Dynamically Distributing Tasks from an Unattended Parallel Compiler with Cloudbook

José J. García-Aranda[1], Juan Ramos-Díaz[1], Sergio Molina-Cardín[1], Xavier Larriva-Novo[2], Andrés Bustos[3], Luis A. Galindo[4], and Rafael Mayo-García[3](✉)

[1] Nokia, María Tubau 9, 28050 Madrid, Spain
[2] ETSIT-UPM, Avda. Complutense s/n, 28040 Madrid, Spain
[3] CIEMAT, Avda. Complutense 40, 28040 Madrid, Spain
rafael.mayo@ciemat.es
[4] Telefónica, Ronda de la Comunicación 2, 28050 Madrid, Spain

Abstract. A dynamic version of Cloudbook is presented in this work, a new tool for automatically and unattendedly parallelizing codes which also lately distributes the tasks dynamically. Cloudbook is designed for Python codes and, above all, makes the parallelization in a way in which the number and main characteristics of the available infrastructure is taken into account for optimizing the execution (performance, bandwidth connection, etc.) in a dynamic way. Cloudbook is designed to allow developers to get the technical benefits of automated distribution and parallelization of programs with a very low learning cost. It only requires labelling the original code with a reduced set of pragmas located at function headers. Results of the tests carried out with Cloudbook with several codes on a real infrastructure are presented as well.

Keywords: Parallel computing · Compiler · Automatization

1 Introduction

In general terms, parallel computing refers to the use in combination of two or more processes (threads, cores, computers…) to solve a single problem. This methodology is carried out by using computing architectures in which several processors execute or process simultaneously an application or computation. Thus, it is possible to perform large computations by dividing the workload between more than one processor, all of which execute their task through the computation at the same time in a predefined scheme.

Compared to serial computing, parallel computing is then much better suited for modelling, simulating, and understanding complex real world phenomena. Thus, the primary objective of parallel computing, also known as parallel processing, is to increase the available computation power for faster application execution or task resolution.

Today's supercomputers employ parallel computing principles to operate and solve complex problems. Some examples of computational science applied to natural sciences are galaxy formation, climate change, weather forecast, energy production, bioinformatics, material science, etc. Although parallel computing was firstly used for scientific computing and the simulation of scientific problems, nowadays it is present in any field, including human and social sciences too. This has led and is still leading to the design of more powerful and efficient parallel hardware and software making a reality the so-called High Performance Computing (HPC).

As a consequence, the use of the HPC infrastructure has become a challenge itself with the advent of many-core systems, i.e. the parallelization of serial programs has become a mainstream and cornerstone programming task.

Additionally, parallel computers based on interconnected networks need to have some kind of routing protocols to enable the transmission of messages between nodes that are not directly connected. The medium used for communication between the processors is likely to be hierarchical in large multiprocessor machines and have a strong influence on the cluster performance when large parallel calculi are executed. The variation of the performance with the number of computer resources is known as scalability and depends on the hardware and architecture of the physical resources as well as on the computational characteristics of the problem to be solved. Scalability must be maximized, but it is usually degraded when the computing resources or the problem size rise.

Summarizing, parallel computing is highly useful, but presents several challenges that become more and more complex to be overcome as the size of the infrastructures increase. Companies must manufacture efficient supercomputers from the energy point of view that, in addition, should be efficiently exploited from the usage point of view. In this sense, the way in which the parallelization is implemented is key as programming to target parallel architectures can be highly difficult and requires human expertise and know-how.

The first aim of this work is to provide a tool that automatically and unattendedly parallelizes a code, releasing the final user of designing such a parallel implantation as well as of debugging processes. Lately, a second goal is achieved by providing dynamic capabilities for distributing the parallel tasks by the tool itself in order to optimize the computational efficiency in terms of performance.

2 Related Work

As aforementioned, multi- and many-core machines are very common nowadays, allowing a number of problems exploiting the tremendous processing power of such machines. Such goal can only be efficiently achieved by parallel compilation. Automatic conversion of serial code into its functionally equivalent parallel version remains as an open challenge for researchers for the last years. These tools are intended to transform legacy serial code into parallel code to execute on parallel architectures.

With respect to parallel compilers, a couple of reviews of the different tools can be found in [1] and [2]. Roughly speaking, these works compare different automatic tools (see references therein) on the basis of technology, language, available platforms and features, and drawbacks. The most important phase within the flowchart for parallelizing the code is the detection of potential blocks, which is also the most time consuming part.

It is also found that most of the tools are either oriented to FORTRAN or C/C++ as they clearly describe the operational flow in the code. Then, it is not strange that even Barve *et al.* developed a serial to parallel C++ code converter for multi-core machines after the publication of their revision [3]. Another posterior work presented a novel architecture based on web services which is able to translate any legacy software application into a parallel code [4]. In a similar way, André *et al.* [5] present an environment for programming distributed memory computers using High Performance Fortran, with emphasis put on compilation techniques and distributed array management. OpenMP should be also highlighted, the well-known application programming interface for shared memory parallel computing [6].

