36th European Modeling & Simulation Symposium, 015
21th International Multidisciplinary Modeling & Simulation Multiconference

2724-0029 © 2024 The Authors.
doi: 10.46354/i3m.2024.emss.015

Pipeline Representation Using Generic Parallel Algorithms in Heterogeneous Parallel Computing. A comparative analysis

Mario Rossainz-López[1,*], Bárbara Sánchez-Rinza[1], Zuriel López-Sosa[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

Using Parallel Objects and Structured Parallel Programming, the parallel representation of the Communication Pattern between Processes called Pipeline is shown, whose implementation is carried out through different models of Generic Parallel Algorithms within Heterogeneous Parallel Computing (HPC): As a Parallel Design Pattern or PDP, as High-Level Parallel Composition or HLPC and as Flow Graph Interfaces or FGI through programming with message passing. An original and effective development methodology for new programmers in parallelism is proposed for each of them. In these three proposals for Generic Parallel Algorithms, the same problem-example is solved as a case study, and a comparative analysis of their models and designs is carried out, as well as the performance obtained in the execution of said proposals and demonstrate their usefulness, programmability, and performance. The objective of this work is to provide the programmer and/or novice user with different multicore programming approaches so that without much effort they can develop their programs according to a sequential programming style, obtaining automatically, easily and the counterpart parallelization of your code with the help of a specific programming environment like the one proposed.

Keywords: Parallel Objects, Structured Parallel Programming, Generic Parallel Algorithms, HPC, HLPC, PDP, FGI, Pipeline

1. Introduction

The present work proposes the design of algorithms that can be parallelized using Parallel Design Patterns (PDPs), High-Level Parallel Compositions (HLPCs), and Flow Graph Interfaces (FGI), as an original proposal for Heterogeneous Parallel Computing, whose definition is found in (Voss, et al., 2019) so that the transformation of existing sequential applications into parallel applications for multiprocessor environments is easy to carry out by the novice programmer when using the specific programming environment proposed here. The Intel OneApi Development Kit is used to program the PDPs and HLPCs using SYCL (Reinders-Hames, et al., 2021), the Threading Building Blocks (TBB) library (Voss, et al., 2019) is used to program the FGIs, and the Intel DevCloud cluster (Bockhorst, 2021). It offers us access to CPUs and GPUs to develop, test and execute applications that are solved with the use of the

pipeline through the 3 proposals. We intend, through Structured Parallel Programming, Parallel Objects, and Object-Oriented Programming (McCool, Robison, and Reinders, 2012), to model, design, and implement specific PDPs, HLPCs, and FGIs. An original and effective development methodology for new programmers in parallelism is proposed for each of them. A Parallel Design Pattern or PDP is defined as a class of algorithms that solve different problems and that have the same control structure with which we represent the pipeline (Collins, 2011), while a High-Level Parallel Composition or HLPC is the composition of a set of parallel objects of three types: A Manager object that controls the references of a set of objects (an object called Collector and several objects called Stage (Brinch Hansen, 1993). A Flow Graph Interface or FGI is a generic parallel algorithm that raises the level of programming abstraction allowing us to express parallelism without having to worry about every low-level detail (Voss, et al., 2019). Finally, this work shows for each Generic Parallel Algorithm, a creation procedure for the definition of the pipeline, under the same way of solving them together with the performance analysis of each one separately and a comparison of their accelerations and execution times. This work is organized by 6 sections. The first section shows the introduction and summary. The second section talks about the background and the state of the art that shows the similarity of existing proposals with the one presented here. The third section talks about the theoretical framework: structured parallel programming, parallel objects, and message passing programming. In the fourth section we explain the design and development methodology of the PDP, HLPC and FGI. The fifth section shows the case study of parallel fractal generation using the pipeline pattern represented as a PDP, HLPC and CGI, and a comparison of the performance and acceleration in their executions. Finally, the sixth section shows the conclusions and future work of this proposal.

2. Background and state of the art

Transforming existing sequential applications into parallel applications for multiprocessor environments has been of great interest for decades. However, currently, there is no single solution for the parallelization of these applications, neither semi-automatically, nor much less automatically since the proposed solution algorithms together with the different parallelization techniques used have a lot to do with it. In (Collins, 2011), the effectiveness and applicability of automatic techniques have been explored, for example:

- On the other hand (Torquati, Aldinucci and Danelutto, 2014) proposes FastFlow as a framework used to accelerate calculations. FastFlow is a parallel programming proposal in C++ that attempts to provide the programmer with high-level parallel programming.
- Currently, some projects develop frameworks and offer users constructs, templates, and parallel patterns of communication between processes that can help accelerate the execution of applications in heterogeneous parallel environments that use CPUs, GPUs, and FPGAs, such as the ParaPhrase project (Torquati, Aldinucci and Danelutto, 2014).
- A more conventional approach is the well-known automatic parallel programming that provides application programmers with the possibility of obtaining loop parallelization and little else from sequential code with relatively little effort to perform (Danelutto and Torquati, 2014); However, this proposal can be somewhat limiting in obtaining good performance in the execution of said applications, which is complicated by the complexity of the algorithm that solves the problem.
- The use of Threading Building Blocks or TBB, which was born more than 10 years ago as a solution for writing parallel programs in C++, which has become the most popular support. and extensive for parallel programming (Voss, et al., 2019). TBB has been the product of parallel programming experts at Intel and is part of the Intel oneAPI Base Toolkit, which is the other proposal that we use for the development of the Generic Parallel Algorithms that we work on.
- OneAPI includes among its main development tools, C/C++ and Fortran compilers, application profiling tools, and optimized libraries. The Data Parallel C++ compiler (DPC++) stands out, providing all the features of the standard C++ compiler plus instructions for data parallelism and heterogeneous computing, which is the commercial implementation of the SYCL proposal that facilitates the portability of applications between architectures as diverse as CPUs, GPUs, FPGAs (Reinders-Hames, et al., 2021; Bockhorst, 2021).

