35th European Modeling & Simulation Symposium
20th International Multidisciplinary Modeling & Simulation Multiconference

2724-0029 © 2023 The Authors.
doi: 10.46354/i3m.2023.emss.020

A parallel proposal with message passing for the implementation of a Pipeline in the development of the video game SIMON

Mario Rossainz-López[1,*], Liosbel Cabrera-Hernández[1], Bárbara Sánchez-Rinza[1] and Manuel Capel-Tuñon[2]

[1] Faculty of Computer Science, Autonomous University of Puebla, Av. San Claudio and 14 Sur Street, San Manuel, Puebla, México, C.P. 72570
[2]Software Engineering Department, College of Informatics and Telecommunications ETSIIT, University of Granada, Daniel Saucedo Aranda s/n, Granada 18071, Spain

[*]Corresponding author. Email address: mrossainzl@gmail.com

Abstract

The design and development of the inter-process communication pattern called Pipeline is presented as a proposal of Parallel Object Composition to solve simple way problems that can be solved with this same parallel control structure. A particular class library called JPMI (Java Passing Message Interface) is used for parallel programming with message passing and to implement an original and particular version of the well-known video game called SIMON with the objective, on the one hand, to show the usefulness of this design within Structured Parallel Programming and, on the other hand, that this proposal serves to guarantee good performance in the execution of real time applications. An example of this type of applications is precisely video games. The parallel algorithm implemented as a Composition of Parallel Objects is based on the development and use of a methodology where the algorithmic design represents the parallel control structure common to a given algorithmic technique that can use the pipeline communication pattern, generating a generic and abstract parallel program from which programs that solve specific problems using the same communication pattern can be derived. The implementation of this proposal within structured parallel programming tries to facilitate to the novice programmer in parallelism the reusability, genericity, and uniformity of code abstract enough to be suitable for any problem that can be solved with a pipeline offered implemented with a parallel message passing structure. This particularized proposal for the implementation of the SIMON video game is compared with another using a thread library called boost and ZeroC Ice for remote invocation of distributed objects. The execution times and speedups of both proposals are compared to identify how similar or different they are in their respective performances with training tests using AI modules with sequences of 500000 colors in a cluster of 2 Intel Xeon CPUs of 8 cores each and 2 nodes, each with 2 NVIDIA cards of 5760 CUDA cores each and a RAM memory of 128 GB.

Keywords: Structured Parallel Programming, HLPC, Pipeline, Message Passing, CSP, JPMI, Videogame, SIMON

1. Introduction

Structured Parallel Programming is based on the modeling, design, construction, and development of communication patterns between the processes defined in an application (McCool et-al, 2012). By achieving this, we can obtain a generic parallel communication pattern that through the use of the object-oriented paradigm we

can particularize to the solution of a problem that can be solved with this pattern through properties such as reuse, inheritance, and polymorphism. This leads to the second objective, which is to facilitate the novice programmer in the "automatic" programming of the parallel part of his algorithmic proposal, focusing his effort on the design and coding of the sequential algorithms of his solution (McCool et-al, 2012). There are currently several communication patterns that represent solution models in the interaction between processes such as Pipelines, Farms, Trees, Cubes, Hypercubes, Mesh, etc., and that are used in different areas and disciplines. The High-Level Parallel Compositions or HLPC model is intended to be a generic model for the design of process communication patterns, which is adapted to a particular pattern in the solution of a sequential problem that can be parallelizable (details can be found in Hoare, 2003). Based on this idea, the HLPC Pipe is created as a communication pattern that implements a Pipeline of processes to solve problems that are decomposed into a series of successive tasks so that the data flows in a certain direction through the process structure and each task is completed one after the other. The problem that is solved by this HLPC model is the development of the video game known as SIMON which consists of the user having to be able to memorize and repeat a sequence of colors that is generated by the application through a pipeline that defines a sequence of colors. The video game is used not only as entertainment or recreational applications but also as applications that help to improve the cognitive skills of people (Simone and López, 2008) as can be the video game presented here. That is why this paper focuses its effort on explaining how structured parallel programming through the HLPC Pipe model can be useful to develop a particular proposal for the implementation of the SIMON video game where the use of the pipeline is inherently natural to use in this parallel development proposal, using message passing programming through a class library that implements the process algebra of Hoare's CSP called JPMI.