With a focus on data-parallel compiler, the aim has been to equal the performance of carefully hand-optimized parallel codes. For tightly coupled applications based on line sweeps, the Rice dHPF compiler [7] and its extension [8] can be cited. Most closed to Data-analytics, the TOREADOR tool has been recently published [9].

Specific developments for GPU environments such as the thesis by Hsu [10] or for executions carried out by virtual machines with the HPVM framework [11] can be consulted, but those works are less related to the one presented here.

On the other hand, literature about optimizing the execution of codes along runtime taking into account the underlying infrastructure is huge. Just to focus on heterogeneous architectures, the OmpSs framework is able to provide dynamic allocation of jobs among other duties [12], but other solutions for heterogeneous resources are available too [13–15]. For loosely coupled applications, such as Monte Carlos codes, the Montera framework provided good results on real in production distributed heterogeneous platforms [16]. Works like this opened the door to widen this kind of solutions to virtualized environments [17].

Analyzing the existing solutions, it can be deducted that all of them are still far away from their expectations, focused on a specific kind of application/environment, or do not stack a parallel compiler to the available infrastructure along runtime. The aim of this work is then to present a general-purpose tool that will both make codes parallel and will also take into account the infrastructure on which those codes are executed in order to maximize their performance.

3 The Unattended Parallel Compiler

There are two main problems in parallel execution: the generation of pieces of code to be executed on each processor as well as the efficient deployment and coordinated execution of these tasks. This work focuses on both problems by splitting a Python source code into the so-called deployable units (DU) and lately distributing these DUs in a coordinated way allowing communication among them if needed. This solution is called Cloudbook.

In order to achieve an efficient parallel execution, the proposed solution Cloudbook defines several pragmas to be integrated in the source code, which will be interpreted by a “maker” designed to split the code into DUs. The main components of the proposed solution are summarized in the following architecture:

– Maker: comprises the graph analyzer of source code and the splitter, which produces the DUs
– Deployer: assigns the DUs to the available resources and launches the execution
– Agents: execute the DUs.

During execution of Cloudbook programs, there is not need for a central server which attends the requests from agents asking for tasks o providing results, because Cloubook allows agents communicate each other and therefore the figure of a central controller server is not needed.

3.1 Requirements

In order to both optimize parallelism and improve performance, the programmer can include a series of labels in the functions that would indicate the agents how to execute those functions.

Certain Cloudbook Pragmas may reflect the fork-join model spirit [18]. However, in Cloudbook the invokers do not match the concept of "parent" of the fork-join model because (among other details) tasks are executed on different agents, do not share a copy of parent's variables, threads can be either created at invoker or invoked, and parallel functions cannot return values.

Cloudbook supports the following language extensions (pragmas) for functions:

– #__CLOUDBOOK:NONBLOCKING__: functions with this label cannot return anything. When Cloudbook detects a non-blocking function, its code is modified to launch a thread at the invoked agent and returns immediately. These functions cannot return any value. Restriction: function parameters cannot be objects, only basic types.
– #__CLOUDBOOK:PARALLEL__: these functions are deployed in all DUs. These functions are non-blocking by construction and therefore are not allowed to return anything. The difference between non-blocking and parallel consists of the number of DUs in which the function is deployed. Non-blocking functions are deployed in only one DU, whereas parallel functions are deployed in all available DUs. Parallel functions are synchronizable by using #CLOUDBOOK:SYNC__ (see below). Restriction: function parameters cannot be objects, but basic types.
– #__CLOUDBOOK:RECURSIVE__: these functions are deployed in all DUs. The behavior is defined to maximize the level of recursivity. Each recursive invocation from any DU invokes other DU, which means that in a circle with 10 machines you have 10 times more recursive level than in one machine. Restriction: function parameters cannot be objects, only basic types.
– #__CLOUDBOOK:LOCAL__: these functions are deployed in all DUs, in order to be available for local invocations, avoiding communications. This pragma is intended to be considered at "tuning" phase of the program. There is no restriction in the parameters. They can be objects as well as basic types.
– #__CLOUDBOOK:DU0__: these functions are deployed in DU0. This pragma is useful if your program has certain interactive functionality such as GUIs or keyboard input, which can be forced to be executed in Agent 0.

