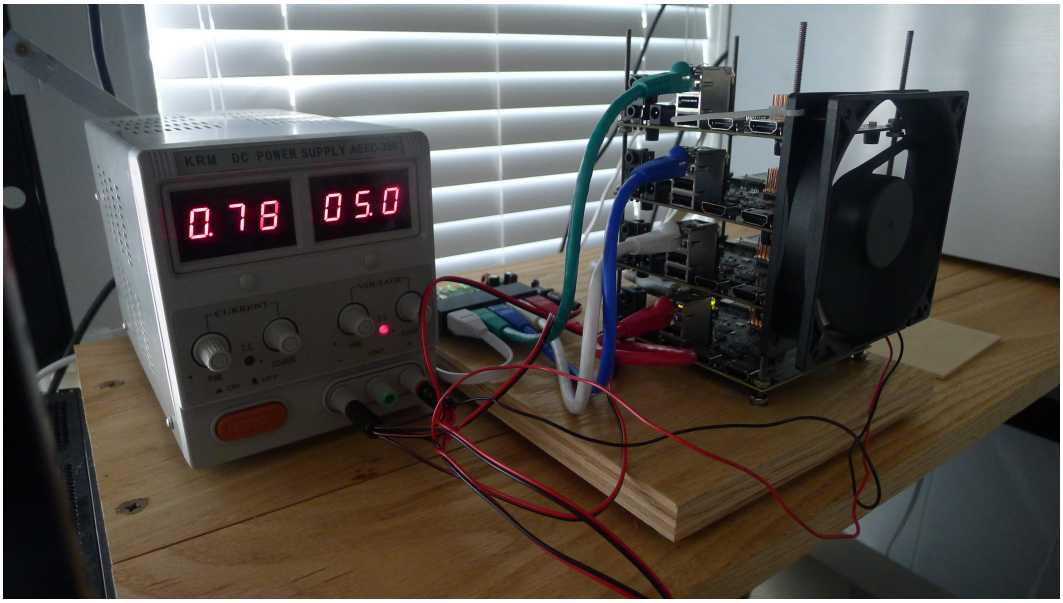


Figure 1 – OMAP4460 Pandaboard ES Architectural Block Diagram

To provide power for the 4 boards, we wanted a power supply that was computer-controllable. With such a power supply, we would be able to read-out the power usage while running codes, which is of prime interest. We chose the very reasonably priced Korad KA3005P.

Section 3) Building a functioning system.

To build our system, we used a base of hardwood plywood about the size of a sheet of paper. We used four 1/16” stainless steel threaded rods through the mounting holes of the Pandaboards to create a small “tower” holding the single-board computers horizontally. The Pandaboards arrive without any heat-sinks on the cpus, so we purchased some appropriately sized small “self-stick” solid copper heatsinks to allow for heat dispersal. The Pandaboards are clocked at 700 Mhz, despite the 1.2 Ghz rating of the OMAP 4460 processor. We use the Korad power supply set at constant 5 volt output to power the system. We use the same 5 volts to run two 12 Volt computer-case fans at low speed. This provides a low-level of noise and sufficient airflow in case of any heating that might occur in the ARM cpus. An early version of this during bringup can be seen here. Note that the Korad is not yet being used, only one board is powered up, and a single 12 V fan is in place.



The view from the other side is shown below. Note the copper heatsinks in place on the cpus, the serial-port adapter being used for bringup, and the SD Digital flash memory card (4 gbyte) which serves as the “hard drive” of the Pandaboard.

Pictured below is the final production version of the micro-cluster. The Korad power supply is in place, there are the two fans for cooling, and we added a small front panel with toggle switches to control power to each of the 4 boards and 2 fans on an individual basis.

In the final configuration used for the reported results, we connect the four 100 Mbit Ethernet cables into our existing consumer-grade Gigabit switch and network.

Software:

We tried several Linux distributions, before settling on a version (12.03) of OpenSUSE available at:

https://en.opensuse.org/HCL:PandaBoard

This has the advantage of matching the current version of OpenSUSE on our suite of systems we use internally, including two Atom-based systems we plan to use for comparison.

Installation of the Linux software was straightforward, following the instructions at the OpenSUSE website. We found the 1 Gbyte memory of the Pandaboards too small for some of the builds we did, so we configured a 1.5 Gbyte swap file on the SD Card of each board. Home file systems were simply NFS-mounted from our normal servers.