3. Theoretical framework

Our theoretical framework is composed of four fundamental definitions: Structured parallel programming, parallel objects, parallel programming with message passing, and Generic Parallel Algorithms.

3.1. Structured Parallel Programming

It is based on the use of predefined communication/interaction patterns between the processes of a user application such as the Pipeline (Danish and Farooqui, 2013). This approach is based on the abstraction of the interaction pattern that allows us to design applications capable of using it and particularizing it to the solution of a specific problem. The encapsulation of an inter-process communication

pattern must follow the principle of modularity and must provide a basis for obtaining effective reusability of the parallel behavior of the software entity that it implements. When this is achieved, a generic parallel pattern is created that provides a possible representation of the interaction between the processes which is independent of their functionality. The contribution of this type of programming is that, instead of programming a parallel application from the beginning, now it is enough to identify the communication pattern between processes appropriate for the parallelization of the problem.

However, the identification and unambiguous definition of a complete set of communication patterns between processes of a parallel application is still far from being a solved problem, since there is no sufficiently general agreement that allows formally defining their semantics (Corradi and Zambonelli,1991). What this work proposes is the definition and use of a PDP, a HLPC and an FGI, as generic parallel algorithms adaptable through the mechanisms of inheritance, composition and/or aggregation of the Object Orientation paradigm, to the particular needs of a application. In this way, the user applications themselves are the ones that specify the semantics of these generic algorithms based on the requirements of the software that is intended to be developed.

3.2. Parallel Objects

Parallel Objects (PO) are objects with the ability to execute themselves. Applications that use PO can exploit both parallelism between objects (inter-object) and parallelism within them (intra-object) (Corradi and Leonardi, 1991). A PO has a structure similar to that of an object in C++, but it also includes an a priori scheduling policy that specifies how to synchronize one or more operations of the object class that can be invoked in parallel (Theelen, et al., 2007). Synchronization policies are expressed in terms of restrictions when parallel service requests occur in a PO so that they can manage several executions flows concurrently and at the same time guarantee the consistency of the data being processed. These restrictions are the following:

MAXPAR – which is the maximum parallelism that indicates the maximum number of processes that can run at the same time within a component in the PO model being described.

MUTEX - Performs a mutual exclusion between processes that want to access a shared object. It preserves critical sections of code to be executed by a single process at a time, as well as allowing it to gain exclusive access to resources.

SYNC - Synchronization of the producer/consumer type, used to program the methods or functions of the POs so that the processes that use them are synchronized in the use of resources.

In addition, every PO provides different types of communication:

The synchronous mode that stops the client activity until the active server object gives it the response.

The asynchronous mode that does not force waiting on the client activity. The client simply sends the request to the active server object and continues its execution.

The asynchronous future mode that makes the client activity wait only when, within its code, the result of the method is needed to evaluate an expression, (Lavander and Kafura).

All parallel objects derive from the definition of "class" plus the incorporation of the process planning policy. Objects of the same class share the same behavioral specification contained in it, from which they are instantiated. Parallel objects support multiple inheritance, which allows a completely new PO specification to be derived from one that already exists (Corradi and Leonardi L., 1991; Danelutto, 1999).

3.3. Parallel Programming with Message Passing

In the general programming model with message passing, the fundamental elements that make it up are identified: a sender process, which is the one who sends the message by executing a send operation, a receiver process, which is the one who receives the message by executing a reception operation, a communication channel through which the message travels; and the message itself to be sent/received (Fujimoto, 2000).

The types of communication between processes that are worked on are:

- Direct Communication: The sender explicitly identifies the receiver of the message in the sending operation and vice versa.
- 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 interested in the communication.

The types of synchronization between processes that are worked on are:

- Asynchronous Communication. The sending process can carry out the sending operation without it being necessary for it to coincide in time with the reception operation by the receiving process.
- Synchronous Communication. The sending and receiving operations must coincide (appointment or meeting) in time with the sending and receiving processes.

The characteristics considered in the communication

channels are the following:

- Data flow. The flow of data passing through a communication channel between two processes can be unidirectional or bidirectional.
- Canal Capacity. The communication link may store the messages sent by the sending process when they are not immediately collected 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, therefore we can have typed or untyped channels.
- Pass by copy or by reference. The information sent by the sending process to the receiving 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 Pipeline

It is a parallel processing technique applicable to a wide range of partially sequential problems, that is, with this scheme, we can solve a problem by decomposing it into a series of successive tasks so that the data flows in a certain direction and each task can be completed one after another (Robbins and Robbins, 1999). In a pipeline each task is executed by a processor or process as shown in Figure 1. Each process or processor that makes up a pipeline is usually called a "stage" (Roosta, 1999).

