Teaching HPC Systems and Parallel Programming with Small-Scale Clusters

Lluc Alvarez
Barcelona Supercomputing Center
Universitat Politecnica de Catalunya
lluc.alvarez@bsc.es

Eduard Ayguade
Barcelona Supercomputing Center
Universitat Politecnica de Catalunya
eduard.ayguade@bsc.es

Filippo Mantovani
Barcelona Supercomputing Center
filippo.mantovani@bsc.es

***Abstract*—In the last decades, the continuous proliferation of High-Performance Computing (HPC) systems and data centers has augmented the demand for expert HPC system designers, administrators, and programmers. For this reason, most universities have introduced courses on HPC systems and parallel programming in their degrees. However, the laboratory assignments of these courses generally use clusters that are owned, managed and administrated by the university. This methodology has been shown effective to teach parallel programming, but using a remote cluster prevents the students from experimenting with the design, set up and administration of such systems.**

This paper presents a methodology and framework to teach HPC systems and parallel programming using a small-scale cluster of single-board computers. These boards are very cheap, their processors are fundamentally very similar to the ones found in HPC, and they are ready to execute Linux out of the box. So they represent a perfect laboratory playground for students experiencing how to assemble a cluster, setting it up, and configuring its system software. Also, we show that these small-scale clusters can be used as evaluation platforms for both, introductory and advanced parallel programming assignments.

***Index Terms*—HPC systems, parallel programming, teaching**

I. Introduction

The importance of High-Performance Computing (HPC) in our society has continuously increased over the years. In the early years, the very few existing HPC systems were based on vector processors specialized for scientific computations, and they were only used by a small number of experts; programmability and usability were not the critical issues at that moment. The trend changed when supercomputers started to adopt "high-end" commodity technologies (e.g., general-purpose cores), which opened the door to a rich software ecosystem. As a consequence programming productivity increased and HPC infrastructure became popular throughout many research and industrial sectors. In the last years, the proliferation of HPC systems and data centers has gone even further with the emergence of mobile devices and cloud services. In the current scenario, the demand for expert HPC system designers, administrators and programmers is higher than ever, and will likely continue growing to keep improving the performance and efficiency of HPC systems in the future.

In the last years, many universities have introduced courses on HPC systems and parallel programming in their degrees. Given the cost of modern HPC infrastructures, the laboratory assignments of most of these courses use clusters that are owned, managed and administrated by the university. This methodology is convenient to teach parallel programming, as the students only need to connect remotely to the cluster to do the programming work for the assignment. However, using a remote cluster prevents the students from experimenting with the design, set up and the administration of such systems.

With the advent of Single-Board Computers (SBCs) for the embedded and the multimedia domains, building a small-scale cluster has become recently extremely affordable, both economically and technically. Modern commercial SBCs for the embedded domain are equipped with processors that are fundamentally very similar to the ones found in HPC systems and are ready to execute Linux out of the box. So, these devices provide a great opportunity for the students to experience with assembling a cluster, setting it up, and configuring all the required software to have a fully operative small-scale HPC cluster.

This paper presents the methodology and framework that we propose for teaching HPC systems and parallel programming using a small-scale HPC cluster of SBCs. This methodology has been successfully used to support teaching activities in Parallel Programming and Architectures (PAP), a third-year elective subject in the Bachelor Degree in Informatics Engineering at the Barcelona School of Informatics (FIB) of the Universitat Politcnica de Catalunya (UPC) - BarcelonaTech. After presenting the PAP course description and environment, the paper gives an overview of the components of the small-scale cluster, which we name *Odroid cluster* after the Odroid-XU4 boards [1] that form it. Then the paper describes the methodology that we use in two laboratory assignments of the course. The first laboratory assignment consists on setting the Odroid cluster up and performing an evaluation of its main characteristics. The cluster setup includes physically assembling the boards, configuring the network topology of the cluster, and installing all the software ecosystem typically found in HPC platforms. In the evaluation part the students discover the main characteristics of the Odroid-XU4 boards, they learn how the threads and the processes of a parallel program are distributed among the processors and the nodes, and they experiment with the effects of heterogeneity. The second laboratory assignment consists on parallelizing an application implementing the heat diffusion algorithm with MPI [2] and OpenMP [3] and evaluating it on the Odroid

DOI 10.1109/EduHPC.2018.00004

cluster. The complete framework presented in this paper greatly facilitates the learning of the design, the setup and the software ecosystem of HPC systems, as well as being a very appealing platform for the evaluation of parallel programming assignments.

The rest of this paper is organized as follows: Section II explains the course and its methodology. Section III gives an overview of the Odroid cluster and its components. Section IV presents the step-by-step process that is followed by the students in the laboratory assignment to set up the cluster. Section V describes the work to be done by the students to evaluate the Odroid cluster and to understand its main characteristics. Section VI then shows how we use the Odroid cluster as a platform for a parallel programming assignment. Section VII reviews other proposals that use of small-scale clusters in courses related to parallel and distributed computing. Finally, Section VIII remarks the main conclusions of this work and presents some future directions to evolve both the Odroid cluster and the assignments.

II. Context, Course Description and Methodology

Parallel Programming and Architectures (PAP) is a third-year (sixth term) optional subject in the Bachelor Degree in Informatics Engineering at the Barcelona School of Informatics (FIB) of the Universitat Politècnica de Catalunya (UPC) - BarcelonaTech. The subject comes after *Parallelism* (PAR), a core subject in the Bachelor Degree that covers the fundamental aspects of parallelism, parallel programming with OpenMP and shared-memory multiprocessor architectures [4]. PAP extends the concepts and methodologies introduced in PAR, focusing on the most relevant aspects of the implementation of runtime libraries for shared-memory programming models such as OpenMP using low-level threading libraries (*Pthreads*), and also explaining distributed-memory cluster architectures and how to program them with MPI. Another elective course called *Graphical Units and Accelerators* (TGA) explores the use of accelerators, with an emphasis on GPUs to exploit data-level parallelism. PAR, PAP, and TGA are complemented by a compulsory course in the Computer Engineering specialization, *Multiprocessor Architectures*, in which the architecture of (mainly shared-memory) multiprocessor architectures is covered in detail. Another elective subject in the same specialization, *Architecture-aware Programming* (PCA), mainly covers programming techniques for reducing the execution time of sequential applications, including SIMD vectorization and FPGA acceleration.