The pragmas at the level of function invocation are:

- #__CLOUDBOOK:NONBLOCKING_INV__: if the function is not defined as NONBLOCKING but the programmer does not want to wait for its execution, can invoke the function using this label. In this case, a thread is launched at invoker agent, whereas when the label is used at function definition, the thread is created at the invoked agent.
- #__CLOUDBOOK:SYNC[:timeout]__: this will wait until all the non-blocking operations have finished. In order to be able to continue executing in the cases where an agent stops working, the optional parameter timeout (specified in seconds) may be set after the SYNC word and a colon (:). In the case the optional parameter is set, the program will continue running whenever the all non-blocking operations have finished or when the waiting time exceeds the timeout value (whatever happens first). Example: #__CLOUDBOOK:SYNC:3__.

Cloudbook supports global variables, but special treatment is needed:

- global: this Python keyword indicates to Cloudbook that must either load or refresh the value of global var. Since then, a local cache copy of the var is used. The use of a local copy benefits the performance, reducing communications. In this case, "global" is not a Cloudbook pragma, but a Python keyword
- Critical sections: in order to support "safe variables" (which only can be used by one DU at the same time) or any other critical resource, Cloudbook supports the definition of critical sections, which can be defined by the pragmas #__CLOUDBOK:LOCK__ and #__CLOUDBOK:UNLOCK__; this way the modifications of global variables or critical data are only accessed by one agent at a time
- #__CLOUDBOOK:NONSHARED__: the variable is created at any agent but non shared among different agents. This type of variables allows having unique identifiers for each agent, and different data at each agent if it is needed
- #__CLOUDBOOK:CONST__: this pragma allows Cloudbook to manage constant global variables in an efficient way (replicate them among all DUs).

The use of global variables implies the creation of the following strategy:

- Each global variable is translated into one non-idempotent management function. It exists only in a unique DU
- The management function includes the global var as a non-volatile internal attribute Additionally, this management function must be a critical section in order to allow multiple access from DU outside
- Each function using the global var requests its fresh value at the beginning, invoking the management function, and stores it into volatile internal variable, which is used during the function execution time
- If inside the body of a function that use the global variable is required a refresh of its value, it can be possible invoking another local function that get at the beginning a fresh global variable value and returns its value.

In order to be refreshed by Cloudbook conveniently, global variables should be defined explicitly, but there is no need for a specific pragma. On the other side, objects work as a function abstraction, i.e., the maker analyzes the procedural part of the program and generates the different DUs.

Last but not least, the generic configuration parameters for Cloudbook are:

– Circle ID, unique identifier of a circle, being a circle a set of available resources
– Circle definition, which includes features of each machine belonging to the circle
– Distributed file system to be used by all agents, which is part of the circle properties
– Desired deployable units, number of DUs, which normally is greater or equal to the number of machines
– Cloudbook_maxthreads, which allows launching up to CLOUDBOOK_MAXTHREADS functions in parallel and waits to launch the next one until any of the previously launched functions ends. This limit allows keeping under control the number of resources at any invocation of parallel functions.

4 Cloudbook Global Architecture

The Cloudbook global architecture for a dynamic behavior is much simple and is composed of the following components (see Fig. 1 too):

– Agent: This is the component that will be in each machine that is part of the Cloudbook circle. Tasks:
  - Executing code and communicating with other agents
  - Start the application (through invocation to "run" at deployer service)
– Maker: This component receives a link to the code (which is located in the distributed FS). The maker performs two tasks:
  - Graph analysis: parses the code and produces the invocations matrix
  - Split the program: groups functions into code pieces, which are the "Deployable Units" (DU). The number of DU depends on circle definition (number of agents and machines) and possible certain additional criteria.
– Distributed file system: This module stores code and data. It is accessible by all agents; the original code is located on folder the "original" and the maker saves the DUs on the "cloudbook" folder. Agents are agnostic to this component. All machines mount the distributed file system as a local directory and use it in the same way as local
– Deployer service: This module is responsible for the creation of the cloudbook directory, which contains the assignment of the deployable units to the different agents, and starts the execution. Tasks:
  - Create the cloudbook directory
  - When "run" command is invoked, checks if all the required agents are online and then start the execution

– Stats monitor: this module contains the statistics associated to the DUs' executions in order to allow a dynamic behavior of the tool

Cloudbook relies on distributed file systems to make DUs accessible to all agents and also as storage for program files, which must be accessed by all agents. Cloudbook is then agnostic to the file system and the programmer must decide which file system to use in order to get a scalable communication mechanism avoiding using centralized servers (for small/medium projects a NFS server may be enough, for big/huge projects a bit torrent FS may be needed).