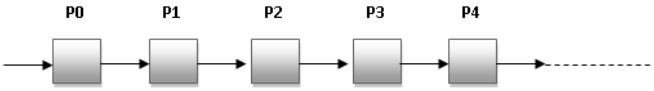

Figure 1. Structure of a Pipeline

Each stage of the pipeline contributes to the overall problem and passes necessary information to the next stage with which it is connected. This type of parallelism is seen as a form of "functional decomposition" or also called "segmented computing" since the problem is divided into separate functions that can be executed individually and independently (Robbins and Robbins, 1999; Roosta, 1999). An algorithm that solves a certain problem can be formulated as a pipeline if it can be divided into a series of functions that could be executed by the stages of the pipeline. Each stage of the pipeline must compute a set of items that, to be processed, require information previously prepared by the previous stage of the pipeline. Once the item has been processed, it has to be sent to the next stage of the pipeline. For simplicity, the algorithm assumes that each stage computes "m" items with the same execution time for each item within the corresponding stage. The exception is in the first stage of the pipeline who does not receive from any other stage and in the last stage who does not send to any other (Wilkinson and Allen, 1999).

If a problem can be divided into a series of sequential tasks, the pipeline approach can provide increased execution speed in the following three types of calculations taken from (Wilkinson and Allen, 1999).

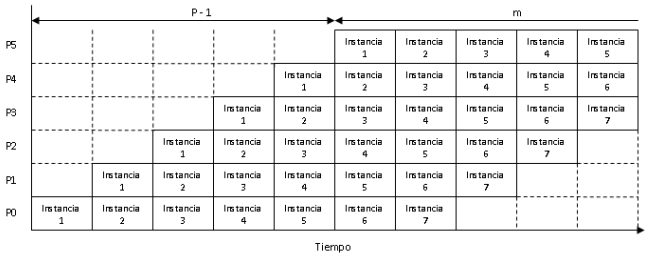

Figure 2. Space-Time Diagram of a pipeline

TYPE A: When more than one instance of the complete problem can be executed in parallel. Figure 2 shows a space-time diagram of the use of the pipeline in this type of calculation. The diagram assumes that all processes have the same execution time to complete their task. Each period is called “a pipeline cycle.” Therefore, each instance of the illustrated problem requires 6 sequential processes: P0 to P5, generating a ladder effect, which upon completion completes an instance of the problem in each “pipeline cycle”. With p-processes (stages) of the pipeline and m-instances of the problem, the number of “pipeline cycles” to execute the m-instances is m+p-1 cycles.

TYPE B: When a series of data can be processed and each of these is used in multiple operations: Appears in arithmetic calculations where a series of data is processed in sequence, such as, for example, multiplying elements of a matrix. In such a calculation, individual elements enter the pipeline as a sequential series of numbers. This type of calculations is illustrated in Figure 3, where, as an example, there are 10 processes (stages) of the pipeline and 10 elements d0 to d9 which are being processed. With p-processes and n-data elements, the overall execution time is again (p-1)+n pipeline cycles assuming that these are all equal.

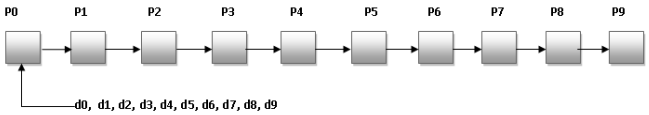

Figure 3. Pipeline for arithmetic calculations

TYPE C: If the information required by the next process to start its calculation is passed before the current process has completed all its internal operations: This type of calculation is used in parallel programs where there is only one instance of the problem to be executed, but Each process (stage) can pass information to the next so that the latter can complete

its task. Figure 4 shows the space-time diagrams when information is passed from one process to another in the pipeline before the completion of the execution of a process.

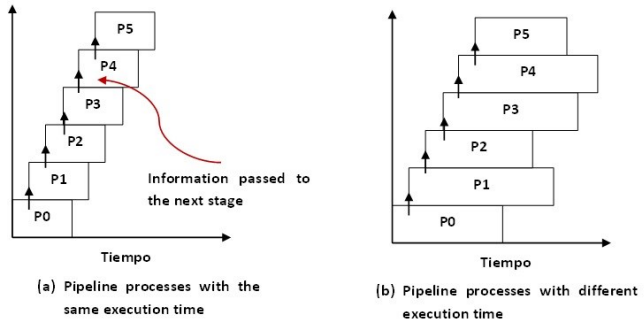

Figure 4. Pipeline processing where the information passes from one stage to another before the completion of the execution of the stages

4. Generic Parallel Algorithms

A Generic Parallel algorithm is an execution pattern common to more than one problem that can be solved in the same way; often represented as function or class templates that capture many of the processing patterns that are the cornerstone of multithreaded programming. This proposal aims to apply them instead of writing our parallel implementations, focusing our effort on designing the sequential solution of the problem to be solved. There are many proposals for this type of pattern such as those proposed by (Mattson, et al., 2004) where it is said that programmers need to work through four spaces in the design of any Generic Parallel Algorithm: find concurrency, identify the algorithmic structure of the parallelism or pattern, identify support structures and define implementation mechanisms. The type of Generic Parallel Algorithm considered in this work is the one proposed by TBB, which starts from a single execution thread. When a thread encounters a parallel algorithm, it distributes the work associated with that algorithm among several threads. When all pieces of work are resolved, execution is merged back and continues again on the initial single thread. The generic algorithms available in TBB are grouped into the following categories: Functional Parallelism, Simple Loops, Complex Loops, Pipelines, and Sorting.