2. State of the art

The industry offers parallel hardware platforms such as GPUs, multi-core processors and the cloud, to speed up data processing with respect to uniprocessor contention. For all these platforms performance and optimization of sequential algorithms is reaching its limit. One alternative is to opt for parallel and concurrent programming algorithms at a high level of abstraction by using patterns of communication/interaction between processes. In (Collins, 2011), the effectiveness and applicability of automatic techniques has been explored. FastFlow is a C++ parallel programming framework intended to propitiate high-level, pattern-based parallel programming, as the research work of (Torquati et al, 2014; Aldinucci et al, 2014) pointed out. The framework provides several predefined, general purpose, customizable and composable parallel patterns or algorithmic skeletons such as the pipeline parallel pattern as described in the work of (Torquati et al., 2014). There are currently projects that develop frameworks and offer to users constructs, templates and parallel communication patterns between processes, such as the ParaPhrase project. (Torquati et al., 2015) aimed at developing a new structured design and implementation process for heterogeneous parallel architectures. A more conventional approach to framework-based parallel programming provides application programmers with the possibility of obtaining loop parallelization from sequential code, with a relatively small amount of programming effort. This is the approach followed in (Danelutto and Torquati, 2014) with the 'ParallelFor'. The work carried out in (Ernsting and Kuchen, 2012) offers the library skeleton 'Muesli' that offers a simplified framework to perform parallel programming helps to find correct solutions to general problems. 'Muesli' skeleton also allows us to write one application that can be executed with no change across a variety of parallel machines ranging from simple shared-memory multi-core processors to clusters of distributed-memory multi- and many-core processors, multi-GPU systems and GPU clusters. MALLBA (Alba et al., 2007) is another software tool intended for assisting in the solution of combinatorial optimization problems using generic algorithmic skeletons implemented in C++. Some environments of parallel programming, as the one called SklECL (Steuwer et al., 2011), are based on skeletons and wrappers that make up the fundamental constructs of a coordination language, defining modules that encapsulate code written in a sequential language and three classes of skeletons: control, stream parallel, and parallel data. Finally, OpenMP and Intel TBB are frameworks that facilitate the automatic parallelization of loops and offer common communication structures between processes such as pipelines.

3. Programming with Message Passing in JAVA

Given the importance of having current tools in Java that provide programming with message passing, this paper shows the use and usefulness of the JPMI (Java Passing Message Interface) class library that implements Hoare's CSP process algebra to improve the performance of applications that can be parallelized using this scheme. The JPMI library provides classes to generate processes, communication channels, and sequential, parallel, and alternative compositions of processes, to communicate and synchronize them. The antecedents of this library are shown in the University of Twente project that resulted in what is known as CTJ (Communicating Threads for Java) (Hiderink et-al, 2000; Hiderink-2 et-al, 2000) and in the University of Kent project that culminated in the proposal of the JCSP (Communicating Sequential Process for Java) class library (Welch et-al, 2007). The advantage of JPMI over CTJ and JCSP is that it shows an updated version of the latter, which are now obsolete, and the programmer has a comprehensive set of rules that help eliminate undesirable conditions of parallelism during the design and implementation

phases, such as strict alternation, lack of mutual exclusion, interlocking and infinite waiting. Processes are shown as active objects with the ability to execute on themselves and with other processes by creating a composition of them, while channels are passive objects that serve as a means of communication between the processes that use them. The general communication model is shown in Figure 1. It identifies the fundamental elements involved in communication in message-passing systems (a sending process, a receiving process, a communication channel, the message to be sent/received, and the sending and receiving operations) (Fujimoto, 2000; Palma, 2003).