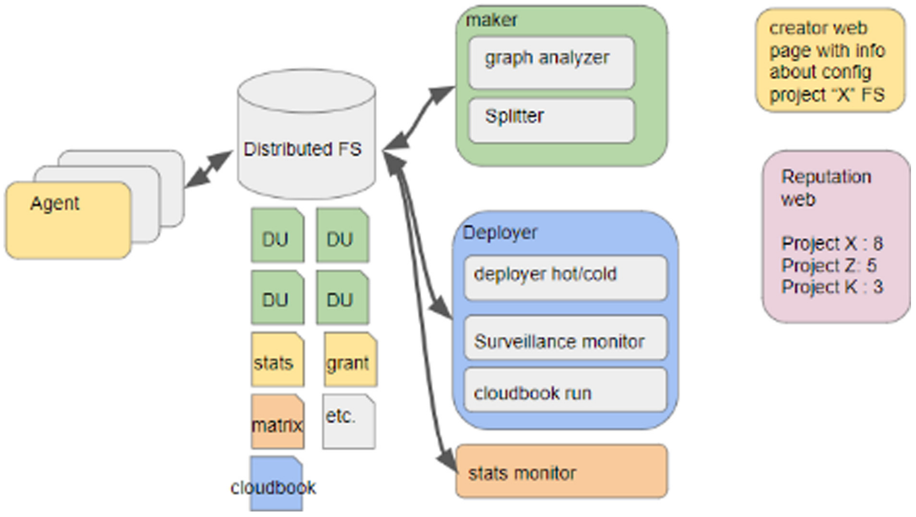

Fig. 1. The Cloudbook architecture

In order to replace these centralized servers, certain files have been defined for allowing communication of all platform components:

– agent_<XXX>_grant.json: written by agent, read by deployer. Includes information of agent identification, power granted by the agent, and public/private IP addresses. There is one file per agent and the deployer reads and deletes them periodically. The agents must re-create the file periodically and the deployer may deduce which agents are new and which agents have stopped based on comparison of existing files
– Alarm files: written by agents, read by the deployer. There are two types of alarm: WARNING (if it is possible to continue executing) and CRITICAL (if not possible). When the deployer reads this file (only one file for all agents exists), it will perform a hot redeployment (WARNING file) or a cold redeployment (CRITICAL file)
– Redeploy messages: written by the deployer, read by agents. Once the deployer has produced a new cloudbook.json dictionary file, it will inform all agents creating a COLD_REDEPLOY file or a HOT_REDEPLOY file. This file will be deleted in the next deployer monitoring period

– stats_agent_<XX>.json: created by agents, read by the stats monitor. This file contains execution stats which also contains information to tune the matrix at make phase
– matrix_<timestamp>.json: created by the stats monitor, read by the maker: contains a new version of the matrix taking into account execution stats
– du_list.json: created by the maker, read by the deployer, this file contains all DUs, in order to be assigned by the deployer to the alive agents
– function_mapping.json: created by the maker, read by the stats monitor. This file contains the mapping between original name functions and final name functions

5 Dynamic Execution

With the previously described architecture, it is possible to perform dynamic (re)deployment and execution of codes. By profiting from the surveillance monitor, it is possible to periodically check changes in the number of available agents and alarms raised by agents in order to perform both a "hot" (without restarting the program) or "cold" (program must be restarted) redeployment. Redeployments are initiated in the following cases:

– Under critical alarms sent by agents (they cannot continue running), the surveillance monitor must restart the deployment and in some cases the maker
– Under warning alarms sent by agents (they can continue running), the surveillance monitor must make a hot redeployment and inform the agents to load the new Cloudbook
– When new agents have been added or others have stopped, in a way in which the new Cloudbook dictionary must be compatible with the previous one, so orphan DUs, stopped agents, new agents, and critical DUs are properly reassigned by Cloudbook.

In order to keep track of the number of available agents, the surveillance monitor will use the agents_grant.json file. Agents will update this file periodically (period is chosen taking into account both the distributed file system synchronization time and processing time of the monitor) and surveillance monitor will explore this file periodically using a larger interval. With the surveillance monitor component, the deployer will never stop because sleeps and wakes up periodically (this strategy is better than a scheduled OS task and allows easily stopping the deployer and the surveillance monitor mechanism).

The dynamic execution also allows improving performance based on collected statistics. The redeployment for improving performance must take into account stats gathered by agents. These stats provided by the agents feed a stats monitor, which dynamically builds a matrix and compares with existing matrix used at current deployment. Stats generated by agents include the number of times that each function has been invoked by each "invoker" function. In order to make it possible, the name of the "invoker" function will be sent at each invocation.

The existing matrix must be an output from the maker. The latter must invoke the graph analyzer to build and fill the matrix only the first time. Therefore, an optional parameter to use existing filled matrix must be included in the invocation to maker, i.e.

it must be possible to do a “remake” and not only a “make”, and for make it possible the matrix parameter is needed. The matrix file used as input is created by the stats monitor and improves the “default” assumptions that maker does when building the matrix. By doing so, the performance can be improved in terms of the way in which the code has been parallelized and distributed, but also in terms of performance based on the underlying infrastructure as additional features can be added for doing an intelligent redeployment. Stats provide real information about invocations among functions and allow taking better decisions when the code of the original program is separated into different DUs, which are executed on different agents. Stats may suggest that certain functions should be deployed together in the same DU.