We also added the following standard packages using yast2:

```
yast2 --install gcc
yast2 --install gcc-fortran
yast2 --install gcc-c++
yast2 --install gdb
yast2 --install ed
yast2 --install perf
yast2 --install strace
yast2 --install tcsh
yast2 --install rpcbind
yast2 --install nfs-client
yast2 --install xorg-x11-server
yast2 --install xorg-x11-server-sdk
yast2 --install rcs
yast2 --install make
yast2 --install papi
yast2 --install papi-devel
yast2 --install papi-devel-static
yast2 --install xosview
yast2 --install sudo
yast2 --install yast2-network
yast2 --install Modules
yast2 --install man
yast2 --install m4
yast2 --install gmp-devel
yast2 --install mpfr-devel
yast2 --install mpc-devel
```

This provided all the Compilers, libraries, and utilities for basic application development for the work described here, and other work to be described in future reports.

Lastly, for MPI and NetCDF support (needed by WRF), we downloaded and built from source:

mpich-3.0.4
netcdf-3.6.3

Building mpich for ARM was done simply, and natively, on one of the ARM systems with the script:

```
#!/bin/sh
set -x
VERS=mpich-3.0.4
rm -rf ${VERS}
gzip -c -d ${VERS}.tar.gz | tar -xo
cd ${VERS}
export CFLAGS=-g
export LDFLAGS=-g
./configure -prefix=/soft/mpi/mpich-3.0.4 --enable-shared
make
sudo make install
```

While netcdf required:

```
#!/bin/sh
set -e -x
rm -rf netcdf-3.6.3
gzip -c -d ../netcdf-3.6.3.tar.gz | tar -xo
cd netcdf-3.6.3
./configure --prefix=/soft/netcdf/3.6.3
make
make check
```

As can be seen, no exotic options or any special extra effort was required to build either a working MPI or NetCDF. Similar scripts were used for x86_64. The resulting libraries were installed in two parallel directories both mounted as "/soft" on our x86_64 and ARM systems. This allows almost identical scripts to be used for most uses of the software.

All the effort required to port and build WRF for the ARM cluster is contained in the following 32-line script. Line numbers have been added to aid the discussion that follows.

```
 1  #!/bin/sh
 2  . /usr/share/Modules/3.2.10/init/sh
 3  module load mpich-3.0.4 netcdf-3.6.3
 4  set -e -x
 5  module list
 6  for i in 1 2 3 4
 7  do
 8  export NETCDF=/soft/netcdf/3.6.3
 9  rm -rf WRFV3_${i}
10  mkdir WRFV3_${i}
11  cd WRFV3_${i}
12  gzip -c -d ../../WRFV3.5.TAR.gz | tar -xo --strip-
components=1
13  ed arch/configure_new.defaults << EOF
14  g/#ARCH.*gfortran/s/x86_64/armv7l
15  w
16  q
17  EOF
18  ed external/io_int/makefile << EOF
19  75
20  s/;/ -lgomp ;
21  w
22  q
23  EOF
24  export WRFIO_NCD_LARGE_FILE_SUPPORT=1
25  ./configure << EOF
26  ${i}
27  1
28  EOF
29  export J="-j 1"
30  compile em_b_wave FCDEBUG=-g
31  cd ..
32  done
```

This script builds the “em_b_wave” WRF test case in the four basic modes that WRF can build in. They are:

1) Serial mode
2) Shared memory parallel (i.e. OpenMP)
3) Distributed memory parallel (i.e. MPI)
4) Hybrid mode (i.e. MPI with OpenMP

The four modes of compilation correspond to the integer input on line 26 of ${i} to the configure script invoked on line 25.

At line 6, the list of the four compilation modes, 1, 2, 3,4 is provided. Lines 9-12 create 4 separate copies of the WRF directory correlating to the 4 different compilation modes. Lines 13 thru 23 reflect the entire amount of work required to port WRF to ARM. The effort was essentially nil. Line 14 simply changes the string "x86_64" in the configure_new_defaults file to the string "armv7l", which is the machine type returned by the uname command. Line 20 adds the gfortran openmp runtime library, -lgomp, to the library search path in the Makefile, in case it may be needed.

This WRF build script reflects a stunning lack of any effort to port the code to ARM. These minor changes were all that was required to port the code to the ARM cluster, and allow successful runs of the indicated test case. The WRF-supplied "diffwrf" command was used to verify the results of all four versions of the test cases compared to the x86_64 versions. The fact that such a complex code as WRF can be ported and run successfully like this with so little effort is a tribute to the teams at OpenSUSE, GNU, and WRF. Of course the teams at Texas Instruments, Pandaboard, ARM and Intel also deserve credit, as well as countless others.