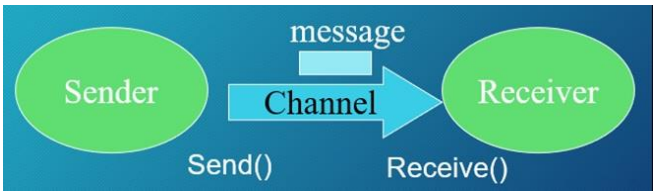

Figure 1. Process Communication Model with Message Passing

3.1. Types of communication between processes (Fujimoto, 2000; Palma, 2003)

- **Direct Communication:** The sender explicitly identifies the receiver of the message in the sending operation and vice versa for the receiving operation by the receiver.
- **Indirect Communication:** The sender and receiver processes are not explicitly identified. Communication is carried out by depositing messages in an intermediate store (mailbox) that is assumed to be known by the processes involved in the communication.

3.2. Synchronization between processes (Fujimoto, 2000; Palma, 2003)

- **Asynchronous communication.** The sending process can perform the sending operation without it being necessary for it to coincide in time with the receiving operation by the receiving process.
- **Synchronous Communication.** There must be a coincidence (appointment or meeting) in the time of the sending and receiving operations by the sending and receiving processes.

3.3. Channel and message characteristics (Capel and Rodriguez,2012)

- **Data Flow.** The data flow passing through a communication channel between two processes can be unidirectional or bidirectional.
- **Channel Capacity.** The ability of channel to store messages sent by the sending process when they are not immediately picked up by the receiving process.
- **Message size.** Messages can be of fixed or variable length.
- **Channels with type or without type.** Some communication schemes require defining the type of data that will flow through the channel, so we can have typed or untyped channels.
- **Passing by copy or by reference.** The information sent by the sender process to the receiver process through a channel is done by making an exact copy of the data (message) or simply sending and receiving the address in the address space where the message is located.

3.4. The JPMI class library

JPMI (Java Passing Message Interface) is a package of classes that implements Hoare's CSP Process Algebra and is used to create processes, process compositions, and inter-process communication channels. JPMI has to implement the Jpmi Process interface and provide an implementation for its run() method which will contain the task that the process wants to carry out when this method is invoked by another one within a composition that can be of one of the allowed types: sequential, parallel, or alternative. The constructor of the process specifies the input channels, output channels, and additional parameters to initialize the state of the process. The run() method is the only public method that a process can invoke directly on another process. Figure 2. shows its architectural design with a class diagram in UML.

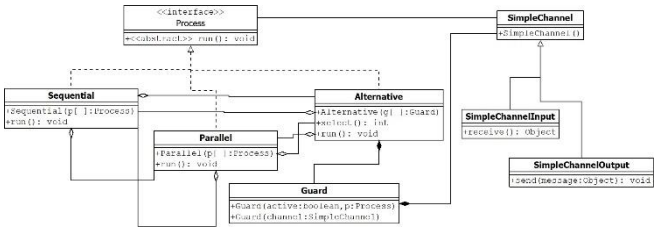

Figure 2. Architectural design of JPMI class library

In JPMI the channels are unidirectional, zero-capacity, untyped (generic), and with send and receive message operations which are of variable length and passed by copy. JPMI is intended to be a bridge between CSP theory and its application in JAVA (for details see Rossainz et-al, 2019).