The way in which the dynamic redeployment is carried out is depicted in Fig. 2.

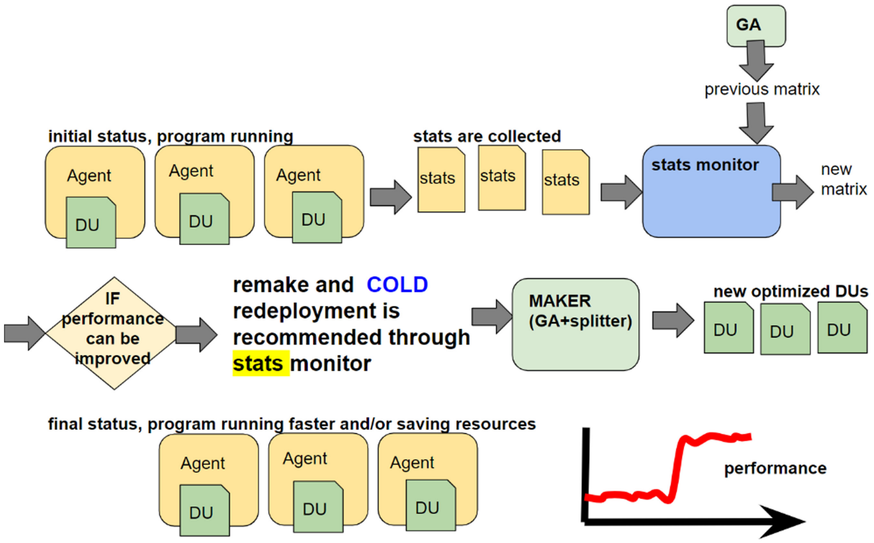
GA
previous matrix
initial status, program running
Agent
DU
Agent
DU
Agent
DU
stats are collected
stats
stats
stats
stats monitor
new matrix
IF performance can be improved
remake and COLD redeployment is recommended through stats monitor
MAKER (GA+splitter)
new optimized DUs
DU
DU
DU
final status, program running faster and/or saving resources
Agent
DU
Agent
DU
Agent
DU
performance

Fig. 2. The Cloudbook dynamic redeployment

6 Results

Experiments have been carried out in two platforms: a group of low-end machines and an HPC cluster.

6.1 Group of Low-End Machines

For proof of concepts tests, platform is composed of four Raspberry Pi2 interconnected with an Ethernet switch and sharing a NFS file system to store the program (DUs) and files published by each agent. The characteristics of this circle of machines are:

– Processor: Broadcom BCM2837B0, Cortex-A53 64-bitSoC@ 1.4 GHz
– RAM: 1 GB LPDDR2 SDRAM
– Wi-Fi+Bluetooth: 2.4 GHz y 5 GHz IEEE 802.11.b/g/n/ac, Bluetooth
– Operating System: Raspbian

In order to test the correctness of Cloudbook, two first examples have been adapted to the tool paradigm in order to include the simple and reduced pragmas that Cloudbook needs to find out within the code in order to successfully make parallel an initial serial code. These two problems are the N-body problem [19] and the tower of Hanoi game [20]. According to the results, they have been used as valid proof of concept for this work.

For the sake of completion, the results related to the N-body problem executed on Cloudbook can be watched in a video [21], where it is demonstrated how the code is run in the four aforementioned raspberries. The time spent in the algorithm by Cloudbook is lower than the sequential version, from a certain number of bodies. The benefit is bigger when the number of bodies processed by one invocation is high, and the communication time becomes non relevant. Regarding the performance and taking into account the test bed, Cloudbook starts performing better than the serial version from ~3,000 bodies on. From this point, the speed up grows linearly, close to a 4x factor as is depicted in Fig. 3.