The course is held in a 15-week term (one semester), with four contact hours per week: two hours dedicated to theory/problems and two hours dedicated to a laboratory. Students are expected to invest about five-six additional hours per week in doing homework and personal study over these 15 weeks. Thus, the total effort devoted to the subject is six ECTS credits[1].

The content of the course is divided into three main blocks. The first block has the objective of opening the black box behind the compilation and execution of OpenMP programs [5], explaining the internals of runtime systems for shared-memory programming and the most relevant aspects of thread management, work generation and execution, and synchronization. In a very practical way, students explore different alternatives for implementing a minimal OpenMP-compliant runtime library using Pthreads, providing support for both the work-sharing and tasking execution models. This block takes four theory/problems sessions (mainly covering low-level Pthreads programming) and six laboratory sessions of individual work to program the OpenMP-compliant runtime library. At the end of the laboratory sessions for this block, a class is devoted to sharing experiences and learnings.

The second block has the objective of understanding the scaling of HPC systems with a large number of processors beyond the single-node shared-memory architectures students are familiar with. This block introduces the most relevant aspects of multi-node HPC clusters and explains its main hardware components (processors, accelerators, memories, interconnection networks, etc.). Also, this block takes four theory/problems sessions devoted to analyze in detail how the ratio FLOPs/Byte evolves in the scaling path (i.e., the number of potential floating-point operations per byte of data accessed from/to memory/interconnect). The roofline model [6], which plots floating-point performance as a function of the compute units peak performance, data access peak bandwidth and arithmetic intensity, is used to understand this evolution and its implications on data sharing in parallel programs. Finally, the evolution of energy efficiency (Flops/Watt) is also covered in the theory classes. The laboratory part of this block consists on three sessions in which students physically assemble a small-scale cluster based on Odroid-XU4 boards. They set up the Ethernet network and the Network File System (NFS), as well as install and configure all the software required to execute MPI and OpenMP parallel programs. They eventually evaluate the cluster and its main characteristics using a set of benchmarks. This laboratory work is complemented with a guided learning assignment in which groups of students propose the design of a real HPC system based on commercially available components and with certain performance/power trade-offs and economic budget. This is an excellent opportunity for the students to take a look at real components and to consider cost as one of the important trade-offs in the design of HPC systems. The proposed designs are presented, discussed and ranked in a session with the idea of sharing the criteria used by each group of students.

The last block in the course has the objective of studying the basics of parallel programming for distributed-memory architectures using MPI. This block covers the aspects related to the creation of processes and the different data communication strategies and their trade-offs. This block has a duration of

[1]The European Credit Transfer System (ECTS) is a unit of appraisal of the academic activity of the student. It takes into account student attendance at lectures, the time for personal study, exercises, labs, and assignments, together with the time needed to do examinations. One ECTS credit is equivalent to 25-30 hours of student work.

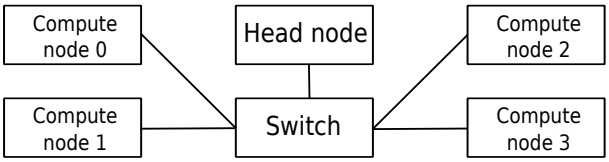


Fig. 1: Scheme of the Odroid cluster.

three theory/problems and three laboratory sessions. In the laboratory students develop a hybrid MPI/OpenMP implementation of the classical heat diffusion problem, evaluating its performance on the Odroid cluster that they have already assembled in the second block of the course. Although the laboratory assignment in this block could be performed in a production cluster available at the university, we preferred to continue in the Odroid cluster.

Throughout the course, the Odroid cluster is used in the laboratory assignments of the two last blocks of the subject. The laboratory assignments are done in groups of two to four students, and we provide an Odroid cluster to each group. We do not allow the students to take the Odroid cluster home, instead, we keep the clusters in the laboratory classroom and the students can contact us to access the room in non-class hours to work with them. So, the students use the Odroid cluster for a total of 12 hours of laboratory classes, spread over six weeks, plus as many extra hours as they need. During these six weeks, students are expected to administrate their Odroid cluster, dealing with all the potential problems that they may encounter, writing scripts to automatize setup and evaluation processes, and installing libraries and tools (editors, debuggers, etc.) to have a productive programming environment. The rest of the paper explains the laboratory activities related with these two last blocks in PAP.

III. Odroid Cluster Overview and Components

This section provides an overview of the Odroid cluster that is used in the laboratory assignments. Figure 1 illustrates the main components of the Odroid cluster, which consists of one *head node* and four *compute nodes* connected through a switch. The *head node* acts as the gateway for accessing the cluster, it hosts the DHCP server, and it is also in charge of providing Internet connectivity to the whole cluster. The hardware components required to assemble each cluster are listed below:

- *Head node*: personal computer with two network interfaces.
- *Compute nodes*: four Odroid-XU4 boards, each with a eMMC card with a pre-installed Ubuntu 16.04.
- One 8-port Gigabit Ethernet desktop switch.
- Power supply for the Odroid-XU4 boards and switch.

The Odroid-XU4 board, shown in Figure 2, is based on the Samsung Exynos5 Octa chip [7], a low-power heterogeneous multicore processor based on the ARM big.LITTLE architecture [8]. Each processor consists of 4 Cortex-A15 out-of-order cores running at 1.9 GHz and 4 Cortex-A7 in-order cores

Fig. 2: Odroid-XU4 board.

