

King Saud University

College of Computer and Information Sciences

Computer Science Department



PM Quadtree Gamma Machine

Submitted By

Nora Salama Al-Twairesh

Under the Supervision of

Dr. Ameer Tour

Project Submitted in Partial Fulfillment
for the Degree of Masters in Computer Science

January 2004

Riyadh, Saudi Arabia

Abstract

Parallelism is a promising solution for many complex applications that require high processing power and resources. Quadtree construction is one of the applications that need high processing power, and that would benefit from the use of parallel programs. Gamma, which is a formalism for programming by multiset transformation, was introduced as a specification language without artificial sequentiality. This allows it construct parallel programs implicitly. In this project we use Gamma to formulate the specification of PM quadtrees construction. We then use the specification to develop our PM Quadtree Gamma Machine, that constructs PM quadtrees parallelly.

Table of Contents

Chapter 1: Overview	5
1.1 Introduction	5
1.3 Problem Definition	8
1.2 Project Goals	8
Chapter 2: Gamma and PM Quadrees	9
2.1 Gamma	9
2.1.1 Introduction	9
2.1.2 Why Gamma	10
2.1.3 The Gamma Style of Programming	11
2.1.4 The tropes	12
2.1.5 Extensions to Gamma	14
2.1.5.1 The Chemical Abstract Machine	14
2.1.5.2 Composition operators for Gamma	15
2.1.5.4 High-order Gamma	15
2.1.5.4 Structured Gamma	16
2.1.6 Gamma Impelmentation	16
2.1.6.1 Sequential implementation of Gamma	17
2.1.6.2 Parallel implementation of Gamma	17
<i>Distributed memory parallel machines</i>	18
<i>Shared memory implementations</i>	18
2.2 PM Quadrees	19
2.2.1 Introduction	19
2.2.2 Quadrees Uses	19
2.2.3 Quadrees classes	20
2.2.4 PM Quadtree	21
2.2.5 Previous work in parallel construction of PM quadrees	22
Chapter 3: Project Specification	24
3.1 Introduction	24
3.2 Adaptation of PM Quadrees vs Gamma	24
3.3 The Specification of PM quadrees Construction	25
3.3.1 PM ₁ Quadtree Construction Specification	25
3.3.2 PM ₂ Quadtree Construction Specification	28
3.3.3 PM ₃ Quadtree Construction Specification	29
Chapter 4: PM Quadtree Gamma Machine Implementation	31
4.1 Introduction	31
4.2 Our Approach	31
4.3 Architecture	32
4.4 Program Components	33
4.4.1 Gamma Machine Components	33
4.4.2 Quadtree Components	34
4.5 Program Execution	36
4.5.1 Sequential Implementation	37
4.5.2 Parallel Implementation	37
Chapter 5: Analysis, Future Work, and Conclusion	42
5.1 Introduction	42
5.2 Analysis	42

5.2.1 Testing.....	42
5.2.2 Experimental Results	44
5.2.2 Experimental Results	45
5.3 Future Work.....	46
5.4 Conclusion	48
References.....	49

Chapter 1: Overview

1.1 Introduction

For many years sequential computing has been the dominant in the design of programming languages. This is due to the wide spread of computer architectures that use single processors. The emergence of modern applications in different computing domains such as image processing, ubiquitous computing, geographic information systems, etc., has made the limitations of sequential computing more obvious. These applications demand high processing power which sequential computing cannot provide. Moreover the dramatic progress in hardware technology has lead to the development of parallel systems at different levels, from multiprocessor machines to networks of single processor machines. Using the traditional sequential paradigm to program these machines is not possible. This has paved the way for parallel computing. The real world is inherently parallel so it is natural and straightforward to use parallel computing when programming real world applications. There are many approaches to parallel programming languages, one interesting approach is the Gamma specification programming language.

The Gamma formalism was proposed 17 years ago precisely to capture the intuition of computation as the global evolution of a collection of atomic values interacting freely. The Gamma language was created as a way to abstract only about the problem to be solved leaving aside precepts belonging, for example, to the imperative paradigm, that make the task of creating parallel programs harder. Jean Pierre Banatre and Daniel Le Metayer created Gamma in 1986, based on the multiset parallel rewriting, making the proofs of correctness and derivations of programs easier.

Gamma is a kernel language which can be introduced intuitively through the chemical reaction metaphor. The unique data structure in Gamma is the multiset

which can be seen as a chemical solution. The main feature of a Gamma program is the free interaction between elements of the multiset, which makes the execution model non-deterministic because there is no restriction on how the data elements are to be manipulated by the program rules (in gamma one says that they can react freely), leaving the programs to be naturally and implicitly executed in parallel. The idea is for every subset of the elements in the multiset that fulfill a certain reaction condition a defined action will be performed

Hierarchical data structures are becoming increasingly important representation techniques in the domains of computer graphics, image processing, computational geometry, geographic information systems, and robotics. They are based on the principle of divide and conquer. One such data structure is the quadtree. A Quadtree is a hierarchical data structure used to represent spatial data. The quadtree data structure is widely used in digital image processing and computer graphics for modeling spatial segmentation of images and surfaces. The basic principle of a quadtree is to cover a planar region of interest by a square, then recursively partition squares into smaller squares until each square contains a suitably uniform subset of the input. There are different classes of quadtrees depending on the data that they are used to represent (points, lines, regions, curves, surfaces, volumes). One of these is the PM quadtree, which is used to store Polygonal Maps that represent road maps, utility maps, railway maps, etc., thus the name PM.

The complexity of constructing and manipulating hierarchical data structures requires the use of parallelism when programming applications built on these data structures. This complexity is reduced in parallel programming languages due to the abstract nature of these languages. It is evident that using a high level language that abstracts about the solution of any problem concerned with hierarchical data structures would be very useful. In the case of PM quadtrees, the key issue is that the volume of the data is large. This leads to an interest in parallel processing of such data.

From this aspect we would like to present in this project a specification for constructing PM Quadtrees using the Gamma programming language. We aim here to separate the concerns of architecture and implementation issues from the task of developing a correct solution for the problem of constructing PM quadtrees. Gamma

being an abstract high level language would best serve as a specification language to solve this problem.

This report presents the work conducted on this project. In chapter 2 we give a brief overview of the Gamma language, and PM quadrees. In chapter 3 we formulize the specification that we wrote for constructing PM quadrees using the Gamma language. Chapter 4 presents the implementation of the project, which proves the correctness of the specification we have wrote. The analysis of our work is presented in chapter 5 along with the conclusion and the future work that could be done.

1.3 Problem Definition

The aim of this project is to explore parallel programming by studying the Gamma language. The area of image processing is one of the areas that needs extensive processing power, which will gain much improvement if, we use parallel algorithms and programs instead of sequential ones. The PM (polygonal map) Quadtree data structure is one of the most widely used data structures in this field to represent spatial data. The problem we would like to tackle is how we could write a specification for a Gamma program used to construct PM Quadtrees leading to a parallel construction of PM Quadtrees.

1.2 Project Goals

The goals of this project can be summarized in the following:

- To write the specification of parallel construction of PM Quadtrees, using the Gamma language.
- To implement a working framework that uses the chemical reaction model of Gamma, to construct PM Quadtrees.
- To deploy the implementation on a network of computers that represent a parallel architecture.
- Study the results obtained from the parallel construction of PM Quadtrees, and compare them with the sequential construction.

Chapter 2: Gamma and PM Quadrees

2.1 Gamma

2.1.1 Introduction

Gamma (General Abstract Model for Multiset manipulation) was originally proposed in 1986 as a formalism for the definition of programs without artificial sequentiality.* The basic idea underlying the formalism is to describe computation as a form of chemical reaction on a collection of individual pieces of data.[3]

Gamma involves two different kinds of terms: the programs and the multisets. The programs are described as collections of pairs (Reaction condition, Action) and the execution involves replacing those elements in the multiset satisfying the reaction condition.

$$x_1, \dots, x_n \rightarrow A(x_1, \dots, x_n) \Leftarrow R(x_1, \dots, x_n)$$

Here R is a predicate, called the reaction condition, and A a function called the action. A rewrite takes place if there is a set of elements (x_1, \dots, x_n) from the multiset that satisfy the reaction condition. In that case the elements are removed from the multiset and the elements in A added to the multiset. This process continues until no elements satisfy the reaction condition. The multiset is the basic data structure in Gamma which can be seen as a chemical solution and, unlike an ordinary set, can contain multiple occurrences of the same element. The computation in Gamma is carried out by transforming the original multiset to the final multiset that contains only elements satisfying the reaction condition. The reaction condition can be applied

* artificial sequentiality means sequentiality that is not implied by the logic of the program

to subsets of the elements in the multiset independently, leading to a parallel execution of the program. A simple example illustrates this point. The following is a Gamma program computing the maximum element of a non-empty set.

$$\text{max} : x, y \rightarrow y \Leftarrow x \leq y$$

the reaction condition here is $x \leq y$ it is applied to the elements x and y from the multiset. The action that happens if the reaction condition is true is that x and y are removed from the multiset and replaced by y . Nothing is said about the order of evaluation of the comparisons. If several disjoint pairs of elements satisfy the condition, the reactions can be performed in parallel. The computation in this example will terminate when there is only one element left in the multiset, which is in fact the maximum element.