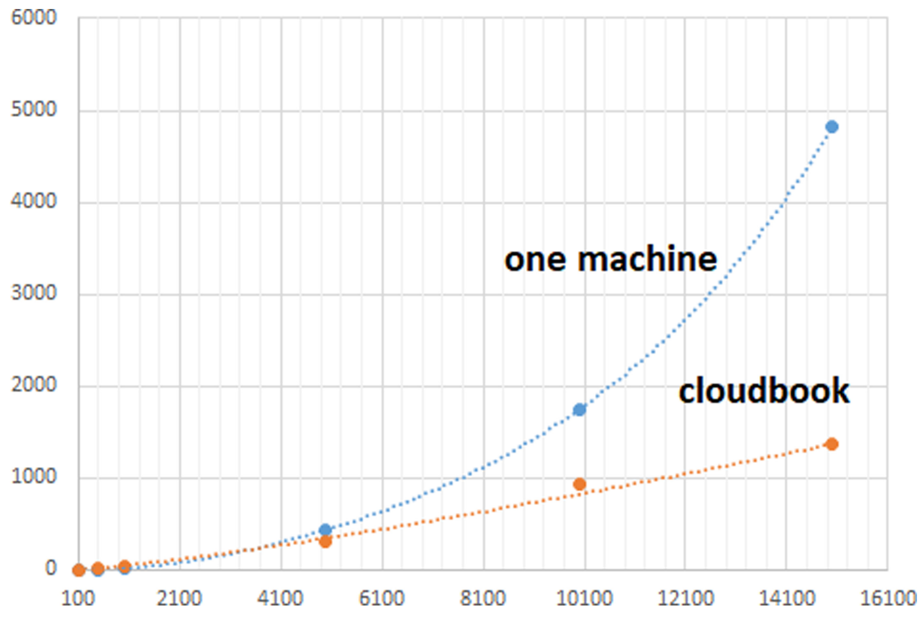

Fig. 3. Results of the N-body execution on the Cloudbook test bed. X-axis reads for number of bodies and Y-Axis for seconds; they are not included in the Figure for readability reasons

With respect to the Hanoi game and in a similar way, it is also found out that Cloudbook provides potentially 4 times bigger stack for recursive invocations in the aforementioned testbed, but what it is most important is to notice how this recursive problem is able not to collapse thanks to Cloudbook. It has been demonstrated that for a ten of pieces a sequential version would crash meanwhile Cloudbook is able to keep on working on finding out the solution. Speed in recursive invocations is not improved but stack size is increased linearly with the number of agents involved.

In order to test the solution proposed with a different approach, some tests have been performed with Cloudbook executing an Intrusion Detection System (IDS). This way, the focus is put most on dataset management and process. The comparison between one machine and Cloudbook execution is shown in Table 1 below.

Table 1. Local and Cloudbook execution times for an IDS.

Data size (lines)	Local execution time (s)	Cloudbook execution time (s)
100,000	5.67	4.25
1,000,000	51.82	25.52
5,000,000	257.06	139.70
10,000,000	529.09	300.96
50,000,000	2,737.52	1,363.32
157,602,189	13,846.43	6,451.53

6.2 HPC Cluster

Two more computationally demanding tests have been carried out on a HPC environment. We run with CloudBook a genetic algorithm in the XULA cluster, located at the CIEMAT data center. We use the new partition of the cluster (upgraded in March 2020, named Xula2), which is composed of 56 computing nodes and connected through IB HDR100. Each node contains 2 processors Intel® Xeon® Gold 6254 (18C, 36T) @3.10 GHz and 192 GB of RAM memory. The common folder for Cloudbook is mounted on a Lustre filesystem.

The genetic algorithm adapted to Cloudbook is DiVoS [22]. DiVoS is a simulation code that finds the minimum energy of a superconducting layer by finding the optimal position of its magnetic vortices. In the genetic algorithm, the chromosomes are the position of the vortices and the fitness function is precisely the (negative) energy of the system. By means of heritage, crossovers, and natural selection rules, the algorithm finds the best individual of the population, i.e. the one with lowest energy and thus the most likely state of the system.

The DiVoS adaptation to CloudBook is rather straightforward: we have parallelized a parameters scan in the input configuration file. In this way, we can easily perform physical parameters sweeps and numerical convergence studies in a fast and easy way from the user point of view. We must point out that this parallelization does not require any communication between the agents. The two tests carried out are intended to show the scaling of the computing time with the number of agents for a fixed problem size and to compare the performance with the Multiprocessing Python built-in library. For Cloudbook, the time measure is the execution time of the cloudbook_run.py program, not taking into account the time needed to make, deploy, or activate the agents.

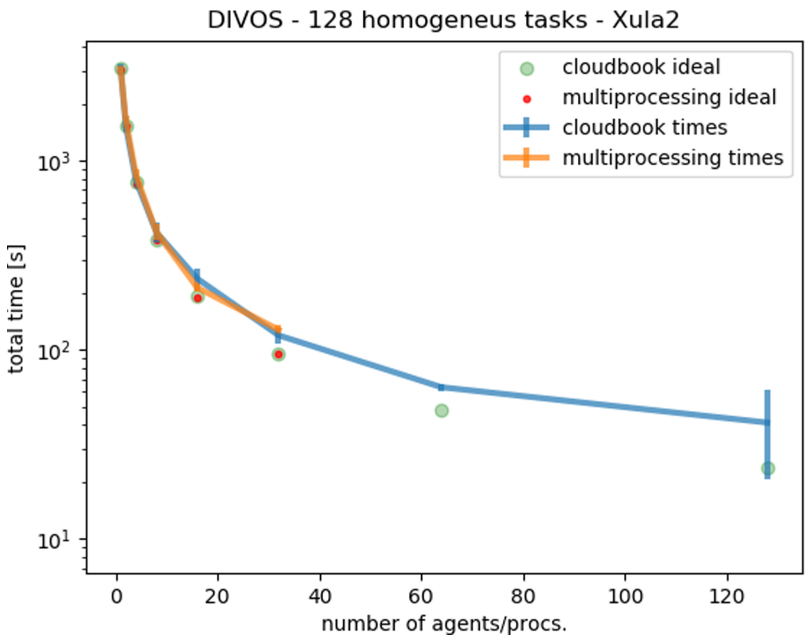

Fig. 4. Results of the DiVoS execution profiting from the Cloudbook solution and the Multiprocessing Python built-in library (homogenous tasks)