4. CSP: Communicating Sequential Processes

It is a Process Algebra proposed by Hoare as a formal algebraic language that is used to describe the communication behavior between processes by message passing that can be verified and demonstrated (Davies and Schneider, 1995). With CSP, the behavior pattern of a process can be described in terms of communication events, operators, and other processes. To include events in a process description, the prefix operator is used (see details in Hoare, 2003). There are several types of process composition in CSP. Given two processes P and Q, they can be communicated through:

- **(P; Q). Sequential Composition:** It is a process that behaves as P until this component is finished and then it behaves as Q.
- **(P||Q). Parallel Composition:** is a process where P is capable of executing in any of its events and Q is capable of executing in any of its events. The two processes can cooperate to carry out any common event.
- **(P|||Q). Parallel Composition (with interpolation):** where the two processes P and Q execute independently without cooperation between them on each occurrence of any of their events.
- **(P□Q). Alternative Composition:** It behaves as P if the first action of this process can be executed, otherwise it behaves as Q if the first action of this process can be executed. If both actions can be performed, then the choice between them is made non-deterministically.
- **(PΠQ). Alternative Composition (non-deterministic):** The choice between P and Q is based on an arbitrary selection, without the knowledge of the external environment.
- **STOP.** It is a process that never executes in any event. Describes blocked process behavior.
- SKIP. It is a process that does nothing but terminates completely.

5. The Pipeline and its representation as a Composition of Parallel Objects

The pipeline is a parallel processing technique applicable to a wide range of problems that are partially sequential. With this scheme we can solve a problem by decomposing it into a series of successive tasks so that data flows in a certain direction through the process structure and each task can be completed one after the other (Robbins and Robbins, 1999). In a pipeline each task is executed by a process as shown in Figure 3. Each process that makes up a pipeline is called a "stage" (Roosta and Séller, 1999).

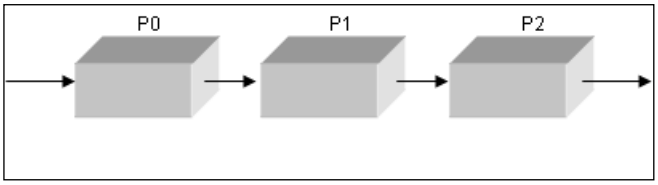

Figure 3. Pipeline Structure

Each stage of the pipeline contributes to the overall problem and passes the necessary information to the next stage with which it is connected. This type of parallelism is seen as a "functional decomposition", as the problem is divided into separate functions that can be executed individually and independently (Robbins and Robbins, 1999; Roosta and Séller, 1999). An algorithm that solves a given problem can be formulated as a pipeline if it can be divided into some functions that could be executed by the pipe stages. Thus, if a problem can be divided into a series of sequential tasks, the pipeline approach can provide increased execution speed for the following three types of computations:

1. When more than one instance of the entire problem can be executed in parallel.
2. Or a series of data can be processed and each of these is used in multiple operations.
3. Or if the information demanded by the next process to start its computation is passed after the current process has completed all its internal operations.

With this technique, many of the computational problems that are carried out sequentially can be easily parallelized as a pipeline (for more details, see Rossainz et-al, 2018).

This PipeLine technique has been developed as a parallel object composition or HLPC (an acronym for High Level Parallel Composition) applicable to a wide range of problems that are partially sequential, such that the HLPC Pipe guarantees code parallelization of the sequential algorithm using the Pipeline pattern. A HLPC represents the composition of a set of parallel objects of three types: A Manager object that represents the HLPC itself. The Manager controls the references of a set of objects (a Collector object and several Stage objects), whose execution is carried out in parallel and coordinated by the Manager itself. The Stage objects encapsulate a client-server interface established between the Manager and the slave objects, which are passive objects containing the sequential algorithm for the solution of a problem, and a Collector object, which is an object in charge of storing in parallel the results received from the connected Stage objects. The Manager, Collector, and Stages objects are Parallel Objects (PO) that can exploit both the parallelism between objects (inter-object) and their internal parallelism (intra-object) (Corradi and Leonardi, 1991).