Power control.

We found no useable software for Linux to control the Korad KA3005P power supply, and so wrote our own. The included manual provided the simple ASCII protocol to use. We created a simple linux command to provide the functions as follows:

```
atomic.site:/home/dpb> korad0
Usage: korad0 option [arg]
Where option is one of:
    on
    off
    id
    status
    vout
    iout
    vset [volts]
    iset [amps]
    rcl1
    rcl2
    rcl3
    rcl4
    save
atomic.site:/home/dpb>
```

This allowed us to power on/off the cluster, and read out the current while running test cases. The power switches on the production front power panel of the cluster allowed us to benchmark the power draw of the fans alone, as well as the cluster in idle, and various other modes. In short spans, we are able to get power information at a resolution of 1 second intervals, with the code to control and read the USB-connected Korad power unit running on a separate system from the ARM cluster. Longer runs at one-second intervals made the host system unstable, and we found a time interval of 4 seconds to be the sustainable choice.

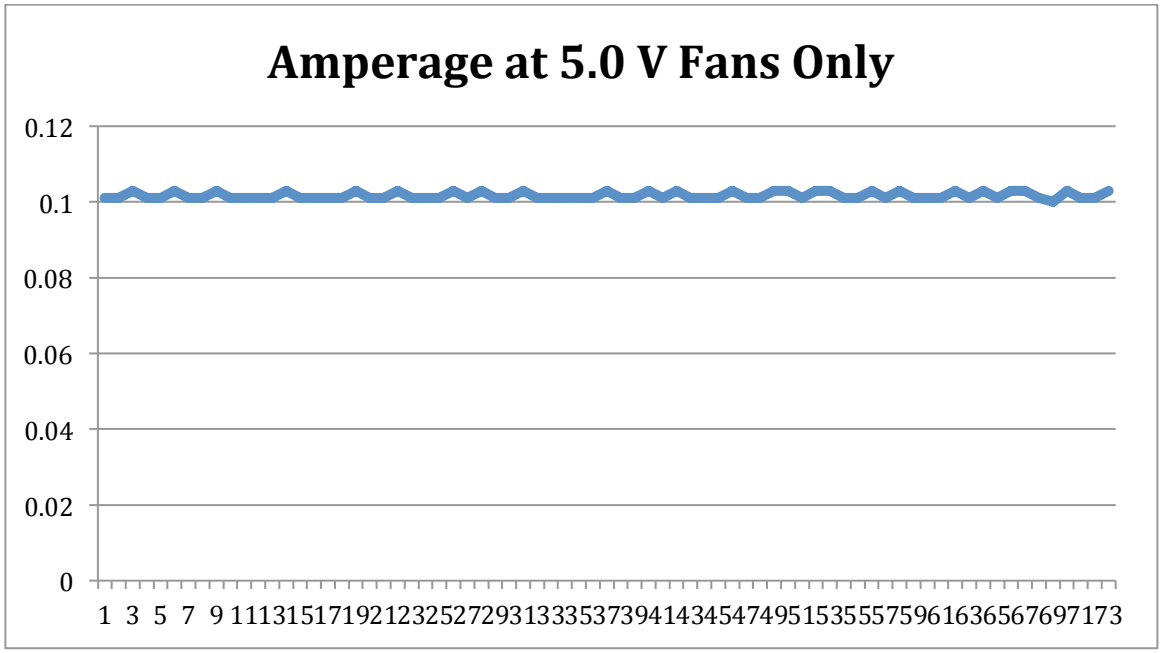
We have encountered no system instability or other issues with the ARM cluster, and found the system, over several months, to be equally as stable as our consumer-grade network of x86_64 systems.

Section 5) Results

We have compared the performance of the ARM systems described above to an Intel Atom D510 system with 4 Gbytes running at 1.666 Ghz. We have two such Atom systems, both connected to the same Gigabit Ethernet network as the ARM systems. To make appropriate comparisons, we have used the same maximum number of nodes (2) and the same maximum number of OpenMP threads per node (2) in the test cases below. Note that we did not measure the power consumption of the Atom systems, but believe them to be approximately 20 watts per node.