A Gamma program can be described in terms of the chemical reaction metaphor: the set can be seen as a chemical solution and the computation terminates when a stable state is reached, the stable state here is when no elements in the set satisfy the reaction condition.

2.1.2 Why Gamma

“The basic problem in programming is managing complexity. We cannot address that problem as long as we lump together concerns about the core problem to be solved, the language in which the program is to be written, and the hardware on which the program is to execute. Program development should begin by focusing attention on the problem to be solved and postponing considerations of architecture and language constructs.”[6]

Correctness should be the main concern in program development, then in a second stage efficiency issues can be taken care of, by deriving efficient versions of the program from the correct solution developed before. Consequently we need a high level language to be able to build an abstract version of the program that is free of artificial sequentiality in order to prove the correctness of the programs written.

Gamma was intended to get rid of artificial sequentiality[3], this leads to two important consequences:

- Programs can be described in a very abstract way, and the language is provided a very high level nature. It can be said that it is possible in Gamma to express the idea of the algorithm without any unnecessary linguistic idiosyncrasy. Gamma may be also considered an intermediate language in the program derivation process: where the logical issues are decoupled from the implementation issues. This is convenient in proving the correctness of a program written in Gamma, which can be further refined in a second stage for the sake of efficiency.
- Gamma programs do not have any sequential bias, this leads to the construction of parallel programs naturally (it is even harder to write sequential programs than parallel programs in Gamma).

2.1.3 The Gamma Style of Programming

The development of a Gamma program involves first finding a suitable representation of the data as a multiset then choosing the type of transformation that should be applied to this data.

Elements of the multiset maybe basic or composed values (even multisets). The essential feature of the Gamma programming style is that a data structure is no longer seen as a hierarchy that has to be walked through or decomposed by the program in order to extract atomic values. Atomic values are gathered into one single bag and the computation is the result of their individual interactions[4]. This leads us to the discussion of the “locality principle” in Gamma which entails that individual elements of the multiset may react together and produce new elements in a totally independent way. As a consequence, a reaction condition cannot include any global condition on the multiset such as \forall -properties or properties on the cardinality of the multiset[3]. It can be fairly said that the locality principle models the computation in Gamma as the global results of the successive applications of local, independent, atomic reactions.

Choosing the type of transformation that is applied to the data involves determining the reaction condition and the action that should be performed. There could be more than one pair of (reaction,action) that are all composed together to form the Gamma program. The composition maybe performed in parallel or sequentially depending on the problem that is solved.

Let us now consider the problem of computing the prime number less than a given value n . The basic idea of the algorithm can be described as follows: “start with the set of values from 2 to n and remove from this set any element which is the multiple of another element”. First we need to find the suitable representation of the data in the mutliset. The data here consists of basic of values which are integers from 2 to n so the mutliset consists of the integers $\{2, \dots, n\}$, the representation here is straightforward. Next we need to determine the required transformations that should be applied to the multiset to obtain the solution. So the Gamma program is built as the sequential composition of *iota* which computes the set of values from 2 to n and *rem* which removes the multiples. The program *iota* itself is made of two reactions: the first one splits an interval (x,y) with $x \neq y$ in two parts and the second one replaces any interval (x,x) by the value x .

$$primes(n) = rem(iota(\{2,n\}))$$

$$\begin{aligned}iota &= (x,y) \rightarrow (x, [(x+y)/2]), ([(x+y)/2] + 1, y) \Leftarrow x \neq y \\ & \quad (x,y) \rightarrow x \Leftarrow x = y \\ rem &= x,y \rightarrow y \Leftarrow multiple(x,y)\end{aligned}$$

The first reaction increases the size of the multiset, the second one keeps it constant and the third one makes the multiset shrink. In contrast with the usual sequential or parallel solutions to this problem, the Gamma program proceeds through a collection of atomic actions applying on individual and independent pieces of data.

2.1.4 The tropes

After some experience with Gamma, the creators of the language noticed that a very small number of program schemes were needed to write most applications. We introduce here five of these schemes which are called *tropes* for:

Transmuter

Reducer

Optimiser

Expander

Selector

The tropes are schemata for basic action-reaction pairs, and are defined as follows:

Transmuter :

$$T(C,f) = x \rightarrow f(x) \Leftarrow C(x)$$

the transmuter applies the same operation to all the elements of the multiset until no element satisfies the condition C.

Reducer:

$$R(C,f) = x,y \rightarrow f(x,y) \Leftarrow C(x,y)$$

The reducer reduces the size of the multiset by applying a function to pairs of elements satisfying a given condition C.

Optimiser:

$$O(<,f_1,f_2,S) = x,y \rightarrow f_1(x,y),f_2(x,y) \Leftarrow (f_1(x,y),f_2(x,y))<(x,y) \\ \text{and } S(x,y) \text{ and } S(f_1(x,y),f_2(x,y))$$

The optimizer optimizes the multiset according to a particular criterion (denoted by the ordering <) while preserving the structure of the multiset (described by the relation S). Sorting a list of elements is an example.

Expander:

$$E(C,f_1,f_2) = x,y \rightarrow f_1(x)f_2(x) \Leftarrow C(x)$$

The expander is used to decompose the elements of the multiset into a collection of basic values according to the condition C and by applying f_1 and f_2 to each element.

Selector:

$$S_{i,j}(C) = x_1, \dots, x_i \rightarrow x_j, \dots, x_i \Leftarrow C(x_1, \dots, x_i) \quad (\text{where } 1 < j \leq (i+1))$$

The selector acts as a filter which removes from the multiset all elements satisfying a certain condition C .

We can rewrite the *primes* example from the previous section using combinations of tropes.

$$\text{primes}(n) = \text{rem}(\text{iota}(\{2, n\}))$$

$$\text{iota}(M) = T(C_1, f_1) (E(C_2, f_2, f_3)(M)) \text{ where}$$

$$C_1((x, y)) = (x = y) \quad f_1((x, y)) = x$$

$$C_2((x, y)) = (x \neq y)$$

2.1.5 Extensions to Gamma

Due to the lack of expressivity in Gamma, many extensions have been introduced to enhance Gamma's expressiveness. In this section we review the most important elaborations on the chemical reaction model that have been proposed in the literature. It is important to note that none of these extensions jeopardize the fundamental characteristics of the model which is the expression of computation as "the global results of the successive applications of local, independent, atomic reactions".[3]

2.1.5.1 The Chemical Abstract Machine

The chemical abstract machine (or cham) was proposed by Berry and Boudol in [5] to describe the operational semantics of process calculi. The most important additions to Gamma are the notions of *membrane* and *airlock mechanism*. Membranes are used to encapsulate solutions and to force reactions to occur locally. In terms of multisets, a membrane can be used to introduce multiset of molecules inside a multiset that is to say "to transform a solution into a single molecule"[5]. The airlock mechanism is used to describe communications between an encapsulated solutions

and its environment. The reversible airlock operator \triangleleft extracts an element m of a solution

$$\{m, m_1, \dots, m_n\} \leftrightarrow \{m \triangleleft \{m_1, \dots, m_n\}\}$$

The new molecule can react as a whole while the sub-solution $\{m_1, \dots, m_n\}$ is allowed to continue its internal reactions, so the main role of the airlock is to allow one molecule to be visible from outside the membrane and thus to take part in a reaction in the embedding solution. The cham has inspired a number of other contributions mentioned in [3].

2.1.5.2 Composition operators for Gamma

The basic version of Gamma does not provide any facility for building complex programs from simple ones. It is desirable that a language offers a set of operators for combining programs, to modularity purposes. The operators should also have a collection of algebraic laws to make it possible to reason about programs. A proposal was made in [10] to present a set of operators for Gamma and study their semantics and the corresponding calculus of programs. The two basic operators considered in this paper are the sequential composition $P_1 \circ P_2$ and the parallel composition $P_1 + P_2$. The sequential composition $P_1 \circ P_2$ refers to that the stable multiset reached after the execution of P_2 is given as an argument to P_1 , i.e. the execution of P_1 postponed until the completion of P_2 . On the other hand, the parallel composition $P_1 + P_2$ refers to that the execution of P_1 and P_2 can be done in any order, possibly in parallel. Gamma programs may be built using any of these two operators or a combination of them as required by the problem at hand.

The use of the sequential composition and parallel composition in Gamma creates some semantical problems which are also studied in this paper. The paper defines a set of program refinement and equivalence laws for parallel and sequential composition, by considering the input-output behavior induced by an operational semantics.

2.1.5.4 High-order Gamma

High-order Gamma [13] is an extension of Gamma formalism unifying the program and data syntactic categories by unifying the multiset and the reaction action

pair into a single notion of configuration. A configuration is made of a program and a record of named multisets. As a consequence active programs can be inserted into multisets and reactions can take place simultaneously at different levels. This extension strengthens the expressivity of Gamma to a great degree. Various operators for combining Gamma programs can be defined in high-order Gamma in a natural way, including the sequential and the parallel composition themselves. Thus, these operators do not need to be introduced as primitives in the language. Cham can also be defined in high-order Gamma.

2.1.5.4 Structured Gamma

As illustrated previously the multiset is the basic data structure in Gamma. It has been proven that this may lead to programs which are unnecessarily complex when the programmer needs to encode specific data structures. For example, it was necessary to resort to pairs (index,value) to represent sequences in the sort program. This lack of structuring is detrimental both for reasoning about the programs and for implementing them. The proposal made in [7] is an attempt to solve this problem without jeopardizing the qualities of the language.