A PO has a structure similar to that of an object in Java, but in addition includes a scheduling policy that specifies how to synchronize one or more operations of the object class that can be invoked in parallel (Corradi and Leonardi, 1991; Danelutto, 1999). Parallel objects support single inheritance with multiple interfaces, which allows for deriving a new PO specification from an existing one. A HLPC has the following properties: synchronous, asynchronous, and future asynchronous communication modes between the parallel objects of the HLPC, objects with internal parallelism, availability of synchronization mechanisms; Maximum Parallelism, Mutual Exclusion and Producer-Consumer type Synchronization, availability of generic type control, transparency in the distribution of parallel applications and satisfactory performance: Programmability, Portability, and Performance (Rossainz et-al, 2014). Figure 4 shows the model of the Pipeline parallel communication pattern as a HLPC.

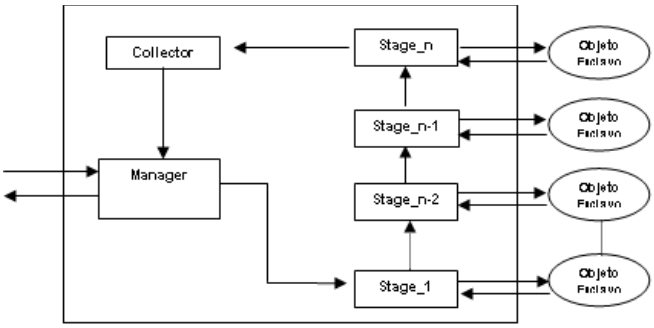

Figure 4. Pipeline model as a parallel object composition or HLPC

6. Implementation of the SIMON video game as PIPE-HLPC using JPMI

A video game is a real-time graphical application with an explicit interaction between the user and the video game itself. The notion of real-time then implies that the video game must make the user have a continuous feeling of realism when playing [19],[20]; this is achieved by generating a 3-step cycle: The user visualizes a rendered image, the user interacts with the application based on what he visualizes and based on that interaction the application responds with an output. This cycle must be executed quickly and constantly so that the user feels immersed in the game and does not have the feeling of seeing static images. Technically this means that the video game must generate a certain number of images per second (frames) based on the interaction with the user and it is precisely here where parallelism and concurrency can help to achieve this accelerated and uninterrupted execution of the video game. The proposal presented in this paper of using the CPAN Pipe programmed with the JPMI library can achieve this goal. As a case study, we show below a proposal for parallelization in the design and implementation of the well-known SIMON video game, which consists of the player having to be able to memorize and repeat a sequence of colors that are generated by SIMON (see Figure 5). We first created the artificial intelligence module responsible for the generation of a video game color sequence; in particular, we adopted the idea of (Rahman and Bawiec, 2023) to incorporate a genetic and deep learning algorithm that represents the Slave Object of the HLPC Pipe model as an instance of the functionality to be executed by a HLPC Stage. Next, using the Process interface of the JPMI library we create the parallel objects of the HLPC Pipe model of Figure 4, particularized to the design of the video game, that is to say, the Manager Object of this new HLPC Pipe which we will call Pipe-HLPC-SIMON is a process that, by a first input channel, will receive in each opportunity for the user the sequence of colors that SIMON defines as the pattern to follow. This sequence will generate the Pipeline of the model according to the number of colors that integrate it, a Stage Object of the pipeline for each color in the sequence generated at the current time instant.