4.1. Parallel Design Patterns

A Parallel Design Pattern or PDP is defined as a class of algorithms that solve different problems and have the same control structure. Examples of this are the PDPs shown in Table 1. For each PDP, a Generic Algorithm is created that defines the common control structure for those problems that can be solved with the same algorithmic design technique. The Generic Algorithm is commonly called the Algorithmic Skeleton (Ernsting and Kuchen, 2012). Subsequently, from a general parallel algorithm, two or more Model Programs are derived that illustrate the use of the PDP to solve specific problems. A Generic Algorithm includes some data types that are not specified and procedures that vary from one application to another. A Model Program is obtained by replacing these data types and procedures with the corresponding data types and procedures of a sequential program that solves a specific problem. In other words, the essence of this proposal is that a model program has a parallel component that implements a PDP and a sequential component for a specific application (Figure 5).

Table 1. Parallel Paradigms and their communication patterns

PDP	Model Program		Communication Pattern
Total Pairs	1.	Householder	Process Pipeline
	2.	N-Body	
Tuple Multiplication	1.	Matrix Multiplication	Process Pipeline
	2.	Graphs Routes	
Divide & conquer	1.	Sorting	Process Tree
	2.	Search	
Cellular Automata	1.	Laplace	Process Matrix
	2.	Simulation	

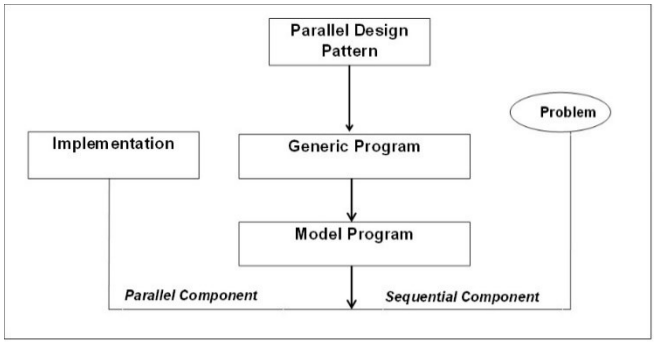

Figure 5. Abstract Model of a Parallel Design Pattern (PDP)

PDP Development Methodology:

1. Identify one, two, or more computational problems with the same control structure.
2. For the identified problem(s), write a tutorial that explains your computational theory and includes a complete program.
3. Write a parallel program for programming the PDP.
4. Test the parallel program on a sequential computer.
5. Derive a parallel program for the particular problem(s) to be solved by substituting data types, variables, procedures, etc., and analyze the complexity of the programs.
6. Rewrite the parallel programs in an implementation language and measure their performance on a multicomputer.
7. Write clear descriptions of parallel programs.

8. Publish the programs and their descriptions in their entirety.

4.2. High Level Parallel Compositions

A HLPC is the composition of a set of parallel objects of three types: A Manager object that represents the HLPC itself and makes it an encapsulated abstraction that hides its internal structure. The Manager controls the references of a set of objects (an object called Collector and several objects called Stage), which represent the components of the HLPC and whose execution is carried out in parallel and must be coordinated by the manager (Rossainz, et al., 2014). The Stage objects are responsible for encapsulating a client-server type interface that is established between the Manager and the slave objects (passive objects that contain the sequential algorithm for solving a problem); and a Collector object, which is an object in charge of storing in parallel the results that arrive from the stage objects that it has connected. (see Figure 6). For implementation details see (Rossainz, et al., 2014).

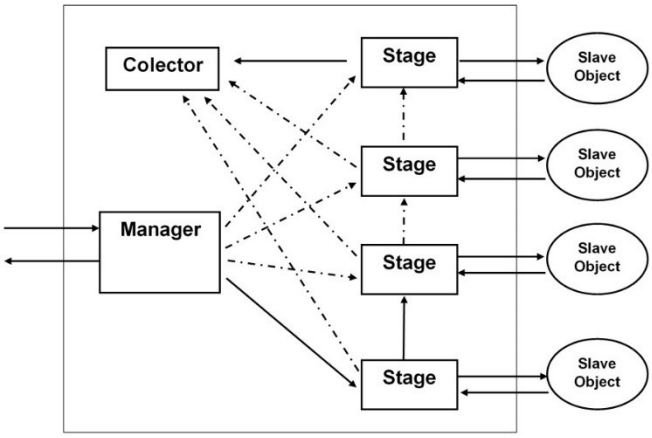

Figure 6. Abstract Model of a HLPC

HLPC Development Methodology:

1. An instance of the manager class is created, that is, one that implements the required parallel behavior according to the following steps:
   a) Initialize the instance with the reference to the slave objects that will be controlled by each stage and the solution algorithm associated with the slave object.
   b) The internal stages are created, and each one is given an association "slave object-solution algorithm", which will be executed by each stage.
2. The user asks the manager to start a calculation by executing the HLPC, which is carried out as follows:
   a) The collector object referring to the request is created.
   b) The input data (without type checking) and the reference to the collector are passed to the stages.
   c) The results are obtained from the collector object.
   d) The collector returns the results to the outside, again without type checking.
3. With this, a manager object has been created and initialized that represents the HLPC itself and execution requests can be dispatched in parallel.