In the first test we consider a problem that consists of 128 identical tasks, and measure the execution time in terms of the number of agents (or CPUs) used for the computation both with Cloudbook and the Multiprocessing library. A number of tasks equal to the number of available agents is run simultaneously, with a #__CLOUDBOOK:SYNC__ pragma at the end of each batch of tasks. Each case is executed 5-10 times, using the average value and assuming an error equal to twice the standard deviation. We also calculate the execution time corresponding to ideal scaling in the two cases. The results are plotted in Fig. 4.

In the second test the problem is formed by 80 inhomogeneous tasks. Due to the synchronization step, the scaling here is a bit worse, as can be seen in Fig. 5:

We can extract two conclusions from these tests:

– Cloudbook presents very similar performance as the Multiprocessing library within the error bars.

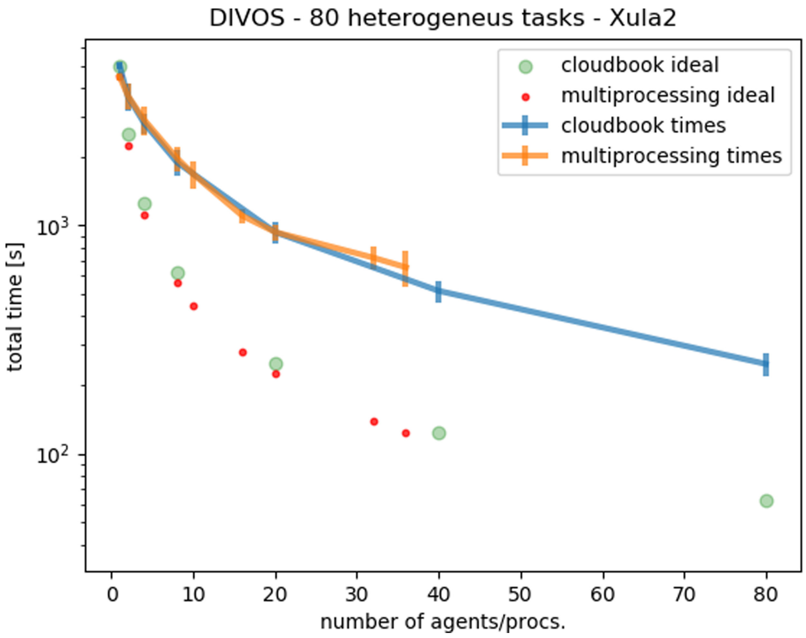

Fig. 5. Results of the DiVoS execution profiting from the Cloudbook solution and the Multiprocessing Python built-in library (heterogeneous tasks)

- Cloudbook can scale up much more than the Multiprocessing library, because the latter is limited to the number of available processors in each node (36 in Xula2) and Cloudbook allows the deployment between any number of nodes.

7 Conclusions

In this work, a new tool called Cloudbook that automatically and unattendedly parallelizes serial codes is presented. Unlike previous similar solutions, it is focused on Python codes and has produced tangible results on production infrastructures at scale, which are also reported via digital content. Cloudbook does not only make the parallelization, but also is aware of the number and main characteristics (performance, bandwidth connection, etc.) that the available resources provide in order to decide a smart distribution of the parallel tasks (DUs) in order to optimize the performance.

The limits of the efficiency of parallel programming with Cloudbook are given by the size of the problem and the cost of communication. Performance results can be improved by taking advantage of the multi-processing in the agents, using their available cores.

Cloudbook follows the model of HPC and HTC computing in a versatile way and can be adapted to a large set of problems, without forcing the programmer to make a distributed design of the problem. The main contributions of Cludbook are:

- Provision of automatic splitting
- Generic, not simply bounded to master-slave based programs, for example

– Valid for both distributed and parallel environments
– Dynamic redeployment based on performance
– Low required level of knowledge

Having demonstrated its correctness, the methodology that Cloudbook applies for making parallel a serial code is also extended to dynamic environments in which resources are continuously integrated and decommissioned into/from the available infrastructure, while the tool successfully responds to that on-the-fly.