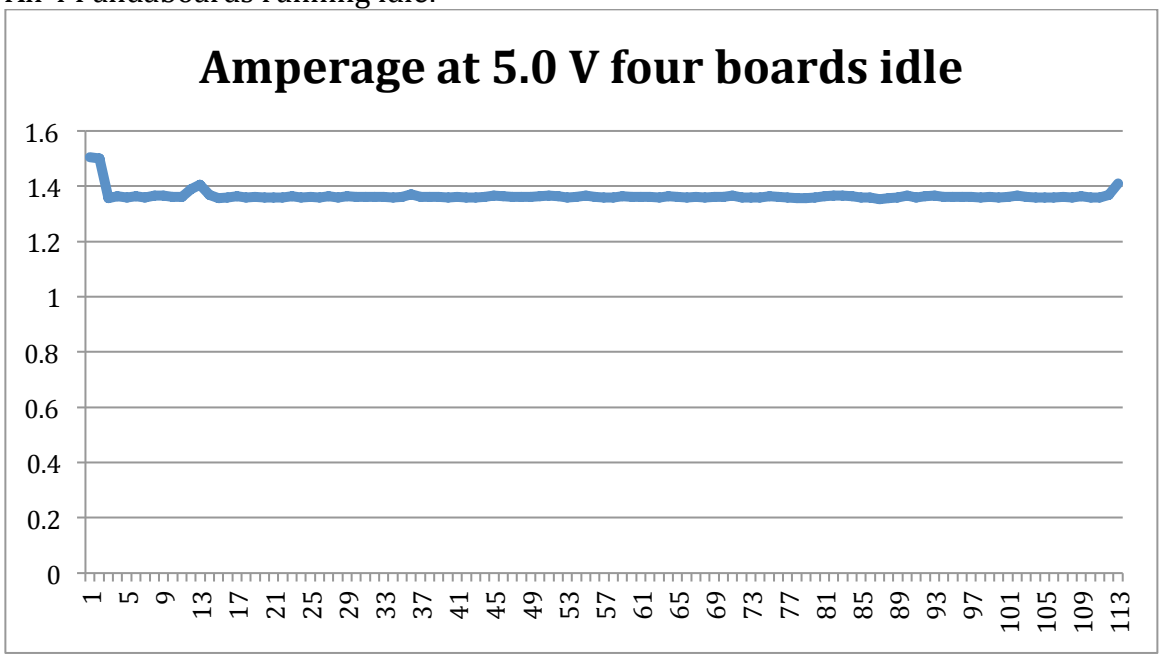
First, some basic ARM power measurements. Note that except for the fan-only data, all the power graphs are for all four Pandaboards, with the fans running, whether or not a given Pandaboard was in use.

Fan-only power data:



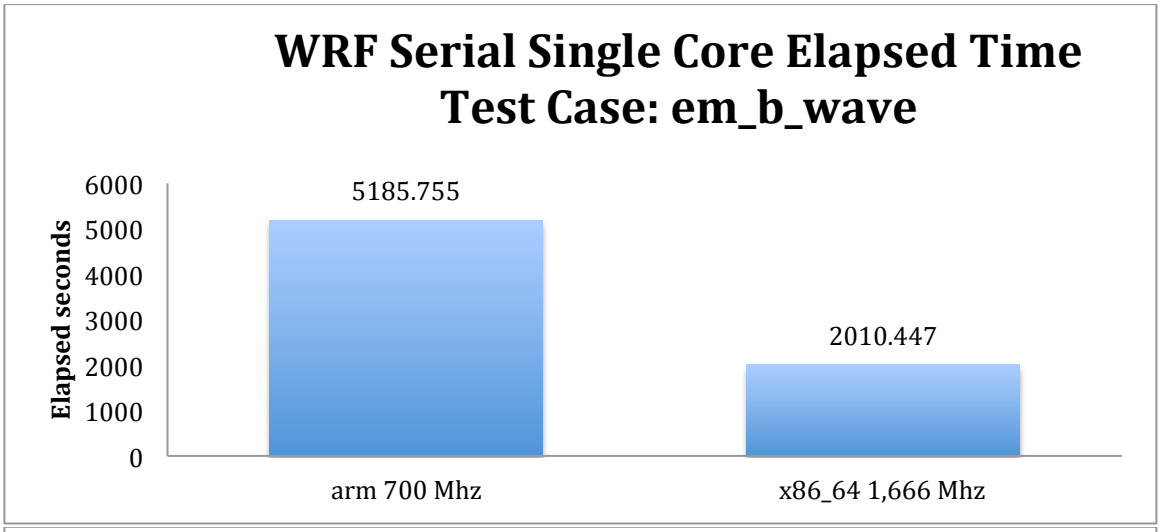
So, the fans consume approximately 0.1 Amps * 5.0 Volts = 0.5 Watts