The solution proposed in [7] is based on a notion of structured multiset which can be seen as a set of addresses satisfying specific relations and associated with a value. A type is defined in terms of rewrite rules and a structured multiset belongs to a type T if its underlying set of addresses satisfies the invariant expressed by the rewrite system defining T. A structured Gamma program is defined in terms of pairs of a condition and an action which can test/modify the relations on addresses, and test/modify the values associated with addresses.

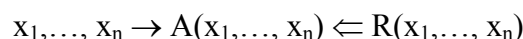
2.1.6 Gamma Impelmentation

As mentioned in previous sections, the philosophy of Gamma is to introduce a clear separation between correctness issues and efficiency issues in programming design. Gamma was introduced as a specification language which does not contain unnecessary sequentiality. As a consequence, a Gamma program is usually far away from a real architecture and designing a reasonably efficient implementation of the language is not straightforward. Several efforts have been made to improve the

implementation of Gamma. Some are sequential implementations and some parallel implementations. In this section we review some of them.

2.1.6.1 Sequential implementation of Gamma

Considering the following reaction-action:



a straightforward implementation can be described by the following[3]:

```
while tuples remain to be processed
do
    choose a tuple  $(x_1, \dots, x_n)$  not yet processed;
    if  $R(x_1, \dots, x_n)$  then
        1. remove  $x_1, \dots, x_n$  from M
        2. replace them by  $A(x_1, \dots, x_n)$ 
    end
```

This very naive implementation puts forward most of the problems which have to be tackled in order to produce a Gamma implementation with a realistic complexity. The hardest problem concerns the construction of all tuples to be checked for reaction. A blind approach to this problem leads to an untractable complexity but a thorough analysis of the possible relationships between the elements of the multiset and the shape of the reaction condition may lead to improvements which highly optimize the execution and produce acceptable performances.

2.1.6.2 Parallel implementation of Gamma

Due to the locality property of Gamma mentioned before, each subset of the multiset that satisfies the reaction condition can be handled simultaneously leading to a naturally parallel program. But managing all this parallelism efficiently is a difficult task, and several complex choices have to be made in order to map Gamma on parallel architectures. Several approaches were introduced for the parallel implementation of Gamma. They differ in the way the memory and reactions between elements are managed. An implementation of Gamma on shared memory machines rather than

distributed machines would be more natural. [14]. However both memory paradigms can be used with an appropriate management scheme.

Distributed memory parallel machines

Two protocols have been proposed,[1][2], for the implementation of Gamma on a network of communication machines. They differ in the way rewritings are controlled:

Centralized control: The elements of the multiset are distributed over the local memories of the processors. A central controller is connected to all the processors and monitors information exchanges.

Distributed control: Information transfers are managed in a fully asynchronous way. The values of the multiset are spread over a chain of m processors. There is no central controller in the system and each processor knows only its two neighbors. The termination detection algorithm is fully distributed over the chain of processors; however, the cost of this detection can be high compared with the cost of the computation itself [1,2]. The results show a good exploitation of the processing power and speedup.

Shared memory implementations:

The Gamma model can also be seen as a shared memory model: the multiset is the unique data structure from which elements are extracted and where elements resulting from the reaction are stored. Shared memory multiprocessors are good candidates for parallel implementations of Gamma. A specific software architecture has been developed in [9] in order to provide an efficient Gamma implementation on a Sequent multiprocessor machine. Several techniques have been experimented in order to improve significantly the overall performances. A kernel operating system has also been developed in order to cope with various traditional problems and in particular with the synchronization required by Gamma (a multiset element cannot participate in more than one reaction at a time).

2.2 PM Quadrees

2.2.1 Introduction

Hierarchical data structures are becoming increasingly important representation techniques in the domains of computer graphics, image processing, computational geometry, geographic information systems, and robotics. They are based on the principle of divide and conquer. One such data structure is the quadtree. The quadtree data structure is widely used in digital image processing and computer graphics for modeling spatial segmentation of images and surfaces. A quadtree is a tree in which each node has four descendants or none. A quadtree may be described as a recursive partition of a region of the plane into axis aligned squares. One square, the root, covers the entire region. A square can be divided into four child squares, by splitting it with horizontal and vertical line segments through its center. The collection of squares then forms a tree, with smaller squares at lower levels of the tree.

Hierarchical data structures are useful because of their ability to focus on the interesting subsets of the data. This focusing results in an efficient representation and in improved execution times. They are compact and depending on the nature of the data, they save space as well as time and facilitate operations such as search.

2.2.2 Quadrees Uses

Quadrees are used to store spatial data. Spatial data consists of spatial objects made up of points, lines, regions, rectangles, surfaces, volumes, and even data of higher dimension which includes time. Examples of spatial data include cities, rivers, roads, counties, states, crop coverages, mountain ranges, parts in a CAD system, etc. Examples of spatial properties include the extent of a given river, or the boundary of a given county, etc. Often it is also desirable to attach non-spatial attribute information such as elevation heights, city names, etc. to the spatial data[]. The main advantage of quadrees is that they are compact and they save space as well as time and facilitate operations such as search.

The quadtree is particularly useful for performing set operations as they form the basis of most complicated queries. For example, to find the names of the roads that pass through the Al-Olyia region, we will need to intersect a region map with a

line map. For a binary image, set-theoretic operations such as union and intersection are quite simple to implement. In particular, the intersection of two quadtrees yields a black node only when the corresponding regions in both quadtrees are black. This operation is performed by simultaneously traversing three quadtrees. The first two trees correspond to the trees being intersected and the third tree represents the result of the operation. If any of the input nodes are white, then the result is white. When corresponding nodes in the input trees are gray, then their sons are recursively processed and a check is made for the mergibility of white leaf nodes. These are some examples of operations on quadtrees.

2.2.3 Quadtrees classes

There are different classes of Quadtrees, but they all share a common property which is that they are based on the principle of recursive decomposition of space. They can be differentiated on the following bases:

- The type of data they are used to represent. (point data, regions, lines, curves, surfaces, volumes).
- The principle guiding the decomposition process, i.e. whether the decomposition is into equal parts on each level (regular polygons) or governed by the input.
- The resolution of the decomposition, which refers to the number of times the decomposition process is applied. The resolution maybe fixed beforehand or governed by properties of the input data.

The classification is usually based on the type of data they are used to represent. From this aspect we have quadtrees that represent regions, called Region Quadtrees[18]. Point data is represented using point quadtrees[8]. The region quadtree can also be used to represent point data, where it will be called a PR quadtree[16]. Several quadtrees were used to represent line data, from these we mention the MX quadtree[12], the edge quadtree[15,20], and the line quadtree[19]. The PM quadtree which we base our work on is also used for line data.

2.2.4 PM Quadtree

The spatial data that is stored in PM quadtrees consists of a collection of lines or vertices that represent polygonal maps which are found in networks of roads, power lines, rail lines etc.

The development of the PM quadtree was done by Samet and Webber in [17], and it is seen to be an adaptation of the PR quadtree. Their goal was to derive a reasonably compact representation of a data structure that satisfies the following three criteria:

(1) It stores polygonal maps without information loss (i.e., it does not suffer a loss of accuracy resulting from digitization).

(2) It is not overly sensitive to the positioning of the map (i.e., shift and rotation operations do not drastically increase the storage requirements of the map).

(3) It can be efficiently manipulated.

To meet this goal, they first had to find a decomposition criterion that repeatedly breaks up the collection of vertices and edges (forming the polygonal map) until they obtain a subset that is sufficiently simple so that it can be organized by some other data structure. Their approach was to develop three closely related quadtree structures PM_1 quadtree, PM_2 quadtree and the PM_3 quadtree. Where starting from the decomposition criteria of the PR quadtree, they weakened the definition of what constitutes a permissible leaf node reaching to the decomposition criteria of the PM_1 quadtree, which is further weakened to reach the decomposition criteria of the PM_2 quadtree which is also further weakened to reach the decomposition criteria of the PM_3 quadtree. By each weakening operation they enable more information to be stored at each leaf node. The decomposition criteria for each of the PM quadtrees is presented in the following:

PM_1 Quadtree

C1: At most one vertex can lie in a region represented by a quadtree leaf.

C2: If a region contains a vertex, then it can contain no line that does not include that vertex.

C3: If a region contains no vertices, then it can contain at most one line.

PM₂ Quadtree

C1: At most one vertex can lie in a region represented by a quadtree leaf.

C2: If a region contains a vertex, then it can contain no line that does not include that vertex.

C3: If a region contains no vertices, then it can contain only lines that meet at a common vertex exterior to the region.

PM₃ Quadtree

C1: At most one vertex can lie in a region represented by a quadtree leaf.

We notice how the decomposition criteria is weakened each time leading to the fact that a permissible PM₁ quadtree leaf node is also a permissible PM₂ quadtree leaf node and likewise a permissible PM₃ quadtree leaf node.

2.2.5 Previous work in parallel construction of PM quadtrees

Samet and Hoel in [11] propose a parallel algorithm for the construction of PM quadtrees. They dealt specifically with the PM₁ quadtree and the PMR quadtree. They present data-parallel primitives that are used in the construction of the PM quadtrees using the scan model of parallel computation. The primitive operations from the scan model that they used are scanwise, elementwise, and permutations operations, these are the low-level primitives. These low-level primitives are used in defining higher-level spatial primitive operations. The higher-level spatial primitive operations are then used in the construction algorithm. The high-level primitive operations that were defined are: *Cloning*, *Unshuffling*, *Duplicate Deletion*, *Node Capacity Check* (for bucket PMR quadtree), *Should a PM₁ Quadtree Node Split*, *Splitting a Quadtree Node*.