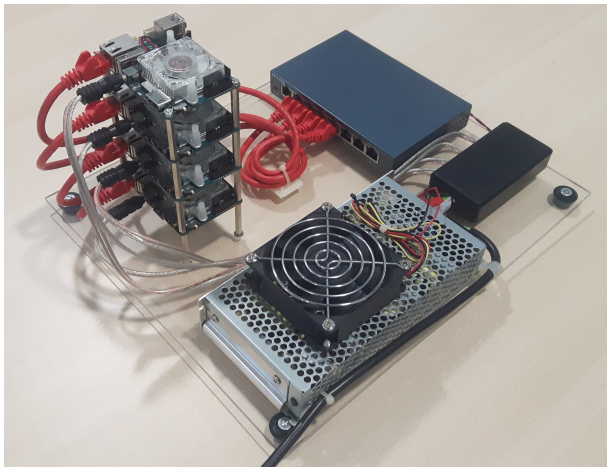


Fig. 3: Picture of the assembled Odroid cluster.

running at 1.3 GHz. Also, the board includes an LPDDR3 RAM chip of 2GB as main memory and ports for Gigabit Ethernet, eMMC 5.0 and μSD Flash storage, USBs and HDMI display. The board also equips an active cooling fan mounted on top of the socket (not shown in Figure 2).

The *head node* and the 4 *compute nodes* are connected to the 8-port Gigabit Ethernet switch. A picture of the assembled cluster is shown in Figure 3. As shown in the picture, we provide a methacrylate plate that has the switch and the power supply already attached, and also some free space for the students to stack the Odroid boards vertically. This way the cluster is much more compact and only a single plug is required for the whole cluster.

The total price of the Odroid cluster is $354. Each Odroid-XU4 board costs $59 without the eMMC card nor the power supply, which have to be purchased separately for $16.50 and $5.50, respectively. The Ethernet switch costs $20 and the 5 Ethernet cables add $2 each.

IV. Odroid Cluster Setup

This section describes the required steps to set the Odroid cluster up. These steps are followed by the students in the laboratory assignment of the second block of the course.

A. Assembling the Odroid Cluster

The students first identify all the components that we provide, which are listed in Section III (with all cables, screws, and separators required). Then they physically assemble the Odroid boards, stacking them vertically using the separators.

B. Head Node Setup

The PC that is used as the head node has two network interfaces and runs a Linux Ubuntu Desktop 16.04. The primary functions of the head node are to share the Internet connection with the compute nodes and to act as the DHCP server.

To set the head node up the PC has to be connected to the Internet using one of the network interfaces and to the Gigabit Ethernet switch using the second network interface. To share the Internet connection with the compute nodes the students use the Linux Network Manager, which allows to easily share the Internet connection using a GUI. To do so, the students need to identify the network interface that is connected to the switch, and select the option "Method: Shared to other computers" in the "IPv4 Settings" tab.

To check that the head node is properly connected to the Internet and to the switch, we ask the students to use and explain the output of the `ifconfig` command. If the connections are properly configured, the output shows that the head node is connected to two networks, one using the network interface that is connected to the Internet and one using the interface that is connected to the switch.

C. Compute Nodes Setup

Each compute node boots its operating system image from the eMMC card and is assigned an IP address by the DHCP server in the head node. The following steps are followed to configure the compute nodes as part of the cluster.

The first step is to install the four eMMC cards in the four Odroid boards and to connect the four boards to the switch using four Ethernet cables. The eMMC cards provided to the students already contain a pre-installed Ubuntu Linux 16.04 operating system.

The second step is to boot the compute nodes and to give them a unique hostname. It is essential to turn on the compute nodes one by one because, since they all run the same operating system image, they all have the same hostname, so they cause hostname conflicts in the network if all of them try to boot at the same time. To boot a compute node and change its hostname the students turn on the board and wait for it to boot. The compute nodes are not connected to any display nor peripheral, so all the interaction with them has to be done via SSH from the head node. The students use the `nmap` command from the head node to check if the compute node has booted and, when it is up, they connect to it via SSH. Once connected to the compute node, changing its hostname can be done by simply editing the file `/etc/hostname` so that it contains the desired name. After changing the hostname the board needs to be rebooted and, after checking that the compute node has booted correctly and is visible in the network (using again the `nmap` command from the head node), the students repeat this step for the rest of compute nodes.

We encourage the students to use a naming convention for the compute nodes that facilitates their identification in the physical rack. For instance, we propose to name the compute nodes as `odroid-0`, `odroid-1`, `odroid-2` and `odroid-3`, being `odroid-0` the board at the bottom and `odroid-3` the board at the top of the rack.

D. Trusted SSH Connections

SSH keys are a well-known way to identify trusted computers in a network. We use SSH keys in the Odroid cluster to allow any compute node to access any other one without requiring any password. To do so, the students have to follow the next steps for each compute node.

The first step is to generate a private authentication key pair (a public and a private key) using the command `ssh-keygen`. The second step is to add the public key to the list of authorized keys for the same compute node. To do so, the students simply need to add the previously generated key pair to the file `.ssh/authorized_keys`. The third step is to transfer the public key generated for the compute node to the rest of compute nodes with the command `ssh-copy-id`.

These previous three steps have to be repeated in all the compute nodes. At the end of the process, each compute node should have the public keys of all the compute nodes, and any compute node should be able to access any other one without entering any password. We ask the students to make sure the whole process worked by trying to access different compute nodes from each other via SSH.

E. NFS Setup

Network File System (NFS) is a distributed file system protocol that allows different nodes to share files over a network. The protocol requires one node to act as an NFS server, while the rest of the nodes act as clients. To set up a shared directory between the compute nodes, the students follow a series of steps.

The first step is to install the NFS packages (`nfs-common`) in all the nodes and to create the directory that is going to be shared across the nodes. The second step is to configure one of the compute nodes to be the NFS server. To do so, the students select one of the compute nodes, they install the NFS server packages (`nfs-kernel-server`), and they export the NFS directory to the rest of nodes by editing the file `/etc/exports`. Once this is done the NFS server has to be restarted with the command `sudo service nfs-kernel-server restart`. The third step is to configure the NFS directory in the rest of the compute nodes. To do so, the students can mount the directory exported by the NFS server node on a local directory using the `mount`

command. However, we encourage the students to automatize this so that it gets mounted at boot time, which can be performed by modifying the `/etc/fstab` file.

Once the NFS server and the NFS clients have been configured, the students check that the NFS works properly. To do so, we ask the students to access all the compute nodes, write a file in the shared directory from that node, and then check that all the files are visible by all the nodes.