Figure 5. SIMON video game implemented as CPAN-HLPC

In the beginning, a first Stage is generated with two input channels and one output channel. The first input channel is connected to the output channel of the Manager to receive the user's sequence, which at the beginning of the game will be of only one color, the one defined by the first Stage that receives through its second input channel from its associated slave object (AI deep learning and genetic algorithm adopted from Rahman and Bawiec, 2023) and compares it with the color received from the Manager to verify that the sequence defined by the user is the same as the one generated by SIMON and the result of this comparison will be sent to the Collector object that will be the third process of the model that receives through its input channel this comparison result to give response to the user through the Manager that receives this result through another input channel and informs the user if the sequence is correct or not. Again, the same process is repeated, generating a second Stage connected to the first one, and then a third Stage connected to the second one, and so on until the user makes a mistake in the sequence that must be followed and that is being dictated by SIMON. The generated pipeline represents the color sequence created by SIMON that the user must follow. Each Stage process will pass to the next one (through the output and input channels connected between the neighboring Stages) a hit or miss flag according to the user's progress in the generation of the sequence, which will be sent to the Collector so that it can formulate the final result and send it to the Manager, which in turn will indicate to the user whether he can continue playing or not. The game will count the number of correct colors generated by the user in the generation of the sequence until the latter fails and will serve for the genetic and deep learning algorithm adopted from (Rahman and Bawiec, 2023) to learn from the user and then, in the next game, generate a more complicated sequence of colors. As can

be seen, the pipeline of the model is dynamic, growing in real-time as the user's color sequence remains correct, until the user fails. The Parallel objects of the Manager, Stages, and Collector are running inside a Parallel Process Composition generated using the Parallel class of the JPMI library based on the behavior that this new process must have, modeled through the CSP algebra. The new graphical model of the Pipe-HLPC-SIMON is illustrated in Figure 6 and corresponds to what has been described so far.

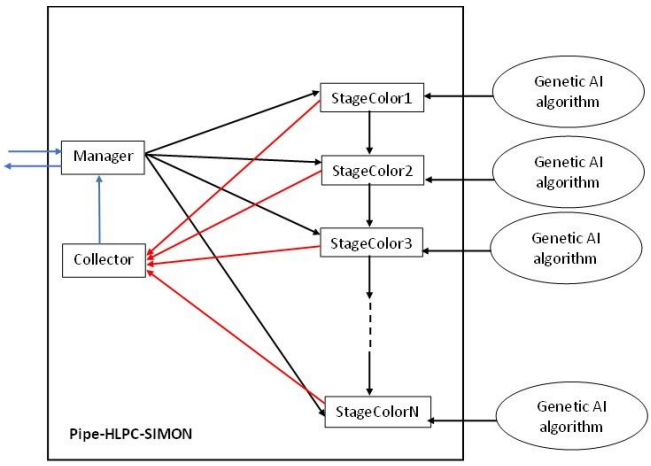

Figure 6. Pipe-HLPC-SIMON model

7. Pipe-HLPC-SIMON Performance Analysis and Results

To measure the performance of the SIMON video game implemented with the CPAN model, we took as a comparative reference the implementation of (Vallejo and Martín, 2015) of the same video game where it uses OGRE as a rendering engine and a thread library called boost and ZeroC Ice for the remote invocation of distributed objects. Both proposals, the one in (Vallejo and Martín, 2015) and the one presented in this paper were executed in a cluster with 2 Intel Xeon CPUs of 8 cores each and 2 nodes, each with 2 NVIDIA cards of 5760 CUDA cores each and a RAM of 128 GB. For the case of our model, the server node was hosted on a CPU where the Manager and Collector objects were placed, while the Pipeline that is dynamically generated together with the slave objects was hosted as clients on the CUDA cores of the corresponding GPUs (see Fig.6.). In both proposals a training was performed to carry out the corresponding speedup's calculations with color sequences of 500000 elements which are shown in the graph in Figure 7.

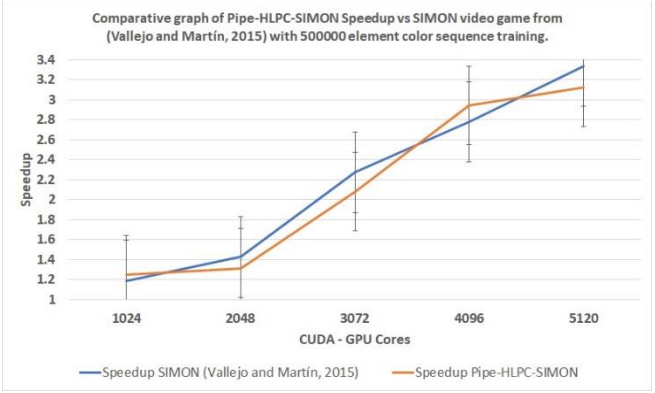

Figure 7. Comparison of speedup scalability found in the Pipe-HLPC-SIMON vs SIMON video game (Vallejo and Martín, 2015) with 500000 element color sequence training.