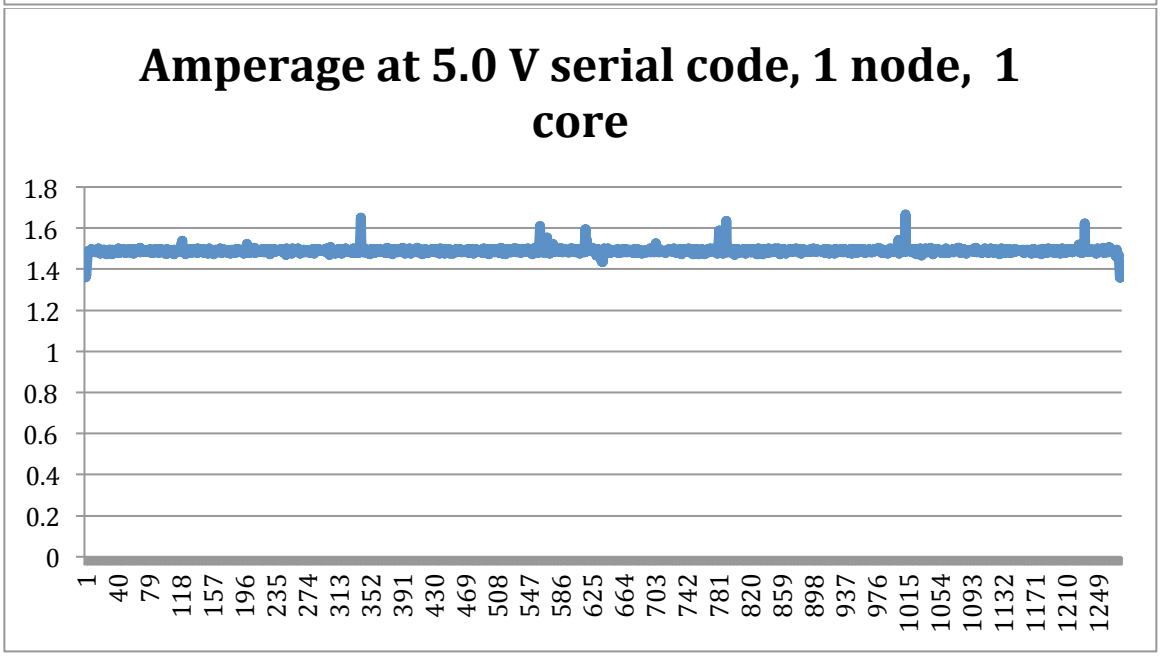
All 4 Pandaboards running idle:



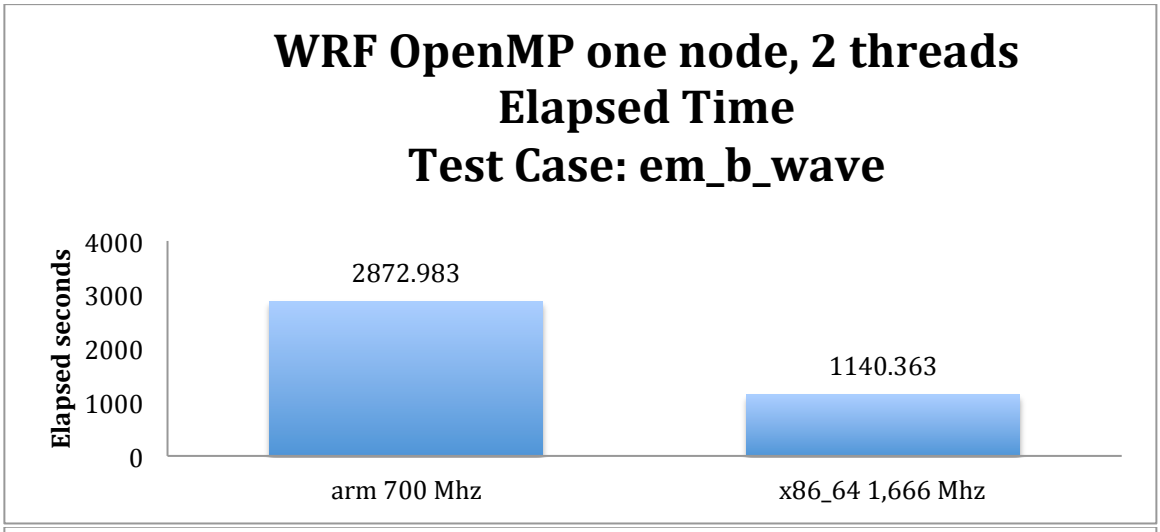
The baseline power consumption for all four Pandaboards is approx. 1.35 * 5.0 = 6.75 Watts.