F. Software Environment for Parallel Programming

Once all the hardware and the system software is set up, the last step to have a fully operational Odroid cluster is to install the software environment for the two standard parallel programming modes used in HPC: MPI and OpenMP. To understand the behavior and the performance of the parallel programs we also install Extrae [9], an instrumentation library to transparently trace MPI and OpenMP programs, and Paraver [10], a trace visualizer and analyzer that will allow students to understand the execution of parallel applications. Both are freely available at the Barcelona Supercomputing Center tools website [11].

OpenMP is available by default with the GNU compiler (`gcc`), so the students do not need to install it manually. In contrast, the MPI packages are not installed by default, so the students must do it. Among the multiple implementations available for MPI we opt to use MPICH, which only requires to install the `mpich` package in every compute node.

To install Extrae, the students first have to install all the required packages in all the compute nodes, and then they manually configure, compile and install the Extrae library in the NFS shared directory. To use Extrae, we provide the students with scripts that automatically enable the tracing of their parallel programs and post-process the generated trace, so it is ready to be analyzed with Paraver.

Finally, the students download Paraver directly as a precompiled binary for x86_64 architectures, which can be executed straight away in the head node.

V. Odroid Cluster Evaluation

A. Cluster Characteristics

To understand the main characteristics of the Odroid boards, we ask the students to use the commands `lscpu` to obtain general information about the node and `"lscpu -e"` to get per-core information. We also encourage them to inspect the file `/cpu/cpuinfo` to get additional details about the cores. Also, the students are asked to use the `lstopo` command (included in the `hwloc` package) to get a graphical representation of all the components of a compute node.

B. Thread and Process Distribution

To find out how the threads and the processes are distributed across the Odroid cluster, we ask the students to experiment with a "Hello world!" benchmark programmed in hybrid MPI/OpenMP. The benchmark simply prints, for each thread, its process ID, its thread ID and the name of the MPI host where it is executed.

```
odroid-0:8
odroid-1:8
odroid-2:8
odroid-3:8
```

(a) 8 processes per node

```
odroid-0:1
odroid-1:1
odroid-2:1
odroid-3:1
```

(b) 1 process per node

Fig. 4: Machine files for MPI programs.

```
1  void pi(int num_steps) {
2     chunk_size = num_steps / num_procs;
3     start = proc_id * chunk_size;
4     end = (proc_id + 1) * chunk_size;
5     h = 1.0 / (double) num_steps;
6     sum = 0.0;
7
8     #pragma omp parallel for reduction(+: sum)
9     for (i = start; i < end; ++i) {
10       x = h * ((double)i - 0.5);
11       sum += 4.0 / (1.0 + x*x);
12    }
13
14    local_pi = h * sum;
15    MPI_Reduce(&local_pi, &global_pi, 1,
16    MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
17 }
```

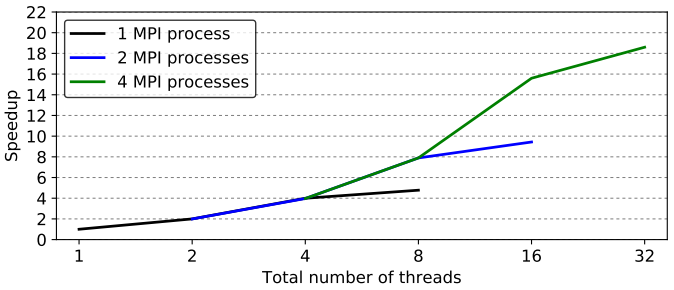
Listing 1: Pi source code.

First, the students experience with the MPI process distribution. To do so, they disable the OpenMP parallelization at compile time and execute the hello world benchmark with the command `mpirun.mpich -np P -machinefile machines ./hello_world`. Note that the `-np` option specifies the number of MPI processes (*P*), and the `-machinefile` option specifies the file that describes the names of the nodes to be used in the execution and the number of processes per node. We provide two machine files, shown in Figure 4, one that specifies eight processes per node and one that specifies one process per node. The students execute with 1 to 32 processes using both machine files and report how the processes are distributed across the nodes.

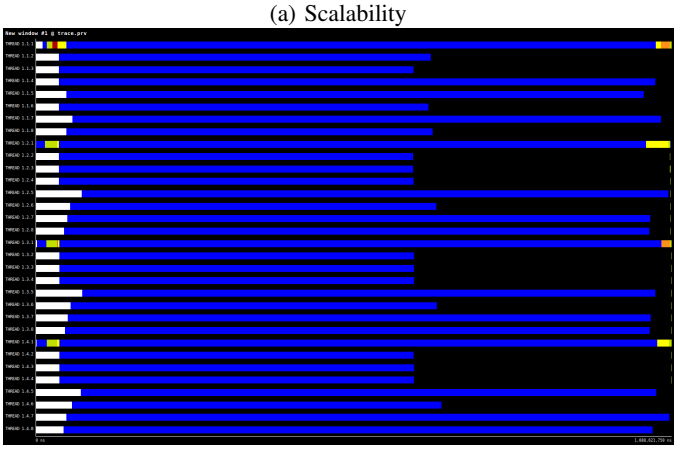
Then the students experience with the distribution of MPI processes and OpenMP threads. To do so, they compile the benchmark with support for OpenMP and execute it with the command `mpirun.mpich -np P -machinefile machines -genv OMP_NUM_THREADS T ./hello_world`. In addition to the number of MPI processes and the machine file, the `-genv` option specifies the number of OpenMP threads (*T*) that each MPI process spawns. The students test both machine files in executions with 1 to 4 processes and 1 to 8 threads and observe how these are distributed in the cluster.

C. Heterogeneity

One of the main characteristics of the Odroid boards is the heterogeneity of the cores, as explained in Section III. We provide the students with a very simple compute-intensive kernel that calculates the number Pi, which allows to clearly understand and experience the heterogeneity of the ARM big.LITTLE architecture.