The PM₁ quadtree construction algorithm is as follows:

Start by assigning all the lines to the first node (i.e. the root node).

Using the primitive *Should a PM₁ Quadtree Node Split* the root node is checked for splitting

If the root node should be split the primitive *Splitting a Quadtree Node* is used.

The new nodes will go through the same iteration:

Each node is first checked for splitting using: *Should a PM₁ Quadtree Node Split*.

And then if needed split the node using: *Splitting a Quadtree Node*.

The algorithm was implemented in C on a minimally configured Thinking machines CM-5 with 32 processors containing 1 GB of main memory.

It was found that for n line segments, the data-parallel PM₁ quadtree construction operation takes $O(\log n)$ time, where each of the $O(\log n)$ subdivision(splitting) stages requires $O(1)$ computations (a constant number of scans, clonings, and un-shuffles).

Chapter 3: Project Specification

3.1 Introduction

In this chapter we will try to apply the principle of the Gamma reaction-action model on the PM quadtree construction and show how they adapt. Thus, formulating the specification of the program that constructs a PM quadtree using Gamma.

3.2 Adaptation of PM Quadtrees vs Gamma

The construction of PM quadtrees can be carried out by applying the decomposition criteria to each node, to determine if it needs splitting or not, then splitting the nodes that do not satisfy the criteria. Looking back at the structure of Gamma programs, we know that they consist of a reaction condition that is tested, and an action that is performed.

The construction of PM quadtrees consists exactly of a condition that is checked and an action that is performed when the condition is true. The reaction condition here is the negation of the decomposition criteria and the action is the splitting of the node. Of course the data we are dealing with here is composed of the quadtree nodes and the input lines that form the polygonal map. So in the Gamma program we will have a multiset of nodes and a multiset of lines that belong to a node. The program will start with one node in the multiset of nodes, which is the root node, and the input lines in the multiset of lines. The initial multiset of lines is assigned to the root node, then the reaction condition will be tested against the root node, if it is true the action that will be performed is splitting this node to four nodes and the root node will be replaced by the new resulting nodes in the nodes multiset. The program continues its execution by performing the reaction-action pair on the new nodes in the multiset, which can be done in parallel, since each node's reaction-action is

independent than the other nodes. The program execution will continue since new nodes will be added to the multiset. The program terminates when all the resulting nodes cannot be split further because they satisfy the decomposition criteria or in the worst case we reach the maximal resolution of quadtrees which is a 1×1 node, which indeed represents a vertex and thus also satisfies the decomposition criteria.

3.3 The Specification of PM quadtrees Construction

We will write a specification for each of the three types of PM quadtrees, PM1, PM2, PM3 according to the specific criteria for each one of them.

Each node (square) N is represented by the following:

1. $(X0, Y0)$: the coordinates of the bottom left corner.
2. i : the power of 2 that represent the length of the square's edge.
3. $M(L)$: multiset of the lines that are in the node.

A line L is represented by

1. Two endpoints $(X1, Y1)$, $(X2, Y2)$.
2. Two labels defining the region on the left of the line, and the region on the right of the line RL,RR.

3.3.1 PM1 Quadtree Construction Specification

According to the definition of the PM1 quadtree, the following criteria must be met:

C1: at most one vertex can lie in a region represented by a quadtree leaf.

C2: if a region contains a vertex, then it can contain no line that does not include that vertex.

C3: if a region contains no vertices, then it can contain at most one line.

First we will need two predicates to represent some conditions we need to test:

inside: which checks if a point P is inside a node N .

through: which checks if a line L passes through a node N .

Now we define the reaction that will cause a node to split, it will be the negation of the decomposition criteria. So a node will be split if:

- it has more than one vertex in it,
- or if several lines that pass through it, and do not share a common vertex that is inside this node
- or if it does not have any vertex in it but more than one line pass through it

To check if a node has more than one vertex, we will use the inside predicate as following:

$$\textit{inside}(N, N.M(Lj(X1, Y1)) \textit{ AND } \textit{inside}(N, N.M(Lj(X2, Y2))$$

To check if several lines pass through the node and do not share a common vertex that is inside this node:

$$\begin{aligned} &(\textit{inside}(N, N.M(Lj(X1, Y1)) \textit{ AND } \textit{inside}(N, N.M(Lk(X1, Y1)) \\ &\textit{ AND } M(Lj(X1, Y1) \langle \rangle M(Lk(X1, Y1)) \\ &\textit{ OR } (\textit{inside}(N, N.M(Lj(X1, Y1)) \textit{ AND } \textit{inside}(N, N.M(Lk(X2, Y2)) \\ &\textit{ AND } M(Lj(X1, Y1) \langle \rangle M(Lk(X2, Y2))) \end{aligned}$$

To check if a node does not contain any vertex inside, but more than one line pass through it:

$$\begin{aligned} &\textit{NOT} (\textit{inside}(N, N.M(Lj(X1, Y1)) \textit{ OR } \textit{inside}(N, N.M(Lk(X1, Y1)) \\ &\textit{ OR } \textit{inside}(N, N.M(Lk(X2, Y2))) \\ &\textit{ AND } \textit{through}(N, M(Lj)) \textit{ AND } \textit{through}(N, M(Lk)) \end{aligned}$$

These are the conditions that will form the reaction.

If the reaction is true then the node needs to split and this will be the action. Splitting the node is the process of decomposing it into four equal squares, we will call them NW, NE, SW, SE, to represent the upper-left square, the upper-right square, the lower-left square, and the lower-right square respectively.

For each new node we need to know the bottom left corner, the length of the nodes' edge, and the lines that this node contains.

The bottom left corner for each new node can be computed with reference to the original node's bottom left corner, and the square's edge length. For each new node we have:

NW: same x-coordinate of the original node, and the y-coordinate will be the original node's y-coordinate plus half of the original node's edge length.

NE: the x-coordinate will be the original node's x-coordinate plus half of the original node's edge length, and the y-coordinate will be the original node's y-coordinate plus half of the original node's edge length.

SW: same x-coordinate of the original node, and same y-coordinate of the original node.

SE: the x-coordinate will be the original node's x-coordinate plus half of the original node's edge length, and the y-coordinate will be the same y-coordinate of the original node

The edge length of each square will be half the original node's edge length. Knowing that the original node's edge length is 2^i , the new nodes' edge length can be computed as follows:

$$2^i / 2 = 2^{i-1}$$

The lines that each new node contains can be computed by inspecting the multiset of lines that belong to the original node, and checking if they pass through the new node or not. We can define a program *Lines* that determines these nodes:

$Lines(N,M) = T(RI, AI)(N,M)$ where

$RI(N,M(L_j)) = through(N,M(L_j))$

$AI(N,M(L_j)) = M(L_j)$

Here we see that if a line L_j from the original node's multiset of lines M passes through the node N , this line is added to the node N 's multiset of lines.

After inspecting the problem in detail, we can now write the specification of the complete program that constructs a PM_1 quadtree. The program takes as input:

- The list of lines that compose the polygonal map, represented with their endpoints and left and right regions. This list is assigned to the root node's multiset of lines.
- The root node N , that has a bottom-left corner as $(0,0)$, and its square's edge length can be set to the nearest power of 2 that is greater than the largest x or y coordinate of the endpoints of the input lines.

This is the program:

$PM1Quadtree(N)=T(RI,AI)(N)$ where

$$\begin{aligned}
 RI(N) = & (inside(N, N.M(Lj(X1,Y1)) \text{ AND } inside(N, N.M(Lj(X2,Y2))) \\
 & \text{ OR } (inside(N, N.M(Lj(X1,Y1)) \text{ AND } inside(N, N.M(Lk(X1,Y1))) \\
 & \text{ AND } M(Lj(X1,Y1)) \neq M(Lk(X1,Y1))) \\
 & \text{ OR } (inside(N, N.M(Lj(X1,Y1)) \text{ AND } inside(N, N.M(Lk(X2,Y2))) \\
 & \text{ AND } M(Lj(X1,Y1)) \neq M(Lk(X2,Y2))) \\
 & \text{ OR } (\text{Not } (inside(N, N.M(Lj(X1,Y1)) \text{ OR } (N, N.M(Lk(X1,Y1))) \\
 & \text{ OR } (N, N.M(Lk(X2,Y2)))) \\
 & \text{ AND } through(N, M(Lj)) \text{ AND } through(N, M(Lk))) \\
 AI(N) = & NNW(N.X, N.Y+2^{N.i-1}, N.i-1, lines(NNW, N.M)), \\
 & NNE(N.X+2^{N.i-1}, N.Y+2^{N.i-1}, N.i-1, lines(NNE, N.M)), \\
 & NSW(N.X, N.Y, N.i-1, lines(NSW, N.M)), \\
 & NSE(N.X+2^{N.i-1}, N.Y, N.i-1, lines(NSE, N.M))
 \end{aligned}$$

3.3.2 PM₂ Quadtree Construction Specification

According to the definition of the PM₂ quadtree, the following criteria must be met:

- C1:** At most one vertex can lie in a region represented by a quadtree leaf.
- C2:** If a region contains a vertex, then it can contain no line that does not include that vertex.
- C3:** If a region contains no vertices, then it can contain only lines that meet at a common vertex exterior to the region.