The graph shows the population mean of a Normal Probability distribution, from a training set of both proposals with correct sequences of 500000 colors and the use of 1024 to 5120 CUDA cores in increments of 1024. Each color of a sequence produces a delay of 0.2 seconds to be obtained. The average of the sequentially generated sequences was approximately 50 hrs of execution, while the average parallel execution times of the 2 proposals are shown in Table I.

Table 1. Average run times in hours of the SIMON (Vallejo and Martín, 2015) VS Pipe-HLPC-SIMON video game with 500000 element color sequence training.

CUDA-CORES	SIMON runtime (hours) (Vallejo and Martín, 2015)	Pipe-HLPC-SIMON runtime (hours)
1024	42	40
2048	35	38
3072	22	24
4096	18	17
5120	15	16

As can be seen both in the values of Table 1 and the graph of Figure 7, the performance of the Pipe-HLPC-SIMON is almost identical to that shown with the SIMON video game proposal of (Vallejo and Martín, 2015). The difference in the errors of the comparative in the graph of Figure 7 of the speedups is very small since the execution times of both proposals under the same conditions are very similar (see Table 1).

8. Conclusions

We have presented the design of a composition of parallel objects to model and implement the Pipeline communication structure as a High-Level Parallel Composition or HLPC whose implementation was carried out using the JPMI library for programming with message passing, particularly in the case study of the well-known video game SIMON. The implementation of this video game was carried out based on the Pipe-HLPC-SIMON model (see Figure 6) through a parallel composition of 3 types of processes: Manager, Collector, and Stages to create a dynamic

pipeline where each Stage of the Pipe represents a color that is randomly generated by the video game in a sequence that must be followed correctly by the user. The performance analysis of this proposal was made by comparing both execution times and acceleration with the proposal proposed by (Vallejo and Martín, 2015), the results of which are shown in Figure 7 and Table 1 and illustrate the similarity between these two implementations even though they were designed and developed with different models in the design and coding of algorithms. The performances are considered good given the conditions of inputs and outputs to and from the video game and the hardware platform used (see Section 6 of this paper) for its execution and speedup. In future work we intend to demonstrate the genericity of the implemented HLPC, using it and adapting it to the development of new video games that require the use of the pipeline for parallelization with AI techniques, for example, the implementation of "alphabet soup" or "crossword puzzles".

References

Alba, E., Luque, G., Garcia, J. and Ordonez, G.: MALLBA: a software library to design efficient optimization algorithms, International Journal of Innovative Computing and Applications, Vol. 1, No. 1, pp.74–85. (2007).

Aldinucci, M., Danelutto, M., Kilpatrick, P. and Torquati, M.: 'FastFlow: high-level and efficient streaming on multi-core', in Pllana, S. and Xhafa, F. (Eds.): Programming Multi-core and Many-core Computing Systems, Wiley. (2014).

Capel I. M., Rodriguez V. S: Sistemas Concurrentes y Distribuidos. Teorìa y Pràctica: Copycentro Editorial: España (2012).

Collins, A.J.: Automatically Optimising Parallel Skeletons, MSc thesis in Computer Science, School of Informatics University of Edinburgh, UK. (2011).

Corradi A., Leonardi L.: PO Constraints as tools to synchronize active objects. Pp: 42-53. Journal Object Oriented Programming 10. (1991).

Danelutto, M.; Orlando, S; et al.: Parallel Programming Models Based on Restricted Computation Structure Approach. Technical Report-Dpt. Informatica. Universitá de Pisa (1999).