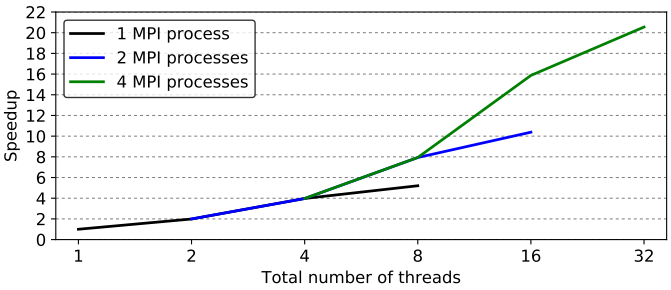


(a) Scalability

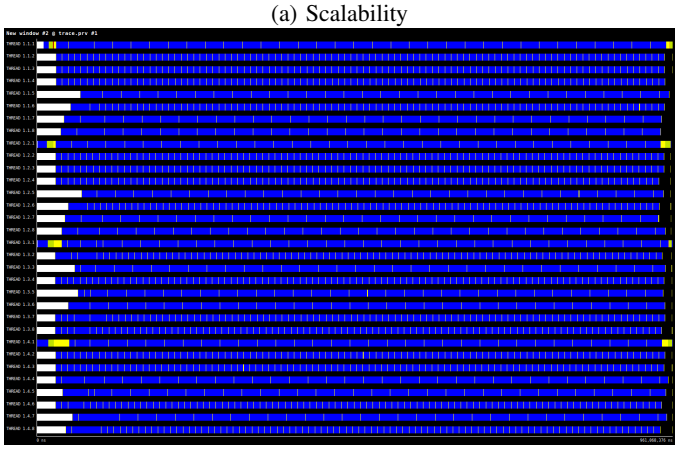


(b) Execution trace

Fig. 5: Scalability and execution timeline of the Pi benchmark with a static OpenMP schedule.



(a) Scalability



(b) Execution trace

Fig. 6: Scalability and execution trace of the Pi benchmark with a dynamic OpenMP scheduler and chunk size 512.

Listing 1 shows the code of the Pi benchmark. The first part (lines 2 to 6) initializes the variables and calculates the start and end of the iteration space (for the loop in line 9) assigned to each MPI process. The second part (lines 8 to 12) is the main kernel, which is a compute-intensive loop that can be parallelized using the OpenMP `parallel for` construct with a reduction. The last part (lines 14 to 16) calculates the partial results (`local_pi`) in each MPI process and reduces them in a global variable (`global_pi`) to obtain the final result.

We ask the students to compile and execute the Pi benchmark with 1.28G steps and a different number of MPI processes (1 to 4) and OpenMP threads (1 to 8). Note that the OpenMP `parallel for` construct in the source code does not specify any schedule for the iterations of the loop, so the static one is used by default (one contiguous block of $(end - start) \div T$ iterations per thread, being T the number of OpenMP threads per MPI process. Figure 5a shows the scalability of the benchmark with the static scheduler. The figure shows the speedup achieved by augmenting the total number of threads. The total number of threads are derived from the executions with 1 to 4 MPI processes and 1 to 8 OpenMP threads per process. The MPI processes are distributed across the nodes. The results show perfect scalability when up to 4 threads per process are used. This happens because the benchmark is computationally intensive and the threads are scheduled on the fast cores by default. The parallel efficiency decreases when eight threads per process are used, achieving speedups of $4.85\times$, $9.42\times$ and $18.32\times$ with 1, 2 and 4 MPI processes, respectively. The heterogeneity of the ARM big.LITTLE architecture causes these performance degradations. As shown in the execution timeline (obtained with *Extrae* and visualized with *Paraver*) in Figure 5b, the execution phases (in blue) of the fast cores end significantly earlier than the ones of the slow cores, so then the fast cores have to wait (in black) for the slow cores to finish. This execution imbalance (not work imbalance) is due to the different computing power of the two kinds of cores in the processor, preventing the Pi benchmark from scaling further in executions with more than four threads per process.

To mitigate the effects of heterogeneity we ask the students to try to improve the performance of the Pi benchmark by using a dynamic schedule in OpenMP with different chunk sizes (number of iterations per chunk dynamically assigned to a thread). Figure 6a shows the scalability of the Pi benchmark with a dynamic scheduler and a chunk size of 512, which is the best granularity for this experiment. As in the case of the static scheduler, the results show perfect scalability with up to four threads per process. When eight threads per process are used, the scalability is slightly better than the one obtained with the static scheduler, achieving respective speedups of $5.26\times$, $10.34\times$ and $20.50\times$ with 1, 2 and 4 MPI processes. The execution trace in Figure 6b clearly shows how the dynamic scheduler perfectly distributes the work among the threads, so

the time the threads spend waiting in the barrier is negligible. However, achieving perfect scalability is impossible because of two important aspects of the architecture. One is the reduced compute capabilities of the slow cores compared to the fast ones, and the second one is the DVFS controller that, in order not to exceed the power and temperature caps, lowers the frequency and the voltage of the cores when they are all active at the same time.

VI. Parallelization of the Heat Diffusion Program

In the last assignment of the course, the students have to parallelize a sequential code that simulates heat diffusion in a solid body using the Jacobi solver for the heat equation. The program is executed with a configuration file that specifies the maximum number of simulation steps, the size of the bi-dimensional solid body and the number of heat sources, with their position, size, and temperature. Two configuration files are provided, one for programming and debugging and one for evaluating the parallelization. The program reports some performance measurements (execution time, floating point operations, the residual and the number of simulations steps performed) and an image file with a gradient from red (hot) to dark blue (cold) for the final solution. Figure 7 shows the output of the program for a solid with two heat sources, one in the upper left corner and one at the bottom center.