4.3. Flow Graph Interfaces o FGI

A Flow Graph Interface or FGI is a generic parallel algorithm that raises the level of programming abstraction allowing us to express parallelism without having to worry about every low-level detail. It represents an interface between the sequential algorithm that you want to parallelize and its parallel execution once the required communication pattern is represented as an FGI. It is made up of parallel objects called nodes, with two parameters: the information to be processed and the calculation to be performed, which communicate through linking channels called edges (see Figure 7). The most common patterns represented by an FGI are transmission graphs, data flow graphs, and dependency graphs.

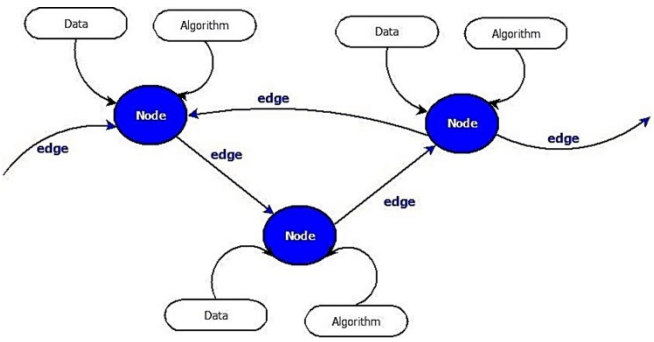

Figure 7. Abstract Model of an FGI

FGI Development Methodology:

1. 1. A graph object, g, is constructed.
2. The nodes that represent the calculations in our Flow Graph are built.
   a) Nodes receive data and process it to send the result to other nodes.
   b) Data processing is carried out through the procedure or operation (algorithm) associated with the node.
3. Once the nodes are created, we connect them using "edges".
   c) The edges represent the dependencies or

communication channels between nodes.

4. Once we complete the construction of the FGI structure, we start its execution through the initial node and wait until the execution of all FGI nodes is completed.

5. Fractals. A case study for PDP, HLPC AND FGI

The following case study was obtained from (Voss, et al., 2019) and involves applying a gamma correction and tint to each image (fractal) in an image vector, writing each result to a file.

A fractal according to (Fernandez, 2018) is a geometric object characterized by presenting a structure that repeats at different scales. In this work, the generic algorithms PDP, HLPC, and FGI were applied to the generation of fractals because they are considered powerful tools that are used in the study of phenomena that occur, for example, in communications, robotics, musical composition, physics, chemistry, geology and even in areas such as economics, mathematics and computing (in image compression), among others (Fernandez, 2018).

In this case study, the elements of a vector are processed by running the corresponding functions to apply gamma correction and tinting, as well as the function to write the resulting image to a file. The first two functions traverse the rows of the image and the elements in each row. The new pixel values are calculated and assigned to the output image. Figure 8 shows the results obtained whose images were generated with the information from (Voss, et al., 2019) in an initially serial loop with repetitions of scales from 2000 to 20000000 on the image vector and then in their parallel versions using the proposed generic parallel algorithms.

5.1. The fractal pipeline and its representation as PDP, HLPC and FGI

The case study in the previous section can be parallelized using the Pipeline pattern that transmits images through a set of stages as shown in Figure 1. The pipeline would be made up of four stages: The one that generates the images, then the one that applies gamma correction, a third stage that applies tint to the image, and one more that writes the resulting image to a file. The graphic models of the design of this pipeline such as Parallel Design Pattern, High-Level Parallel Composition, and Flow Graph Interface are shown in Figure 9, Figure 10, and Figure 11 respectively.

Figure 9 shows the graphic model of the Parallel Design Pattern (PDP) that is developed to implement the Pair-Total design technique through a pipeline and that represents the parallel component used to solve the fractal processing problem. (see Figure 8). The Pipeline in its PDP version is made up of a first stage (stage) that provides the images to be processed so that in the next stage the sequential component is executed, which is the fractal correction operation through the pair-total: problem (initial image) and its solution (corrected image). The next stage of the pipeline will execute the next sequential component, which is the fractal tinting operation, again using the par-total: problem (corrected image) and its solution (tinted image). Finally, the last stage of the pipeline will write the resulting fractal to a file.

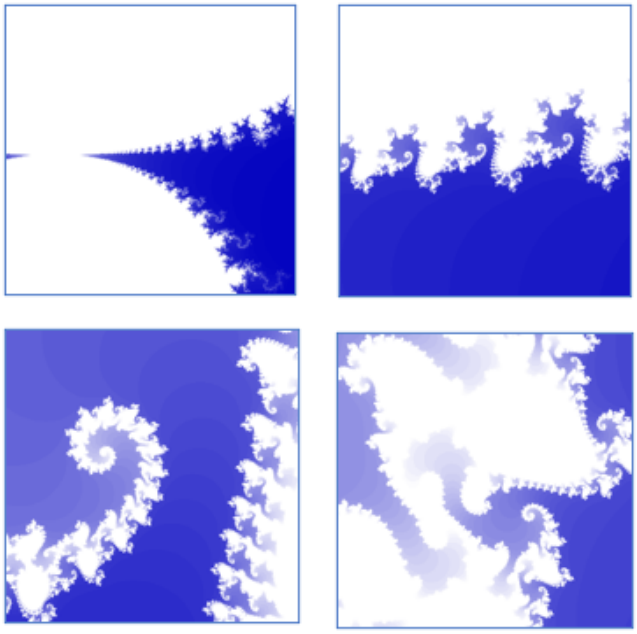

Figure 8. Fractals obtained from (Voss, et al., 2019) after having been applied a gamma correction and tinted with a blue dye, generated by the generic parallel algorithms PDP, HLPC and FGI