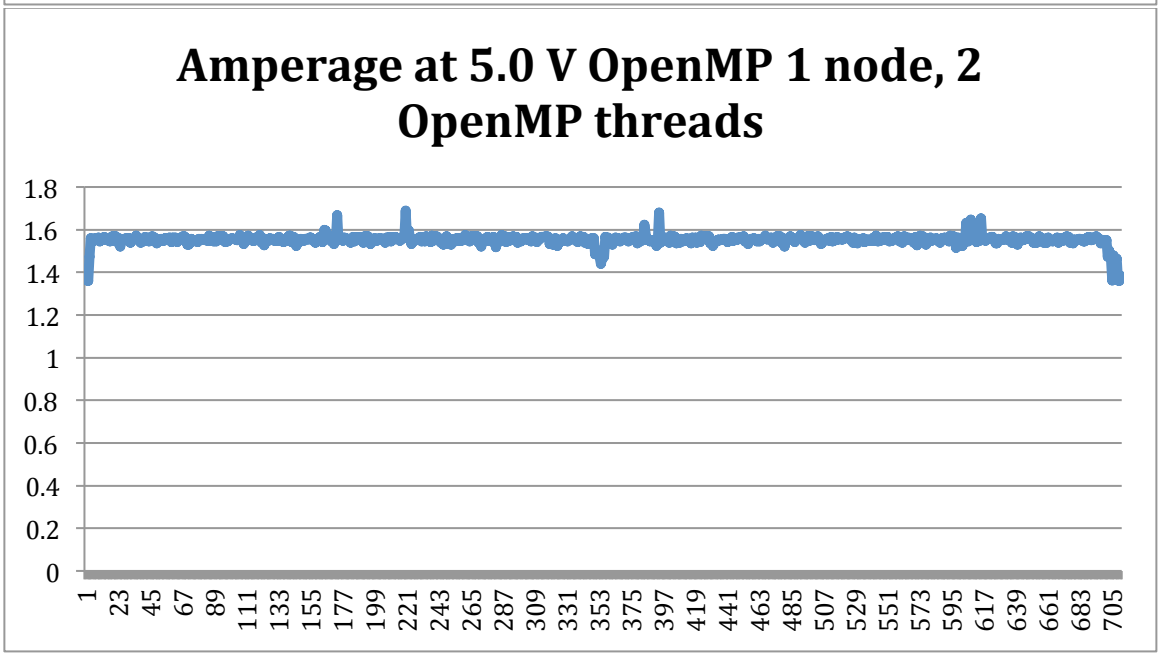
Section 5.1) Serial mode. One node, single cpu.