References

1. Barve, A., Khandelwal, S., Khan, N., Keshatiwar, S., Botre, S.: Serial to parallel code converter tools: a review. Int. J. Res. Advent Tech. Special Issue National Conference "NCPCI-2016" (2016)
2. Varsha, K.R.: Automatic parallelization tools: a review. IJESC **7**(3), 5780–5784 (2017)
3. Barve, A., Khomane, S., Kulkarni, B., Katare, S., Ghadage, S.: A serial to parallel C++ code converter for multi-core machines. In: Proceedings of the International Conference on ICT in Business Industry & Government (2016)
4. Alsubhi, K.: An architecture for translating sequential code to parallel. In: Proceedings of the 2nd International Conference on Information System and Data Mining, pp. 88–92, April 2018
5. André, F., Le Fur, M., Mahéo, Y., Pazat, J.-L.: The Pandore data-parallel compiler and its portable runtime. In: Hertzberger, B., Serazzi, G. (eds.) HPCN-Europe 1995. LNCS, vol. 919, pp. 176–183. Springer, Heidelberg (1995). https://doi.org/10.1007/BFb0046627
6. Chapman, B., Jost, G., van der Pas, R.: Using OpenMP. The MIT Press, Cambridge (2008)
7. Chavarria-Miranda, D., Mellor-Crummey, J.: An evaluation of data-parallel compiler support for line-sweep applications. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, pp. 7–17. IEEE, New York (2002)
8. Chavarría-Miranda, D., Mellor-Crummey, J., Sarang, T.: Data-parallel compiler support for multipartitioning. In: Sakellariou, R., Gurd, J., Freeman, L., Keane, J. (eds.) Euro-Par 2001. LNCS, vol. 2150, pp. 241–253. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44681-8_36
9. Di Martino, B., Esposito, A., D'Angelo, S., Maisto, S.A., Nacchia, S.: A compiler for agnostic programming and deployment of big data analytics on multiple platforms. IEEE Trans. Parallel Distrib. Syst. **30**(9), 1920–1931 (2019). https://doi.org/10.1109/TPDS.2019.2901488
10. Hsu, A.W.: A Data Parallel Compiler Hosted on the GPU. Indiana University, Bloomington (2019)
11. Kotsifakou, M.: HPVM: heterogeneous parallel virtual machine. In: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 68–80. ACM Digital Library (2018)
12. Iserte, S., et al.: Dynamic management of resource allocation for OmpSs jobs. In: Proceedings of the First Ph.D. Symposium on Sustainable Ultrascale Computing Systems, NESUS COST Action, Timisoara (2016)
13. Becker, T., Karl, W., Schüle, T.: Evaluating dynamic task scheduling in a task-based runtime system for heterogeneous architectures. In: Schoeberl, M., Hochberger, C., Uhrig, S., Brehm, J., Pionteck, T. (eds.) ARCS 2019. LNCS, vol. 11479, pp. 142–155. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-18656-2_11

14. Ramírez-Velarde, R., Tchernykh, A., Barba-Jimenez, C., Hirales-Carbajal, A., Nolazco-Flores, J.: Adaptive resource allocation with job runtime uncertainty. J. Grid Comput. **15**(4), 415–434 (2017). https://doi.org/10.1007/s10723-017-9410-6
15. Becker, T., Busse, P., Schuele, T.: Evaluation of dynamic task scheduling algorithms in a runtime system for heterogeneous architectures. In: 31st International Conference on Architecture of Computing Systems, Braunschweig, pp. 1–8 (2018)
16. Rodriguez-Pascual, M., Mayo-García, R.M., Llorente, I.M.: Montera: a framework for efficient execution of Monte Carlo codes on grid infrastructure. Comput. Inform. **32**, 113–144 (2013)
17. Rubio-Montero, A.J., Rodríguez-Pascual, M.A., Mayo-García, R.: A simple model to exploit reliable algorithms in cloud federations. Soft. Comput. **21**, 4543–4555 (2017). https://doi.org/10.1007/s00500-016-2143-9
18. Kumar, A., Shorey, R.: Performance analysis and scheduling of stochastic fork-join jobs in a multicomputer system. IEEE Trans. Parallel Distrib. Syst. **4**, 1147–1164 (1993). https://doi.org/10.1109/71.246075
19. Heggie, D.C.: The Classical Gravitational N-Body Problem. Encyclopaedia of Mathematical Physics, Elsevier (2006)
20. Romik, D.: Shortest paths in the Tower of Hanoi graph and finite automata. SIAM J. Discret. Math. **20**, 610–622 (2006)
21. Demo of the N-Body proof of concept and how it performs. https://drive.google.com/open?id=193f30luFq22cy8QUjzzWHgMA8zKfUAv4
22. Rodríguez-Pascual, M.A., et al.: Superconducting vortex lattice configurations on periodic potentials: simulation and experiment. J. Supercond. Nov. Magn. **25**, 2127–2130 (2012). https://doi.org/10.1007/s10948-012-1636-8