A skeleton of the code for the heat diffusion benchmark is shown in Listing 2. Note that the code contains comments to guide the parallelization strategy that is explained in the two next subsections. In the initialization (lines 2, 3 and 4), the program reads the input configuration file to establish the input parameters, including the size of the matrices (N), the threshold value for convergence (R), and the maximum number of iterations. Then the program allocates the memory for two matrices (`u` and `utmp`) and initializes them according to the heat sources. Then the code enters the main loop (lines 9 to 39), that has three main parts. The first part (lines 14 to 26) computes one step of the heat diffusion simulation. The

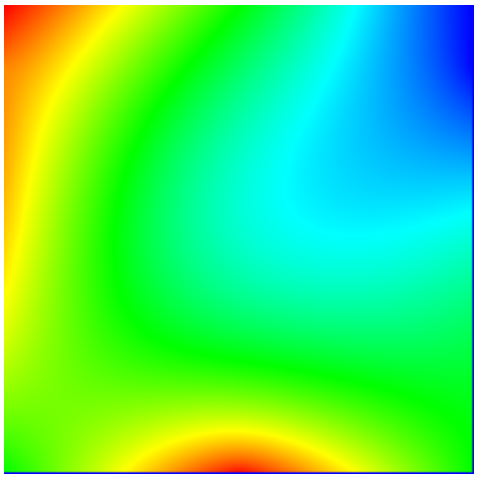


Fig. 7: Image of the temperature of the solid body.

```
1  void heat() {
2     Read configuration file
3     Allocate memory
4     Initialize matrices and heat sources
5
6     // MPI-1: Broadcast input parameters
7     // MPI-2: Distribute matrix
8
9     while(1) {
10
11       // MPI-3: Exchange halos
12
13       // OPENMP-1: Parallelize loop
14       for(i=1; i<N-1; i++) {
15          for(j=1; j<N-1; j++) {
16             utmp[i][j] = 0.20 * (
17                          u[i][j]   + // center
18                          u[i][j-1] + // left
19                          u[i][j+1] + // right
20                          u[i-1][j] + // top
21                          u[i+1][j]); // bottom
22
23             diff = utmp[i][j] - u[i][j];
24             residual += diff * diff;
25          }
26       }
27
28       // MPI-3: Exchange halos
29
30       aux = u;
31       u = utmp;
32       utmp = aux;
33
34       // MPI-4: Communicate residual
35
36       iter++;
37       if(residual < R && iter == MAX_ITERS)
38          break;
39    }
40
41    // MPI-2: Distribute matrix
42 }
```

Listing 2: Heat diffusion source code.

computation is a 5-point 2D stencil that uses two matrices, one as input and one as output. The second part (lines 30 to 32) swaps the matrices so that the output of the previous stencil becomes the input of the next stencil in the following iteration. The third part (lines 36 to 38) checks if the solution has converged or the maximum amount of iterations has been reached.

A. MPI Parallelization Strategy

The heat diffusion code we provide to the students with contains a commented skeleton for the MPI and the OpenMP parallelization, as shown in Listing 2. The students first parallelize the application with MPI following a series of steps.

The first step, labeled as `MPI-1` in line 6, consists of exchanging the information regarding the parameters of the execution between the processes. On the one hand, the master first reads the configuration file, allocates memory and sends the execution parameters to the workers; then it computes the heat equation on the whole 2D space, and finally, it reports the performance metrics and generates the output file. On the other hand, each worker receives from the master the information required to solve the heat equation, allocates

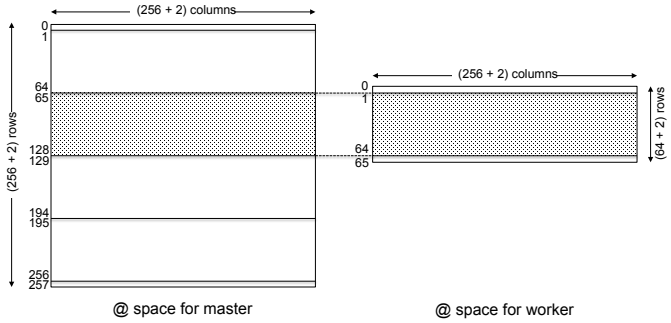


Fig. 8: Global and local storage for the matrix of the heat diffusion algorithm with different processes.

memory, performs the computation on the whole 2D space and finishes. Note that this version does not benefit from the parallel execution since workers replicate the work done by the master.

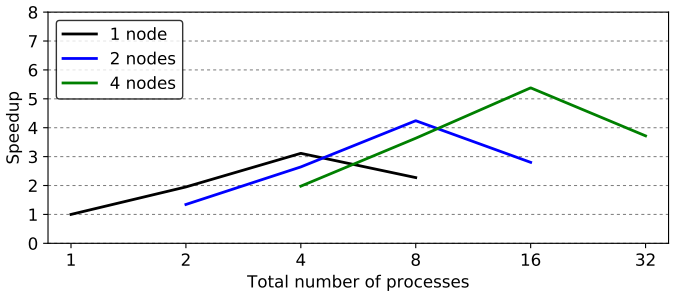
The second step, labeled as `MPI-2` in lines 7 and 41, consists of distributing the matrices among the processes so that the master and the workers solve the equation for a subset of consecutive rows. Figure 8 shows an example for a 256x256 matrix (plus the halos) distributed among 4 processes. After computing its part of the matrix, the workers return the part of the matrix they have computed to the master to reconstruct the complete 2D data space. This version of the code does not generate a correct result because the processes do not communicate the boundaries to the neighbor processes at each simulation step.

The third step, labeled as `MPI-3` in line 11 and 28, consists of adding the necessary communication so that, at each simulation step, the boundaries are exchanged between consecutive processors. The students have the freedom to use the MPI communication routines they find more appropriate. This version of the code still generates an incorrect solution because the total number of simulation steps done in each process is controlled by the local residual of each process and the maximum number of simulation steps specified in the configuration file, instead of computing the residual and checking the convergence globally.

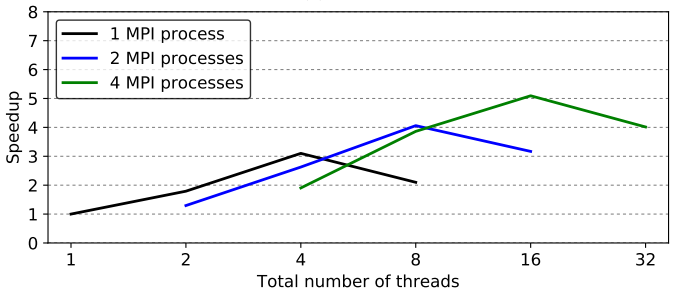
The fourth and last step, labeled as `MPI-4` in line 34, consists on adding the necessary communication to control the number of simulation steps of all the processes by computing a global residual value at every simulation step. The students again can use the communication strategy that they find most appropriate. This version of the code generates the same result as the one generated by the sequential code.