We see that the criteria are almost the same criteria of the PM₁ quadtree, except for the last one C3. And the splitting process will be the same. So by just

modifying the condition that tested the C3 criterion in the PM1 quadtree specification, we can rewrite the specification to be for the construction of the PM2 quadtree.

The modification here will be to check that the region that contains no vertices contains lines that do not share a common vertex exterior to the region.

It could be written as follows:

*Not (inside(N, N.M(Lj(X1, Y1)) OR inside(N, N.M(Lk(X1, Y1))
OR inside(N, N.M(Lk(X2, Y2))
AND through (N,M(Lj)) AND through(N,M(Lk))
AND M(Lj(X1, Y1) <> M(Lk(X1, Y1))
AND M(Lj(X1, Y1) <> M(Lk(X2, Y2))*

And the specification of the PM2 quadtree construction will be as follows:

PM2Quadtree(N)=T(R1,A1)(N) where

*R1(N)= (inside (N, N.M(Lj(X1, Y1)) AND inside(N,N.M(Lj(X2, Y2))
OR (inside (N, N.M(Lj(X1, Y1)) AND inside(N,N.M(Lk(X1, Y1))
AND M(Lj(X1, Y1) <> M(Lk(X1, Y1))
OR (inside (N, N.M(Lj(X1, Y1)) AND inside(N,N.M(Lk(X2, Y2))
AND M(Lj(X1, Y1) <> M(Lk(X2, Y2))
OR (Not (inside(N, N.M(Lj(X1, Y1)) OR (N, N.M(Lk(X1, Y1))
OR (N, N.M(Lk(X2, Y2))
AND through (N,M(Lj)) AND through(N,M(Lk))
AND M(Lj(X1, Y1) <> M(Lk(X1, Y1))
AND M(Lj(X1, Y1) <> M(Lk(X2, Y2)))*

*A1(N)= NNW(N.X,N.Y+2^N.i-1,N.i-1,lines(NNW,N.M)),
NNE(N.X+2^N.i-1,N.Y+2^N.i-1,N.i-1,lines(NNE,N.M)),
NSW(N.X,N.,N.i-1,lines(NSW,N.M)),
NSE(N.X+2^N.i-1, N.Y,N.i-1,lines(NSE,N.M))*

3.3.3 PM₃ Quadtree Construction Specification

According to the definition of the PM3 quadtree, the following criterion must be met:

C1: At most one vertex can lie in a region represented by a quadtree leaf.

So if a node contains more than one vertex it has to be split, we can write this condition as follows:

*(inside (N, N.M(Lj(X1, Y1) AND inside(N,N.M(Lj(X2, Y2))
OR (inside (N, N.M(Lj(X1, Y1) AND inside(N,N.M(Lk(X1, Y1)
AND M(Lj(X1, Y1) <> M(Lk(X1, Y1))
OR(inside (N, N.M(Lj(X1, Y1) AND inside(N,N.M(Lk(X2, Y2)
AND M(Lj(X1, Y1) <> M(Lk(X2, Y2))*

And the specification of the PM3 quadtree construction will be as follows:

PM3Quadtree(N)=T(R1,A1)(N) where

*R1(N)= (inside (N, N.M(Lj(X1, Y1) AND inside(N,N.M(Lj(X2, Y2))
OR (inside (N, N.M(Lj(X1, Y1) AND inside(N,N.M(Lk(X1, Y1)
AND M(Lj(X1, Y1) <> M(Lk(X1, Y1))
OR (inside (N, N.M(Lj(X1, Y1) AND inside(N,N.M(Lk(X2, Y2)
AND M(Lj(X1, Y1) <> M(Lk(X2, Y2)))
A1(N)= NNW(N.X,N.Y+2^{N.i-1},N.i-1,lines(NNW,N.M))
NNE(N.X+2^{N.i-1},N.Y+2^{N.i-1},N.i-1,lines(NNE,N.M))
NSW(N.X,N.,N.i-1,lines(NSW,N.M))
NSE(N.X+2^{N.i-1}, N.Y,N.i-1,lines(NSE,N.M))*

Chapter 4: PM Quadtree Gamma Machine Implementation

4.1 Introduction

After formalizing the specification of the PM Quadtrees construction using Gamma in chapter 3, we now present the implementation scheme we followed to produce our PM Quadtree Gamma Machine. We describe our approach in section 4.2, and the architecture we used in section 4.3. In section 4.4 the various components in our system are presented. And in section 4.5 we describe the program execution of our Gamma Machine.

4.2 Our Approach

As we saw in chapter 2, Gamma can be implemented either using a sequential method or a parallel method. The sequential method is straightforward, it contains a loop that loops through the elements of the multiset, testing them against the reaction condition. This sequential implementation does not serve any usefulness in exhibiting the implicit parallelism nature of the Gamma programs. However, it is useful in testing the correctness of the developed specification. Also it can play as a good candidate when comparing it with the parallel version of the same Gamma program to demonstrate the performance of parallel programs.

From this aspect we chose to provide both a sequential and parallel version of our program.

4.3 Architecture

The sequential version was executed on one machine since there is no distribution of the work, and it is all done on one computer.

For the parallel version we chose a distributed memory parallel implementation that was executed on a network of computers. In contrast to the protocols used in [1] and [2] that use a Centralized Control or Distributed control, we chose a hybrid solution of the two protocols. First we used the notion of a *Worker* that represents an entity that carries out a reaction-action pair on one element, with many *Workers* executing at the same time we can carry out several reaction-actions in parallel. However we still need some controller that coordinates the work between the *Workers*, for which we used a *Worker Manager* to present it. The communication between the *Workers* and between the *Workers* and the *Worker Manager* is done through message passing.

When deciding on the type of coordination that is carried out by the *Worker Manager*, we confronted two alternatives. Either let the *Worker Manager* work as a centralized controller which is responsible for distributing the work over the *Workers* and receiving the results from the *Workers*, or just let the *Worker Manager* provide the *Workers* that have work to distribute, with the addresses of free workers and let the *Workers* communicate among themselves, which kind of distributes (decentralizes) the task of distributing the work over the *Workers*. Considering the amount of messages that the *Worker Manager* would be receiving in the first approach and the coordination it needs to carry out, we reached the fact that the *Worker Manager* would turn into a bottleneck which would slow the overall performance of the program. So to minimize the overhead of these messages, we chose the second approach where the *Workers* contact the *Worker Manager* only to get the addresses of other free workers to give them the new work that needs to be done. The following figure demonstrates the configuration of the *Worker Manager* and the *Workers*:

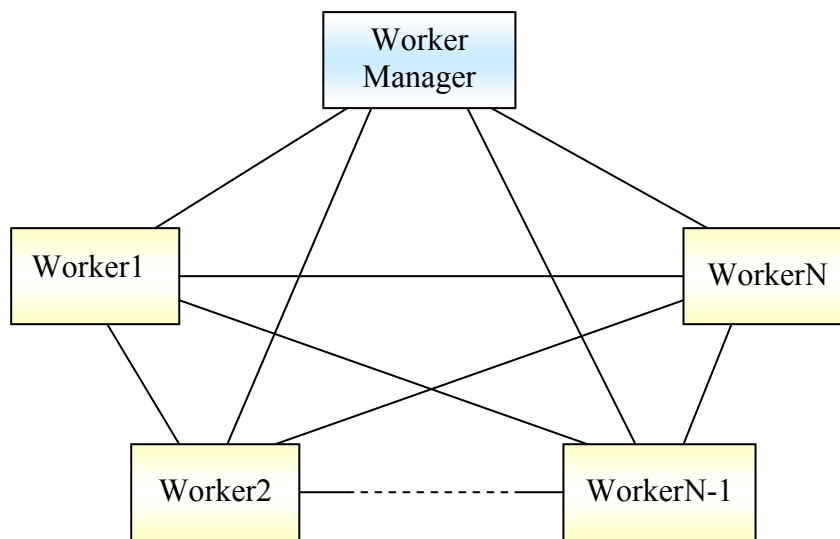


Figure 1 Worker Manager and Workers

4.4 Program Components

The program consists of several components that we chose to decompose into two packages. One package contains the components of the Gamma Machine, and the second package contains the components of the PM Quadtree. The class diagram of these components is given next followed by a brief description of each component.

4.4.1 Gamma Machine Components

The following are the classes in the Gamma Machine package:

GammaProgram: this class contains the definition of the reaction and action of the Gamma program. They were written depending on the specification that we proposed in chapter 3. The class receives as input a node that it will perform the reaction-action on.

Worker Manager: the coordinator that is responsible for managing the workers in the system. It provides the Workers with the addresses of the free workers so they can communicate. It maintains the list of free workers in a queue. There is only one copy of this component running on one computer.

Worker: this is where the gamma program is executed i.e. the reaction-actions are performed. There are several copies of this component running on several computers all waiting for a new element to work on.