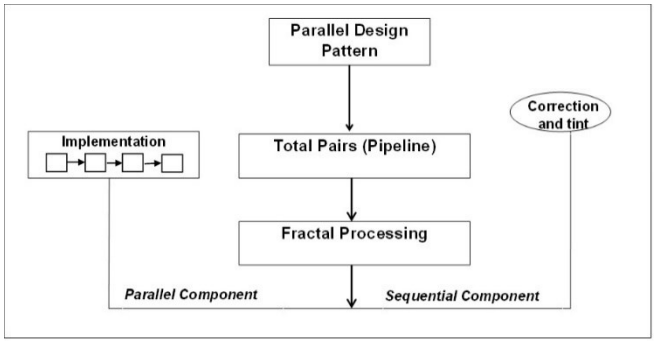

Figure 9. Pair-Total PDP Model for Fractal Processing (correction and tinting)

The type of elements and procedures for dividing the problem in a pipeline is part of the parallel algorithm that depends on the nature of the specific program or model program (see Figure 5 and Figure 9.) This is the main characteristic that makes the PDP simply solve specific problems in parallel; Well, you only have to add to the PDP the sequential problem that is intended to be solved using this technique, in addition to the types of data and procedures referring to said sequential problem associated with the stages of the pipeline (Robbins and Robbins,

1999). The usefulness of the proposal presented here is that different sequential problems, such as the one presented in this work (section 8) and the creation of 3D stereoscopic images (Voss, et al., 2019) to name a few that have to do with image processing, are solved. using the same parallel component, that is, the par-total pipeline designed as PDP.

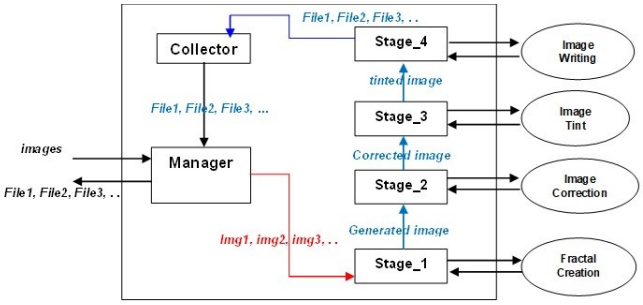

Figure 10. HLPC Pipeline Model for Fractal Processing

Figure 10 shows the graphic model of the parallel processing pipeline technique as a High-Level Parallel Composition applicable to the resolution of the fractal problem already mentioned, in such a way that the HLPC Pipe guarantees the parallelization of the algorithm codes. sequential (creation, correction, tinting, and writing of the fractals) using the Pipeline pattern (Rossainz, et al., 2014). In this HLPC model, the parallel Manager object receives from the user the number of fractals to create, this information is sent to the first stage of the pipeline which executes the algorithm for creating the associated Fractal as a slave object. Once the fractal is created, it is sent to the next stage of the pipeline which executes the associated image correction code as its slave object. The corrected image is sent to the third stage of the pipeline which executes the associated image tinting algorithm as a slave object and once the image has been tinted, it is sent to the fourth and final stage which has the algorithm for writing the image to a file. As the fractals are created, the stages of the pipeline are executed in superposition and once the fractal files have been received by the Collector object, it sends them to the Manager to deliver them to the user. As already mentioned, the execution of the Manager, Pipeline Stages, and Collector objects are carried out in parallel (inter-object parallelism) but internally each of these objects has internal parallelism of its components (intra-object parallelism).

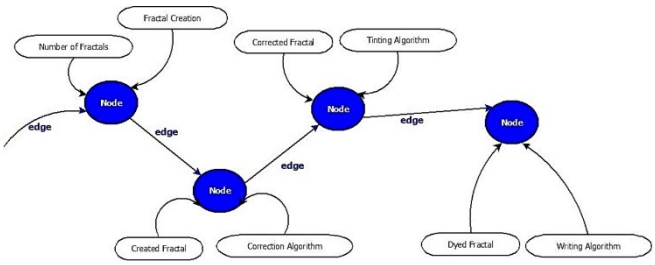

Figure 11. FGI Pipeline Model for Fractal Processing

Finally, Figure 11 shows the graphic model of the pipeline process communication pattern for solving the problem of creating fractals. As in the previous models, in this proposal, we can also superimpose the execution of different stages (nodes) of the process as they are applied to different images. For example, when a first image, img0, is completed on the node that corrects the image, the result is passed to the tinting node, while a new image img2 is created on the first node and passed to the correction node. Similarly, when the next step is performed, img0, which has now passed through the fix and tint nodes, is sent to the writer node. Meanwhile, img1 is sent to the tinting node and a new image, img2, begins to be created at the initial node and is sent for processing at the correction node. In each step, the executions of the nodes are independent of each other, so these calculations can be distributed among different cores or threads, just like the previous proposals. Finally, highlights the importance of expressing fractal processing in its creation, correction, tinting, and writing operations through a pipeline represented as a PDP, an HLPC, and an FGI to show how with these models it is possible to use parallelism driven by message passing.