B. Hybrid MPI/OpenMP Parallelization Strategy

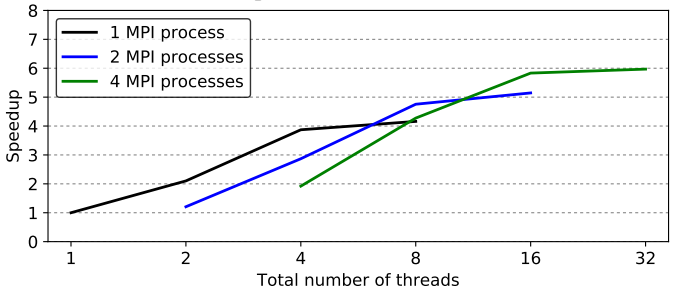
Once the students have successfully parallelized the heat diffusion algorithm with MPI, they proceed with the hybrid MPI/OpenMP implementation. Adding the OpenMP parallelization on top of the MPI implementation is quite trivial, as it only requires introducing an OpenMP `parallel for reduction(+:residual)` construct in the stencil kernel (labeled as `OPENMP-1` in line 13). We also encourage the



(a) MPI



(b) MPI+OpenMP with static scheduler



(c) MPI+OpenMP with dynamic scheduler

Fig. 9: Speedup of the different parallel implementations of the heat diffusion algorithm.

students to try different OpenMP schedulers to mitigate the effects of the heterogeneity, as they have learned in the previous laboratory assignment.

C. Evaluation on the Odroid Cluster

After successfully parallelizing the heat diffusion program the students proceed with its performance evaluation on the Odroid cluster.

Figure 9a shows the scalability of the MPI parallel implementation of the heat diffusion benchmark. The figure shows the speedup achieved by augmenting the total number of processes, which are derived from executions with 1 to 8 MPI processes per node and 1 to 4 nodes. It can be observed that, when up to 4 processes are used in each node, the program scales up to $3.06\times$, $4.17\times$, and $5.41\times$ with 1, 2, and 4 nodes, respectively. In addition, the performance drops with eight processes per node, reducing the speedups to $2.16\times$, $2.89\times$, and $3.81\times$ with 1, 2, and 4 nodes, respectively. This exact trend is repeated in the hybrid MPI/OpenMP implementation with a static scheduler, as shown in Figure 9b. As in the case of the Pi benchmark, the performance degradations

when using eight threads or processes per node are caused by the heterogeneity of the cores and the inability of the parallel implementation to dynamically balance the load in such scenario. Figures 9a and 9b also reflect the higher cost of communicating data across the nodes rather than inside the nodes. It can be observed that, in the MPI version executing with four total processes, using one node for all the processes is 54% faster than spreading them across four nodes.

The scalability of the hybrid MPI/OpenMP implementation with a dynamic scheduler is shown in Figure 9c. It can be observed that, when using eight threads per node, the performance does not drop as in the previous two versions of the code. However, the performance gains are very low compared to the executions with four threads per node and the same number of nodes. Another important difference is that the scalability obtained with up to four threads per node is slightly higher than in the other two implementations. With four threads per node the results show that the dynamic scheduler is 18%, 10% and 8% faster than the static one in executions with 1, 2, and 4 nodes, respectively.

VII. Previous and Related Work

The widespread availability and low cost of single-board computers have provided educators the possibility of building small-scale clusters and exploring different ways to use them in their courses related to parallel and distributed computing. Several small-scale clusters based on various Raspberry Pi and Odroid models, Nvidia's Jetson TK1 or Adapteva's Parallella are presented in [12]. The authors of that joint publication also present the various strategies they follow regarding the use of the clusters in class and summarise some earlier examples of small-scale clusters that have been inspirational for the recent proposals based on inexpensive single-board computers.

David Toth introduced in 2014 the first version of the Half Shoebox Cluster (HSC) [13], which equipped two compute nodes based on ARM Cortex-A7 dual-core processors, with the purpose of teaching Pthreads, OpenMP and MPI. The HSC has been continuously evolving, and six subsequent versions have been built with various Odroid SoCs such as the U3, the C1, the XU3-Lite, the C1+, the XU4, and the C2. The XU3-Lite and the XU4 have 8-core ARM CPUs, while the rest of Odroid boards have 4-core CPUs. Rosie [14], constructed in 2014 by Libby Shoop, consisted of 6 NVIDIA TK1 single-board computers, each equipped with a quad-core Cortex-A15 processor and an integrated Kepler GPU with 192 CUDA cores. The cluster has been used to teach heterogeneous computing techniques with OpenMP, MPI, and CUDA. In 2016, Suzanne Matthews designed the StudentParallella [15], which is composed of 4 Parallella nodes, each with a Zynq 7000-series dual-core ARM Cortex-A9 and an Epiphany coprocessor with 16 cores. The StudentParallela was intended to teach the native Epiphany programming model, Pthreads, OpenMP and MPI in a parallel computing elective course. Cu-T-Pi [16], created by James Wolfer, is composed of 1 Nvidia Jetson TK1 head node and 4 Model B+ Raspberry Pi worker nodes with 2 ARM cores each. The Cu-T-Pi was used to teach parallel programming and also to demonstrate benchmarking concepts.

Most of the previously mentioned small-scale clusters were designed as an alternative to non-dedicated networks of workstations in laboratory classrooms, departmental/university HPC clusters, and cloud services such as Amazon's EC2 or the XSEDE [17] and the Blue Waters programs [18]. However, their main motivation was to overcome the practical and economic limitations of using such systems, and their small-scale clusters were only intended to be used as an evaluation platform for parallel programming assignments. For these reasons, in many courses the small-scale clusters were given to the students pre-assembled, configured, and ready to be used to learn parallel programming using Pthreads, OpenMP, MPI and/or CUDA.