Message: this is an abstract class of the component that represents the messages that are transferred between the *Workers*, and between the *Workers* and the *Worker Manager*. There are several messages that extend this class. These are:

- *NewNode*: which represents a new node that hasn't been processed yet i.e. we didn't perform the reaction-action on it yet. This message is usually sent from a *Worker* to another *Worker*.
- *RequestAddress*: which is a message that is sent from a *Worker* to the *Worker Manager* to request addresses of free workers which it can communicate with.
- *WorkersAddresses*: which is the reply that the *Worker Manager* sends to a *Worker* that has requested addresses of free workers.
- *IamFree*: which is a message that a *Worker* sends to the *Worker Manager*, to inform it that it is done with its work and it is free now. The *Worker Manager* will add this *Worker* to list of free workers.

4.4.2 Quadtree Components

. As stated in chapter 3 a quadtree is represented with a set of nodes and a set of lines in each node. The following are the classes of the Quadtree package:

Node: which represents a quadtree node that is defined by its bottom-left corner point, the power of 2 that equals its square edge length, and the set of lines that are in or pass through the node.

Line: which represents a line that is defined by its two endpoints. This class is used to represent the lines in the node.

Point: which represents a point that is defined by its X and Y coordinates. This class is used to represent the endpoints of a line and the bottom-left corner point in the node.

The implementation was done for the PM1 quadtree.

The following is the class diagram of the PM Quadtree Gamma Machine.

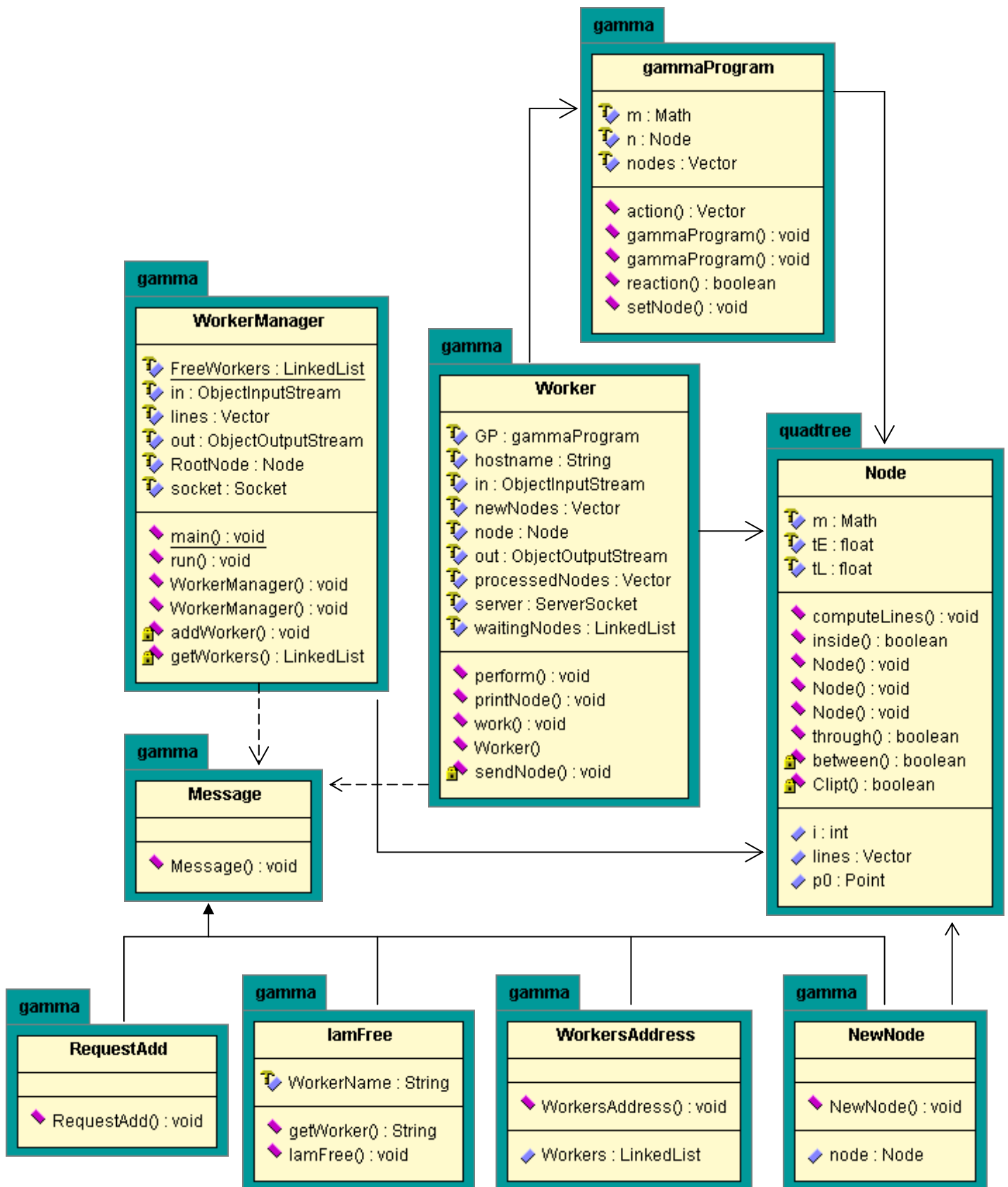


Figure 2 PM Quadtree Gamma Machine Class diagram

4.5 Program Execution

4.5.1 Sequential Implementation

In the sequential implementation, we use the quadtree package for representing the quadtrees, and from the gamma machine package we only use the *GammaProgram* class that contains the reaction-action.

The program reads the input lines and creates a new node that represents the root node with a bottom left corner of (0,0). Then it assigns the input lines to the root node. It creates a new instance of the *GammaProgram* class called *GP* and assigns to it the root node. The program maintains two sets. The first set contains the new nodes that result from every action performed, we call this set *newNodes*. The other set contains the leaf nodes that do not satisfy the reaction, we call this set *leafNodes*. The computation in the program is carried out inside a loop that checks each node for the reaction. The following is the loop:

```
newNodes.add(node); //node is the root node

for (int j=0;j<newNodes.size();j++)
{
    GP.setNode((Node)newNodes.get(j));

    if (GP.reaction())
    {
        nodes=GP.action(); //action() returns the new nodes

        for (int i=0;i<nodes.size();i++)
            newnodes.add(nodes.get(i));

        nodes.removeAllElements();
    }
    else
        leafNodes.add(newnodes.get(j));
}
```

The loop will continue its execution until there are no more nodes to process, i.e. there are no more nodes that satisfy the reaction condition.

4.5.2 Parallel Implementation

In the parallel implementation the program execution will be carried out as follows: First we will have to run the *Worker Manager* and the set of *Workers* on the computers that we have. Execution starts at the *Worker Manager*, it first reads the set

of lines that represent the polygonal map we have to build the quadtree for and the list of *Workers*, workers are represented by their host names. We initialize the first node to having a bottom left corner point of (0,0) and assign to it, the original set of input lines as its multiset of lines. The *Worker Manager* will pack this node in a *NewNode* message and send it to one of the *Workers*. Then it will sit waiting for requests of free workers addresses.

The *Worker* that received the root node will perform the reaction-action on the node it is assigned, then according to the result of the reaction it will do one of the following:

If the reaction condition is true, the node will be split into four new nodes. The *Worker* will send a *RequestAddress* message to request the addresses of three free workers from the *Worker Manager*. Then the *Worker Manager* will send back to the *Worker* the addresses of three free workers in a *WorkersAddress* message. The *Worker* will contact these three free workers and send three of the new nodes to them packed in *NewNode* messages, and the fourth node it will start processing it itself. There is a possibility that there are no free workers or there are less than three free workers, in this case the *Worker* that has new nodes that need processing will add the nodes that he didn't find workers to work on them, to a list of waiting nodes that it maintains, so that it will process them himself later.

If the reaction condition is false, meaning that the node is a leaf node, the *Worker* will save the node in the set of processed nodes it has. Then the worker will first check its list of waiting nodes, if the list is empty i.e. there are no nodes that need processing then the *Worker* will send a *IamFree* message to the *Worker Manager* and sit waiting for a new node to arrive. The *Worker Manager* will add this worker's address to the queue of free workers. If the waiting nodes list has nodes, then the *Worker* will start processing one of these nodes.

The following shows how this is done in the *Worker*:

```
newMessage=(Message)in.readObject();//read an incoming message

    if (newMessage instanceof NewNode)
    {
        nodeMessage=(NewNode)newMessage;
        node=nodeMessage.getNode();
```

```

GP.setNode(node);
if (GP.reaction())
{
    newNodes.removeAllElements();
    newNodes=GP.action();

    out.writeObject(reqmsg);
    out.flush();
    workers=(WorkersAddress)in.readObject();

    w.clear();
    w=workers.getWorkers();

// do 3 times:
//if there is an address sendNode
for (int i=0;i<3;i++)
{
    n=(Node)newNodes.get(i+1);
    if (!w.isEmpty()){
        worker=(String)w.removeFirst();
        sendNode(worker,n);

    }else waitingNodes.addFirst(n);//no worker free, so add
        // node to waiting nodes
    }
//set node to one of the new nodes
node=(Node)newNodes.get(0);
GP.setNode(node);
}
else //reaction false
{
    processedNodes.add(node); //add node to processed nodes
//first check if there are any unprocessed nodes waiting in
//the queue
//send to WM i am free msg
if (waitingNodes.isEmpty()){
    out.writeObject(freemsg);

}else
{
    node=(Node)waitingNodes.removeFirst(); //work on
        //waiting nodes
    GP.setNode(node);
}
}

```

The program terminates when all the nodes have been processed i.e. there are no more elements that satisfy the reaction condition. This can be detected when all the workers are free. So by inspecting the queue of workers maintained by the *Worker Manager*, we find that when all the workers are in the queue then we can conclude that there is no processing going on and the quadtree has been successfully built.