5.2. Comparison of the performance of the PDP, HLPC and FGI

For the analysis of performance concerning execution times and acceleration or speedup of the PDP, HLPC, and FGI proposed in the generation of the fractals in Figure 8, the implementations of the serial and parallel pipeline of (Voss, et al., 2019) and their executions were carried out in Intel's DevCloud cluster using an 8-core Intel Xeon CPU and up to four NVIDIA GPUs with 5760 cores each with 128GB RAM.

In these three proposals for Generic Parallel Algorithms, the generation of the pipeline follows the same model taken from (Voss, et al., 2019). On the host side (CPU), the scale repetitions are defined that will define the number of fractals to be generated, and the 4 stages of the pipeline are created that will be sent first to a device (GPU), then to two GPUs, then to 3 GPUs and finally to 4 GPUs; having at the end a pipeline stage for each device: creation stage, correction stage, tinting stage and writing stage to the output file. In Host programming, the stages are connected through communication channels that express message-driven parallelism (see Figure 9, Figure 10, and Figure 11). Sending the pipeline stages to the devices (GPU) guarantees the overlapping execution of the different stages of the process as they are applied to different images; That is, in each step of using the pipeline, the execution of its stages is independent of each other, which makes it possible to distribute it among the cores of a GPU and the GPUs used. The graph in Figure 12 shows the sequential and parallel execution times of the PDP-Pipeline, HLPC-Pipeline, and FGI-Pipeline algorithms on the CPU and the GPUs used respectively. In it, we observe that the generic parallel algorithm of the PDP-Pipeline is the

one that takes the longest execution time, unlike the generic parallel algorithm of the FGI-Pipeline which is the one that takes the least execution time to generate the fractals, remaining in the middle part the generic parallel algorithm of the HLPC-Pipeline. This trend remains the same both in the sequential execution of the proposals and in their corresponding parallel executions with one, two, three, and four GPUs respectively. The graph in Figure 12 also shows a clear trend of decreasing the execution time of the proposals as the GPUs are used for their executions (see Table 2).

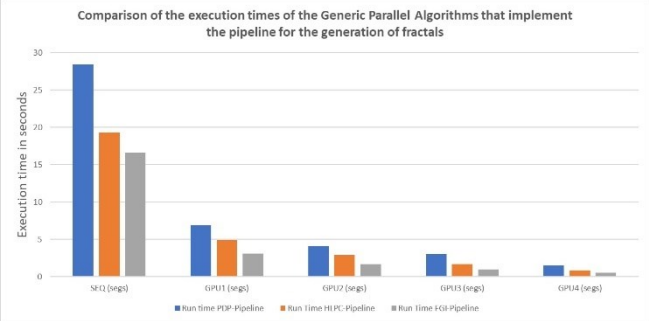

Figure 12. Execution times (in seconds) sequential-CPU and parallel-GPUs of the PDP-Pipeline, HLPC-Pipeline and FGI-Pipeline in the generation of fractals

TABLE 2. Sequential and parallel Execution Times in seconds of the proposed generic parallel algorithms

	SEQ (segs)	GPU1 (segs)	GPU2 (segs)	GPU3 (segs)	GPU4 (segs)
Run time PDP-Pipeline	28.4	6.88	4.1	3.01	1.5
Run Time HLPC-Pipeline	19.33	4.92	2.87	1.67	0.88
Run Time FGI-Pipeline	16.6	3.11	1.68	1.01	0.56

On the other hand, the result of the speedup of the proposed generic parallel algorithms using 1 to 4 GPUs is shown in the graph in Figure 13.

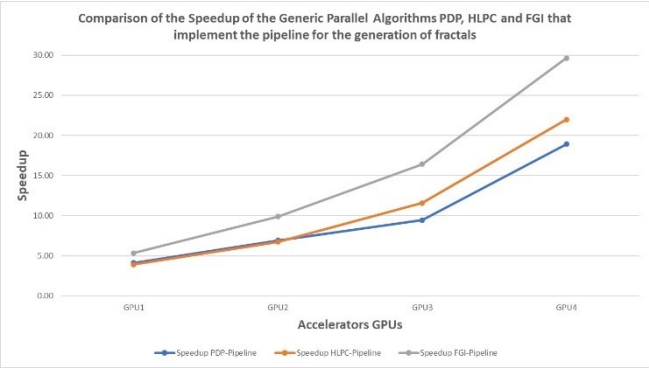

Figure 13. Comparison of the scalability of the speedup or acceleration of the generic parallel algorithms PDP-Pipeline, HLPC-Pipeline and FGI-Pipeline in the generation of fractals with 1,2,3 and 4 GPUs

In it, we observe that the proposal that has the best acceleration and scalability is the FGI-Pipeline which goes from a speedup value of 5.34 with one GPU to 29.64 with four GPUs. In contrast, the proposal that shows the least acceleration as the GPUs scale is the PDP-Pipeline with an initial speedup of 4.13 with one GPU and up to 18.93 with four GPUs. The middle part in terms of acceleration is the HLPC-Pipeline with a speedup factor of 3.93 with one GPU and up to 21.97 with four GPUs. Table 3 shows the rest of the accelerations found.