Based on our experience at the Barcelona Supercomputing Center (BSC-CNS), where we host the Marenostrum cluster-in-a-chapel supercomputer, we have observed that well-trained HPC system administrators are offered good positions to set up and administrate data centers in companies and research institutions. For this reason, we decided to design a module, practical in nature, to teach these skills. In contrast to the previously commented initiatives, in our course we incorporate a laboratory assignment specifically intended to teach the cluster setup process, the HPC software ecosystem configuration and the initial performance testing of the cluster, trying to mimic as much as possible the set up of real systems found in supercomputing centers. This laboratory assignment is self-contained and evaluated separately from the parallel programming assignments, and complements the theory classes on HPC systems as well as the assignment where the students design a real cluster that uses commodity components and has different power/performance trade-offs and economic budgets.

Our first Odroid-XU3 cluster was build during the spring semester in 2015 by a couple of undergraduate students doing an optional laboratory assignment. The initiative was followed by the current Odroid-XU4 cluster that has been used since then. Using the Odroid cluster cluster to learn and practice MPI programming (the second assignment) was a proposal coming from our students after building, configuring and testing the first XU4 Odroid cluster during the 2016 spring semester.

VIII. Conclusions and Evolution

The continuous expansion of HPC systems and data centers has come together with an increasing demand for expert HPC system designers, administrators and programmers. To fulfil this demand, most university degrees have introduced courses on parallel programming and HPC systems in recent years. However, very often the laboratory assignments of these courses only focus on the parallel programming part, and the students never experiment with the design, set up and administration of HPC systems.

This paper presents a methodology and framework to use small-scale clusters of single-board computers to teach HPC

systems and parallel programming. In contrast to the traditional paths of teaching parallel programming with remote clusters managed by the university, using small-scale clusters allows the students to experience with assembling a cluster, setting it up, configuring all the software ecosystem, and administrating it during the duration of the course. In this paper, we show that modern single-board computers have very appealing characteristics for being used in laboratory classes, given their low cost, their ability to execute the same software stack as HPC systems, and the similarity between their processors and the ones used in HPC. Moreover, we show that these small-scale clusters are also a valuable platform to experience with relevant aspects of today HPC systems such as heterogeneity, dynamic voltage frequency scaling with different number of active cores, or cost of data communication.

A very positive adoption of the two assignments in the Parallel Architectures and Programming (PAP) course has been observed. Between 15 and 20 students yearly follow the course and build five independent clusters working in groups of three or four students. Although the number of students per group may seem large, the curiosity and surprises found while doing the assignments motivates interesting discussions among them.

In the future we plan to include some additional learning outcomes from the experimental evaluation of the Odroid cluster, allowing students to perform a more insightful comparative study with other real full-scale HPC systems. In particular: *i)* to measure the power drain of the single-board computers and obtain power profiles of benchmarks and applications as well as global energy-to-solution figures integrated in the Paraver performance analyzer as described in [19]; *ii)* to obtain peak memory (STREAM) and network (ping-pong) bandwidth, as well as compute (LINPACK) performance in order to fit them in the theoretical roofline model for the Odroid board; and *iii)* to complement the knowledge of the systems software needed to operate a cluster used by a large community of users, teaching how to install and configure a queueing system (SLURM) and an environment module implementation (lmod/modules).

ACKNOWLEDGMENT

This work is partially supported by the Spanish Government through Programa Severo Ochoa (SEV-2015-0493), by the Spanish Ministry of Science and Technology (contracts TIN2015-65316-P and FJCI-2016-30985), by the Generalitat de Catalunya (contract 2017-SGR-1414), and by the European Union's Horizon 2020 research and innovation programme (grant agreements 671697 and 779877).

REFERENCES

[1] https://wiki.odroid.com/odroid-xu4/odroid-xu4.
[2] *MPI: A Message-passing Interface Standard. Version 3.1*, Jun. 2015.
[3] *OpenMP Application Program Interface. Version 4.5*, Nov. 2015.
[4] E. Ayguade and D. Jimenez-Gonzalez, "An approach to task-based parallel programming for undergraduate students," *Journal on Parallel and Distributed Computing*, vol. 118, no. P1, pp. 140–156, Aug. 2018.
[5] E. Ayguade, L. Alvarez, and F. Banchelli, "OpenMP: what's inside the black box?" *Peachy Parallel Assignments (EduHPC 2018)*, 2018.
[6] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009.
[7] https://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-5-octa-5420.
[8] P. Greenhalgh, "big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7," *ARM White paper*, 2011.
[9] Barcelona Supercomputing Center, *Extrae User Guide Manual. Version 2.2.0*, 2011.
[10] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A Tool to Visualize and Analyze Parallel Code," in *Proceedings of WoTUG-18: transputer and occam developments*, vol. 44, no. 1, 1995, pp. 17–31.
[11] https://tools.bsc.es/downloads.
[12] S. Holt, A. Meaux, J. Roth, and D. Toth, "Using inexpensive microclusters and accessible materials for cost-effective parallel and distributed computing education," *Journal of Computational Science Education*, vol. 8, no. 3, pp. 2–10, Dec. 2017.
[13] D. Toth, "A portable cluster for each student," in *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, 2014, pp. 1130–1134.
[14] J. C. Adams, J. Caswell, S. J. Matthews, C. Peck, E. Shoop, D. Toth, and J. Wolfer, "The micro-cluster showcase: 7 inexpensive beowulf clusters for teaching pdc," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, 2016, pp. 82–83.
[15] S. J. Matthews, "Teaching with parallella: A first look in an undergraduate parallel computing course," *Journal of Computer Sciences in Colleges*, vol. 31, no. 3, pp. 18–27, Jan. 2016.
[16] J. Wolfer, "A heterogeneous supercomputer model for high-performance parallel computing pedagogy," in *Proceedings of the IEEE Global Engineering Education Conference*, 2015, pp. 785–791.
[17] XSEDE. https://www.xsede.org/.
[18] Blue Waters. http://www.ncsa.illinois.edu/enabling/bluewaters.
[19] F. Mantovani and E. Calore, "Performance and power analysis of hpc workloads on heterogeneous multi-node clusters," *Journal of Low Power Electronics and Applications*, vol. 8, no. 2, May 2018.