The execution can be more clarified through an example. If we had the following simple polygon, and we wanted to construct its quadtree.

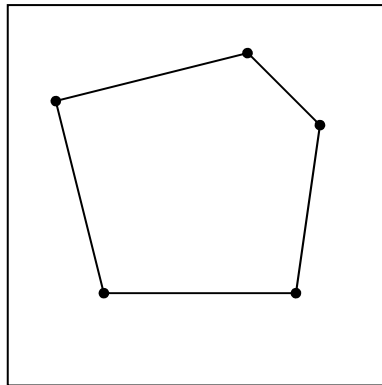


Figure 3 polygon example

Assuming we had 5 computers, one of them is the *Worker Manager* and the rest are *Workers*. The program would first start by reading in the coordinates of set of lines representing the polygon, and the names of the *Workers* computers (we can assume the names to be W1, W2, W3, W4). It would create a new node to represent the root node and assign to it the set of input lines. Then it would pack the root node in a *NewNode* message and send it to one of the workers in the list of workers. The *Worker* that receives the *NewNode* message lets say W1 would unpack the message and get the node object, it would perform on it the reaction, as we can see that the node violates the first decomposition criterion, because it has more than one vertex, so the node has to be split. We perform the action on this node and we get the following new nodes:

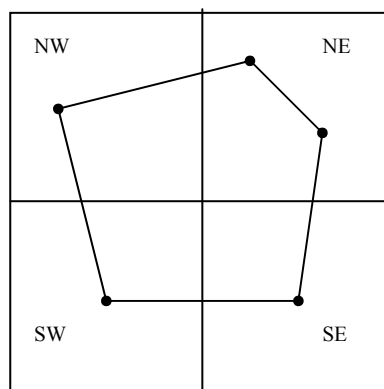


Figure 4 polygon example quadtree after first split of nodes

The Worker now contacts the Worker Manager to request the addresses of three free workers. Of course since only one Worker is busy and we have four workers, there will be three free workers W2, W3, W4. The Worker Manager sends the addresses of the free workers to the Worker. The Worker will send three of the nodes to the workers. Lets say that it sends the NW node to W2, the NE node to W3, the SE node to W4, and it starts working on the SW node itself. The SW node does not violate any of the decomposition criteria so it does not need any splitting, thus W1 will add this node to its set of processed nodes and send a *IamFree* message to the *Worker Manager*. The NW node also does not violate the decomposition criteria so it does not need any splitting, thus W2 will add this node to its set of processed nodes and send a *IamFree* message to the *Worker Manager*. The SE node does not violate the decomposition criteria so it does not need any splitting, thus W4 will add this node to its set of processed nodes and send a *IamFree* message to the *Worker Manager*. The NE node does violate the decomposition criteria because it contains more than one vertex so it needs to be split, and will we have the following quadtree:

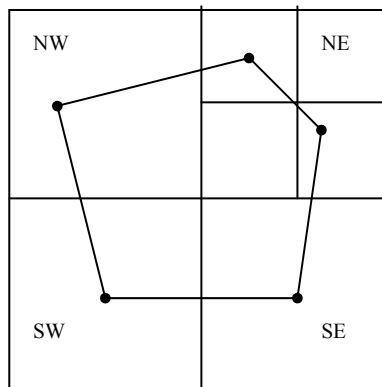


Figure 5 polygon example quadtree

W3 will contact the Worker Manager to get the addresses of free workers. And the same process we mentioned in the first case will be carried out again, of course the resulting nodes do not need any more splitting so this will be the result of the quadtree construction.

Chapter 5: Analysis, Future Work, and Conclusion

5.1 Introduction

Our aim was to write the specification of the PM quadtrees using Gamma, and this is what we presented in chapter 3. We chose Gamma because its abstract nature provides implicit parallelism, and the construction of quadtrees would be best done in parallel. Gamma can be implemented both sequentially and parallelly, we implemented both methods for the purpose of comparing between them in terms of execution times. We conducted several tests which we present in this chapter along with the results of these tests. During our work on this project we found several topics that relate to our work that we would like to suggest as future work. In this chapter we also present the conclusion of our work.

5.2 Analysis

5.2.1 Testing

We performed the testing on Pentium III computers with a speed of 1.0 GHz and 128 MB Ram. The computers were connected using a switching hub. We used as input data two varying sets of lines (figure -39 lines, figure -113 lines).

The following are the test cases we performed:

	implementation version	number of lines	number of computers
test case 1	sequential	39	1
test case 2	parallel	39	17
test case 3	sequential	113	1

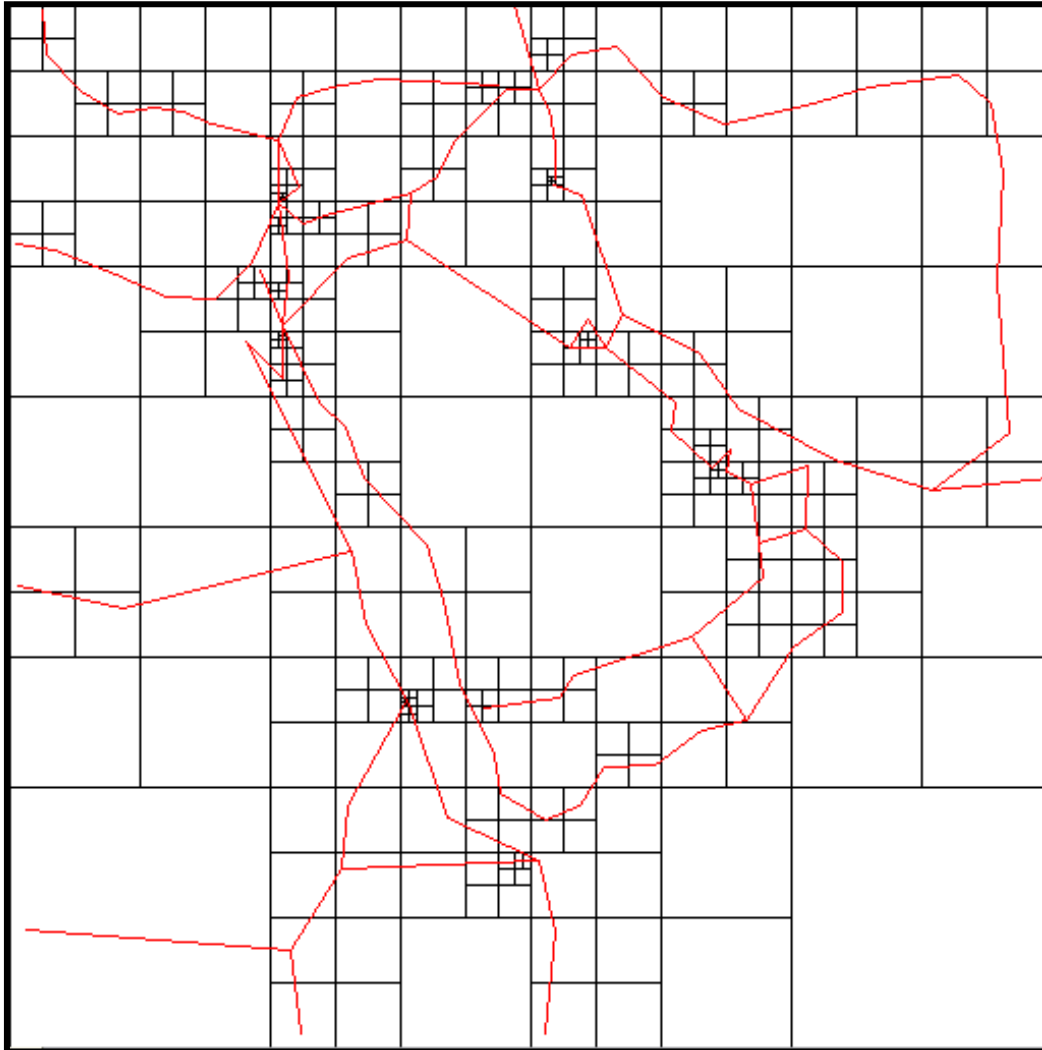


Figure 7 Second set of input lines: 113 lines

5.2.2 Experimental Results

In the following table we present the statistics we collected for each of the test cases:

	implementation version	number of lines	number of computers	execution time
test case 1	sequential	39	1	9.423
test case 2	parallel	39	17	6.429
test case 3	sequential	113	1	27.019
test case 4	parallel	113	17	7.641
test case 5	parallel	113	16	7.952
test case 6	parallel	113	11	5.598
test case 7	parallel	113	9	6.690
test case 8	parallel	113	5	8.452
test case 9	parallel	113	4	11.236
test case 10	parallel	113	3	18.767
test case 11	parallel	113	2	26.088

We can see that the parallel version performed better in comparison to the sequential version in all the test cases. In the best case (test case 6), we find that the execution time was reduced by 80% than that of the sequential version in test case 3. which is a very good result. Also when comparing the parallel versions with different number of computers, we can see that 11 computers performed better than 17 computers, so we can conclude that at a certain stage adding more computers would not give the best execution time, the reason for this could be that the message passing time would be a big dominant in the case of a large number of computers. So the number of computers chosen should be appropriate and depending on the number of input lines. (a study of the relation between the number of computers needed and the number of input lines would be a good suggestion for future work).