Table 3. Acceleration of Generic Parallel Algorithms in the generation of fractals

	GPU1	GPU2	GPU3	GPU4
Speedup PDP-Pipeline	4.13	6.93	9.44	18.93
Speedup HLPC-Pipeline	3.93	6.74	11.57	21.97
Speedup FGI-Pipeline	5.34	9.88	16.44	29.64

6. Conclusions

Three proposals for Generic Parallel Algorithms have been presented that represent the communication pattern between processes called Pipeline: The PDP-Pipeline Parallel Design Pattern, the HLPC-Pipeline High Level Parallel Composition, and the FGI-Pipeline Flow Graph Interface; whose implementations were carried out through SYCL programming of Intel's OneApi and the use of Threading Building Blocks or TBB for programming with message passing. These three proposals were used in the case study explained in section 8 where, using a four-stage pipeline, four fractals were created, a gamma correction was applied to them, a dye was applied to them, and they were written to a file.

The objective was to show the usefulness of these three structured parallel programming proposals and have a comparative reference regarding their execution times, accelerations, and scalability in multicore programming for the generation of fractals which are powerful tools that are used in the study of phenomena that occur in different areas of knowledge for problem-solving. This is intended to show how, simply, the novice programmer can make use of these generic parallel algorithms and adapt them to the problem they intend to solve, focusing their efforts solely on the problem and its domain since parallelization is provided by the PDP, HLPC, and FGI that are proposed in this writing. The analysis of the performance of these proposals was carried out through a comparison in both execution times and acceleration and the results are shown in Figure 12, Figure 13, Table 2, and Table 3, respectively which illustrate the similarity in behavior between these three implementations even though they were designed and developed with different models in the design and coding of their algorithms. The performances are considered good given the input and output conditions for the generation of the fractals through the pipeline communication model.

Until now, with the publication of this work, three generic parallel algorithms have been implemented: The Parallel Design Pattern (PDP-Pipeline), the High-Level Parallel Composition (HLPC Pipeline) and the Flow Graph Interface (FGI-Pipeline) that can be used to adapt them to specific problems that can be parallelized with the communication pattern between pipeline-type processes. As future work, we will work on developing, testing and executing applications that process images in bioinformatics to identify DNA patterns and genomes that help find treatments that improve the health of patients with hepatitis and its different derived types. We will use Fractal Geometry techniques so that from Fractals in parallel with the proposed generic algorithms we can model patterns and processes of chain sequences in the identification of DNA and GNOMAS through a pipeline-type architecture.

References

Brinch Hansen (1993). Model Programs for Computational Science: A programming methodology for multicomputers. Concurrency: Practice and Experience. Volume 5, Number 5.

Bockhorst H. (2021). Intel DevCloud for oneApi. Intel Corporation. USA. Recovered from: https://doku.lrz.de/files/17826165/16942048/1/1685941020677/Intel_Devcloud_LRZ.pdf

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

Corradi A., Zambonelli I. (1995). Experiences toward an Object-Oriented Approach to Structured Parallel Programming. DEIS technical report no. DEIS-LIA-95-007.

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

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

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

Danish S.A., Farooqui Z. (2013). Approximate multiple pattern string matching using bit parallelism: a review, International Journal of Computer Applications, Vol. 74, No. 19, pp.47–51.

Ernsting S. and Kuchen H. (2012). 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.

Fernandez-Lara E. (2018). Fractales: bellos y sin embargo útiles. Universidad de Sevilla, España. Recovered from: https://institucional.us.es/blogimus/2018/10/fractales-bellos-y-sin-embargo-utiles/

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

Lavander G.R., Kafura D.G., A Polymorphic Future and First-class Function Type for Concurrent Object-Oriented Programming. Journal of Object-Oriented Systems. Recovered from: http://www.cs.utewxas.edu-users/lavender/papers

Mattson T., Sanders B., and Massingill B. (2004). Patterns for Parallel Programming (First ed.). Addison-Wesley Professional. USA.

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

Reinders Hames, et-al. Data Parallel C++ (2021). Mastering DPC++ for Programming of Heterogeneous System using C++ and SYCL. Apress Open. USA.

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

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

Rossainz M., Pineda I., Dominguez P. (2014). 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.

Theelen B.D., Florescu O., Geilen M.C.W., Huang J., Vander Putten and Voeten J.P.M. (2007). Software/Hardware Engineering with the Parallel Object-Oriented Specification Language. IEEE/ACM International Conference on Formal Methods and Models for Codesign. Pp. 139-148, doi: 10.1109/MEMCOD.2007.371231. Nice, France.

Torquati, M., Aldinucci, M. and Danelutto, M. (2014). FastFlow documentation, Parallel programming in FastFlow, Computer Science Department, University of Pisa, Italy. Recovered from: http://calvados.di.unipi.it/storage/refman/doc/html/index.html

Voss M., Asenjo R., Reinders J. (2019). Pro TBB. C++ Parallel Programming with Threading Building Blocks. Apress Open. USA.

Wilkinson B., Allen M (1999). Parallel Programming. Techniques and Applications Using Net-worked Workstations and Parallel Computers". Prentice-Hall. U.S.A.