Danelutto, M. and Torquati, M.: Loop parallelism: a new skeleton perspective on data parallel patterns, in Proc. of Intl. Euromicro PDP 2014: Parallel Distributed and Network-based Processing, Torino, Italy. (2014).

Davies J. and Schneider S.: Real-Time CSP, UK, (1995).

Ernsting, S. and Kuchen, H.: Algorithmic skeletons for multi-core, multi-GPU systems and clusters, Int. J. of High-Performance Computing and Networking, Vol. 7, No. 2, pp.129–138. (2012).

Fujimoto: Parallel and Distributed Simulation Systems: Wiley-Interscience: USA, (2000).

Hiderink J., Broenink J., Bakkers A.: Communicating Threads for Java: University of Twente: Draf-Rev 5.: Netherlands (2000).

Hiderink-2 J., Broenink J., Vervoort W., Bakkers A.: Communicating Java Threads. University of Twente: Netherlands (2000).

Hoare C.A.R.: Communicating Sequential Processes: Prentice Hall, London, UK, (2003).

McCool M., Robison A.D., and Reinders J. Structured Parallel Programming. Patterns for Efficient Computation. Morgan Kaufmann Publishers Elsevier. USA. (2012).

Palma J.T., Garrido M. C., et al: Programación Concurrente: Thomson. España (2003).

Rahman A., Bawiec M., Simon Says. Amazon Web Services. (2023). Recuperado de: https://aws.amazon.com/es/deeplens/community-projects/deeplens_simon_says/

Robbins, K. A., Robbins S.: UNIX Programación Práctica. Guía para la concurrencia, la comunicación y los multihilos. Prentice Hall. (1999).

Rossainz M. and Capel M. Design and implementation of communication patterns using parallel objects. Especial edition, Int. J. Simulation and Process Modelling, Volume 1, Number 17. (2017).

Rossainz M., Sánchez B., Rangel A., Ballinas A.L., JPMI: Un paquete de clases en Java para la Programación Paralela con Paso de Mensajes. Modelado, TIC y Sistemas Distribuidos: avances y aplicaciones. Dirección General de Publicaciones. BUAP, México (2019).

Rossainz M., Capel M., Carrasco O., Hernández F., Sánchez B. Implementation of the Pipeline Parallel Programming Technique as an HLPC: Usage, Usefulness and Performance. Annals of Multicore and GPU Programming. Volume 4, Number 1, Spain (2018).

Rossainz M., Pineda I., Dominguez P.: Análisis y Definición del Modelo de las Composiciones Paralelas de Alto Nivel llamadas CPANs. Modelos Matemáticos y TIC: Teoría y Aplicaciones. Dirección de Fomento Editorial. ISBN 987-607-487-834-9. Pp. 1-19. México. (2014).

Roosta, Séller: Parallel Processing and Parallel Algorithms. Theory and Computation. Springer (1999).

Simone B., López C. Breve historia de los videojuegos. Athenea Digital, Revista de Pensamiento e Investigación Social. Número 14, Universidad Autónoma de Barcelona Barcelona España (2008).

Steuwer, M., Kegel, P. and Gorlatch, S.: SkelCL a portable skeleton library for high-level GPU programming, Proceedings of the 16th IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments, May, Anchorage, AK, USA. (2011).

Torquati, M., Aldinucci, M. and Danelutto, M.: 'FastFlow documentation', Parallel programming in FastFlow, Computer Science Department, University of Pisa, Italy. (2014).

Torquati, M., Aldinucci, M. and Danelutto, M.: FastFlow Testimonials, Computer Science Department, University of Pisa, Italy. (2015)

Vallejo D., Martín C. Desarrollo de Videojuegos. Un Enfoque Práctico. Volumen 1. Arquitectura del Motor. Creative Commons License. España (2015).

Welch P., et al: Integrating and Extending JCSP: Communicating Process Architectures. IOS Press. (2007).