In the following graph we plot the relationship between the execution time and the number of computers that were obtained from the testing performed on the input set on 113 lines.

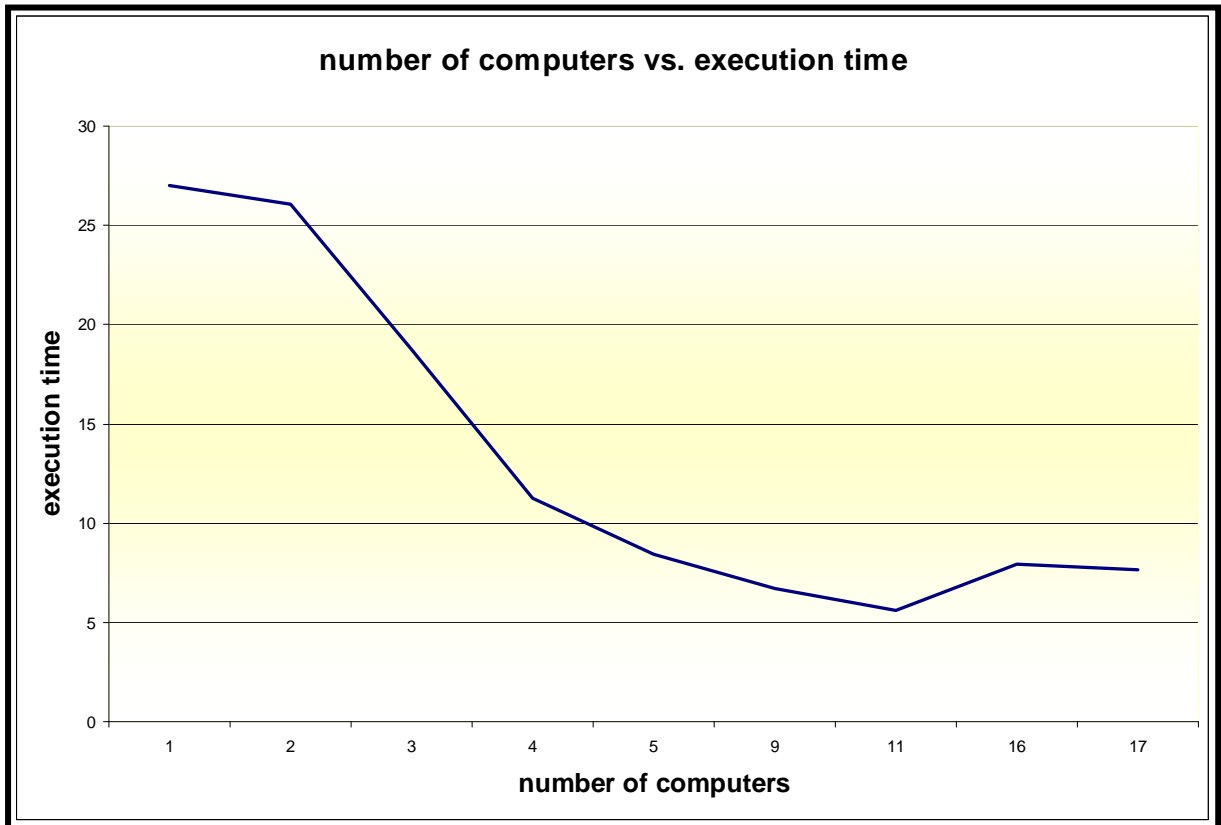


Figure 8 number of computers vs execution time

5.3 Future Work

Quadtrees are used to store spatial data with the aim of performing some spatial operations on the data such as search. By using the parallel construction method we proposed in this project, operations on the nodes of the quadtree can also be done in parallel. We suggest an extension to our PM Quadtree Gamma Machine that performs spatial operations such as searching on the constructed PM quadtree.

Also our Gamma Machine can be generalized to be used for the construction of other quadtrees. This can be done by adding an abstract class quadtree class where the various types of quadtrees can be represented as classes that inherit from the abstract class, and inside those classes the reaction-action they represent the decomposition criteria of these quadtrees can be defined.

Figure

In [11] authors proposed the parallel construction of PM quadtrees using the scan vector model, and in [13] the scan vector model was written using Gamma. We suggest that the work of the two papers be combined to produce a PM quadtree Gamma specification using the vector model.

5.4 Conclusion

Parallelism is a promising solution for modern complex applications that require high processing power and resources. One of these complex applications that would benefit from parallelism is the construction of quadtrees that are used to store spatial data. The Gamma language was proposed as a specification language without artificial sequentiality.

In our project we formulated the specification of the construction of PM quadtrees using Gamma. Using this specification we built our PM Quadtree Gamma Machine. We chose to present two implementations of our Gamma machine, a sequential version and a parallel version for the sake of comparing between the two paradigms and showing the power of parallelism. The parallel version was executed on a network of computers where the computation is distributed over the computers so that the construction of the PM quadtrees is carried out in parallel.

We performed several experimental tests where we compared the execution time of each of the two versions, the sequential version and the parallel version. Also the execution time of the parallel version on a different number of computers was compared. As expected the parallel version had a less execution time than the sequential version.

In general, we found that the Gamma language served as a good candidate in specifying the construction of PM quadtrees, due to the fact that each quadtree node can be processed independently leading to a parallel construction. This result can be generalized to include other hierarchical data structures. In other words we can say that the Gamma language can be used also in the construction of other hierarchical data structures as long as the decomposition criteria is well defined making it easy to transform it into a Gamma specification.

References

1. J.P. Banatre, A. Coutant and D. Le Metayer, "A parallel machine for multiset transformation and its programming style," *Future Generation Computer Systems*, pp. 133-144, 1988.
2. J.P. Banatre, A. Coutant and D. Le Metayer, "Parallel machines for multiset transformation and their programming style", *Informationstechnik*, Oldenburg Verlag, Vol.2/88, pp. 99-109, 1988.
3. J.P. Banatre, P. Fradet and D. Le Metayer, "Gamma and the Chemical reaction model : 15 years after ", LNCS 2235, pp.17-44, Springer Verlag 2001.
4. J. Banatre and D. Le Metayer, "Programming by Multiset Transformation," *Communications of the ACM*, vol. 36(1), pp. 98-111, January 1993.
5. G. Berry and G. Boudol, "The chemical abstract machine," *Journal of Theoretical Computer Science*, vol 96, pp. 217-248, 1992.
6. M. Chandy and J. Misra, "Parallel Program Design: A Foundation," Addison-Wesley, Reading, Mass, 1988.
7. P. Fradet and D. Le Metayer, "Structured Gamma," *Science of Computer Programming*, vol. 31,no. 2, pp.263-289, 1998.
8. R.A., Finkel, and J.L. Bentley, "Quad-trees; a data structure for retrieval on composite keys," *Acta Informatica*, vol. 4, pp. 1-9, 1974.
9. K. Gladitz and H. Kuchen, "Parallel implementation of the Gamma-operation on bags," *Proc. of the PASC0 conference*, Linz, Austria, 1994.
10. C. L. Hankin, D. Le Metayer, and D. Sands, "A calculus of Gamma programs," In *Proceedings of the Fifth International Workshop on Languages and Compilers for Parallel Machines*, LNCS 745, pp. 342-355, Berlin, 1993. Springer-Verlag.
11. E. G. Hoel and H. Samet, "Data-parallel primitives for spatial operations using PM quadtrees," *Proceedings of Computer Architectures for Machine Perception '95*, Como, Italy, September 1995, pp. 266-273
12. G. M Hunter and K. Steiglitz, "Operations on images using quad trees," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 1,no. 2 , pp.145-153, Apr. 1979.

13. D. Le M'etayer, "Higher-order multiset programming," in Proc. of the DIMACS workshop on specifications of parallel algorithms (American Mathematical Society, Dimacs series in Discrete Mathematics, vol. 18, 1994.
14. H. Lin, "Chemical Reaction Model Based Parallel Programming, Synthesis, Semantics, and Implementation," Phd thesis, University of Science and Technology of China, 1997.
15. J. J. Martin, "Organization of geographical data with quad trees and least square approximation," In Proceedings of the IEEE Conference on Pattern Recognition and Image Processing (June, LasVegas), IEEE, New York, 1982, pp. 458-463.
16. J. A., Orenstein, "Multidimensional tries used for associative searching," Inf. Process. Lett., vol. 14,no. 4, pp.150-157, June 1982.
17. H. Samet and R. E. Webber, "Storing a collection of polygons using quadtrees," ACM Transactions on Graphics, vol.4, no. 3, pp. 182-222, July 1985.
18. H. Samet, "The Quadtree and Related Hierarchical Data Structure," ACM Computing Survey, Vol. 16, No. 2, pp. 187-260, 1984.
19. H. Samet and R. E. Webber, "On encoding boundaries with quadtrees" IEEE Trans. Pattern Anal. and Mach. Intell.vol. 6, no.3, pp. 365-369, May 1984.
20. M. Shneier, "Two hierarchical linear feature representations: Edge pyramids and edge quad-trees," Corn@. Graph. and Image Process, vol.17,no. 3 , pp. 211-224, Nov. 1981.