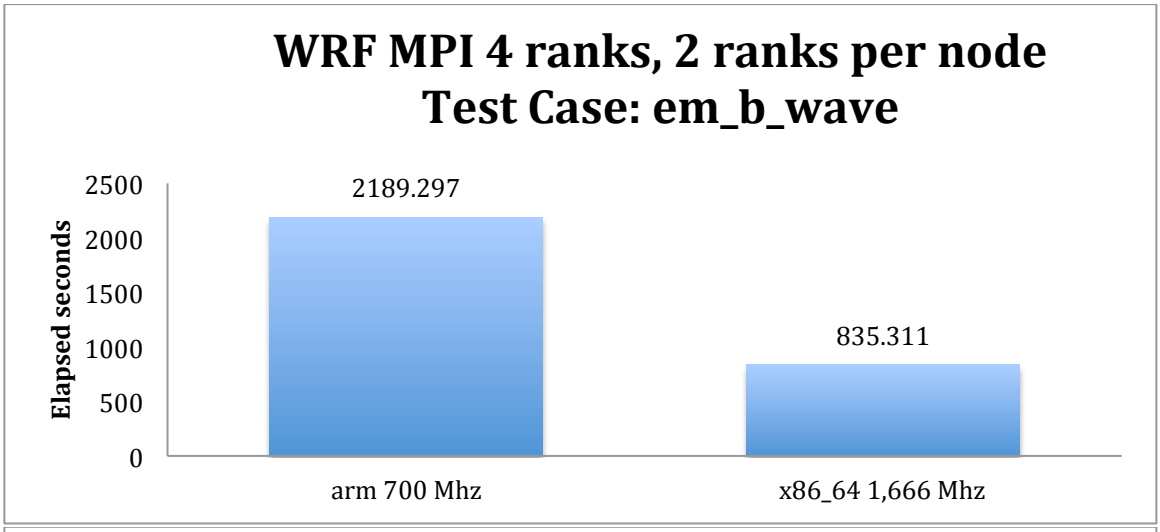


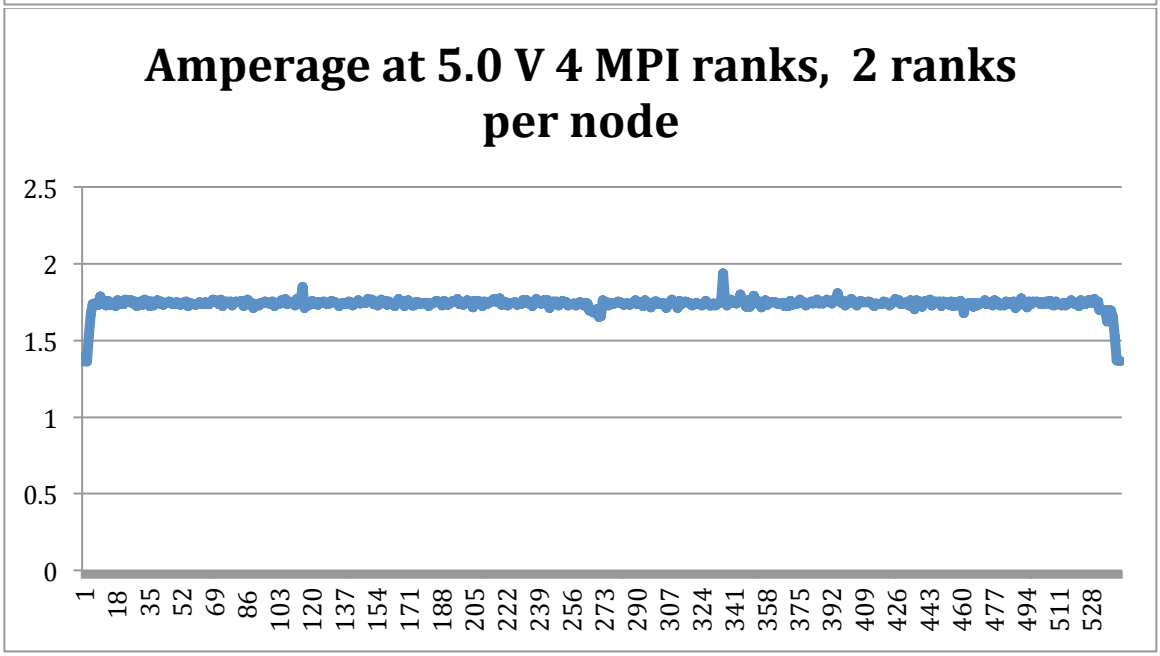
Section 5.2) OpenMP. Shared memory, single node, 2 OpenMP threads



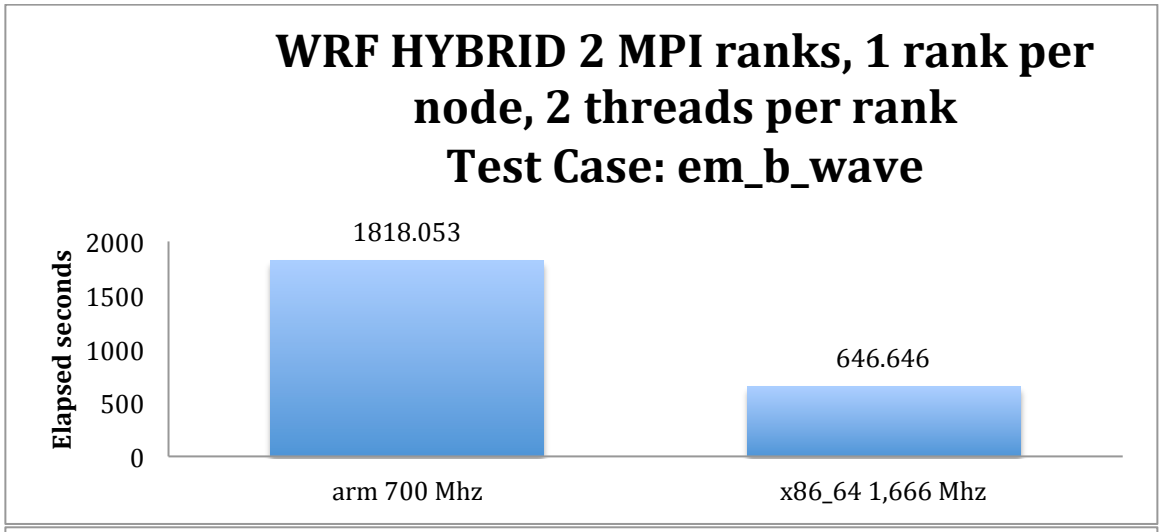


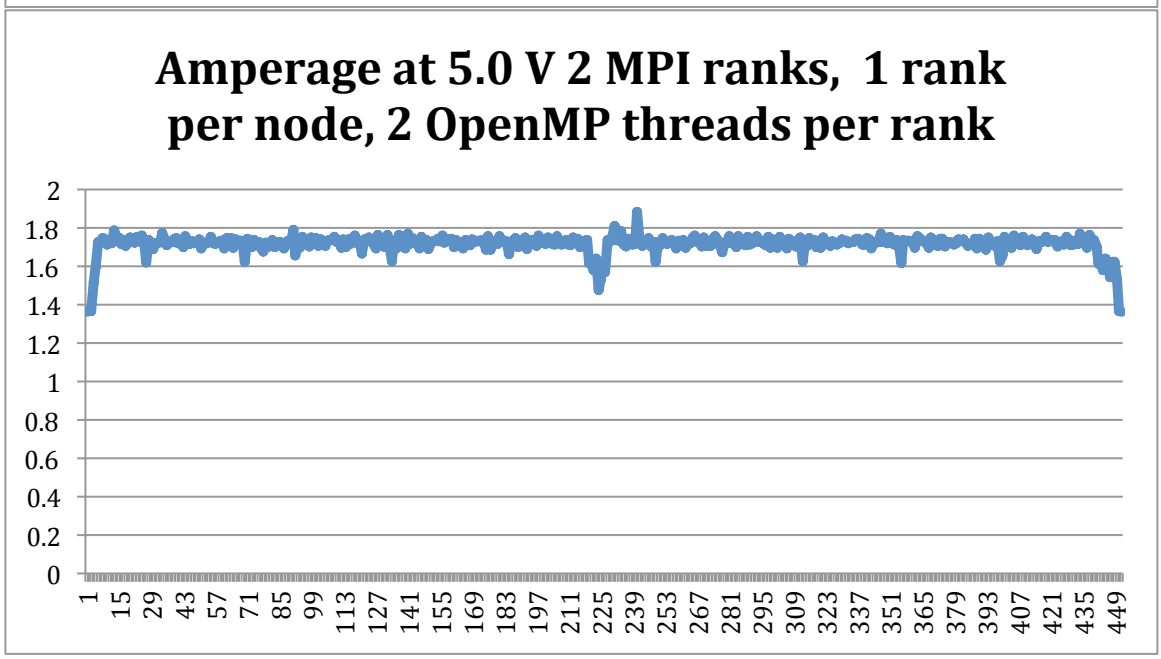
Section 5.3) Pure MPI. 4 single threaded MPI processes.





Section 5.4) Hybrid. 2 MPI Ranks, 1 rank per node, 2 OpenMP threads per rank.





Section 6) Conclusions and future work.

We have made a preliminary study of whether a complex application can be ported to an existing ARM platform. The results for the WRF weather code is that it can be ported with stunning ease.

We have shown performance on an existing ARM platform that is generally within a factor of 2 or 3 of the lowest end of a sample x86_64 platform. Note that the ARM platform under study has it's cpu clock rate artificially limited to 700 Mhz out of a possible 1.2 Ghz.

Power usage has been demonstrated to run the WRF test case with at most one or two watts of power over the idle state of the sample ARM platform, with the base idle power of that platform in the approximate 2 watts per node range.

Further work would consist of running the WRF test case on a more powerful x86_64 system, to get more information about work per watt, and to investigate further the use of floating point by WRF to perhaps obtain power per flop information.

Note that one of the packages we installed on the ARM system was PAPI. We have an in-house performance analysis tool based on PAPI that we have successfully ported to the ARM platform. Further work will also consist of a very detailed performance analysis comparison of the ARM platform and several levels of x86_64 performance platforms.

We conclude that the ARM platform is a viable candidate for development into an effective HPC system, especially when power consumption and ease of programming are taken into account.