A Pattern Language for Parallelizing Irregular Algorithms

Pedro Monteiro
CITI, Departamento de Informática
Faculdade de Ciências e Tecnologia Universidade Nova de Lisboa
2829-516 Caparica, Portugal
+351 212 948 536
pmfcm@fct.unl.pt

Miguel P. Monteiro
CITI, Departamento de Informática
Faculdade de Ciências e Tecnologia Universidade Nova de Lisboa
2829-516 Caparica, Portugal
+351 212 948 536
mmonteiro@di.fct.unl.pt

ABSTRACT
In irregular algorithms, data set's dependences and distributions cannot be statically predicted. This class of algorithms tends to organize computations in terms of data locality instead of parallelizing control in multiple threads. Thus, opportunities for exploiting parallelism vary dynamically, according to how the algorithm changes data dependences. This paper presents the first part of a pattern language for creating parallel implementations of irregular parallel algorithms and applications. Four patterns are proposed: *Amorphous Data-Parallelism*, *Data-Parallel Graph*, *Optimistic Iteration* and *In-Order Iteration*.

1. INTRODUCTION
Gustafson's law [1] states that any sufficiently large problem can be efficiently parallelized and has proven that parallelization is an effective way to accelerate the processing of massive data. However, in practice not all applications are easily parallelized and finding the right programming model and architecture for a given algorithm is quite challenging in the multicore era. Issues such as race conditions, communication, scalability, load balancing, data distribution, and locality further add to the effort of achieving efficient parallel programs.

The parallelization of *irregular algorithms* [2-3] is constrained by irregular accesses to dynamic pointer-based data structures whose data-dependence set can only be uncovered at run-time. To date, not much attention has been granted to this class of algorithms. By developing the pattern language presented here, we aim to increase the knowledge base of best practices in parallel programming of irregular algorithms and reduce the effort of producing new core synchronization concepts and other parallelism related components.

Patterns capture formal solutions to specific problems, while maintaining a level of abstraction above design models (e.g., UML) and source code. This way, patterns support a high-level form of reuse, which is independent from language, paradigm and hardware. Identifying and documenting patterns of complex concurrent software problems is one of key practices that will allow concurrent software development to be established as an engineering discipline – one which requires thorough systematic understanding and documentation of successful practices [4].

The rest of this paper is organized as follows: Section 2 discusses algorithmic irregularity and proposes a definition. Section 3 describes four patterns for our pattern language. Section 4 discusses related work and section 5 concludes the paper.

2. IRREGULAR ALGORITHMS
To date, the programming community did not reach a consensus on the definition of irregular algorithm, though algorithm irregularity is frequently considered in the literature. Most references to irregularity stressing the problem of indirect access to data [5-6] are found in articles published until around the mid 1990s, roughly when object-oriented programming became widespread in the programming community [7]. Hereafter, object-orientation, and essentially pointer-based programming, became the tool of choice for the implementation of most algorithms, including irregular algorithms, giving rise to the definition of irregularity as a function of the dynamism of pointer-based data structures [8-10]. Other definitions arise from the fact that, in pointer-based data structures, data growth can be unpredictable, resulting in irregular distributions of data among partitions [11] and input dependent communication patterns [12].

We propose that the problem of irregularity be defined in a more abstract way as a problem of *unpredictability of data dependences*. Having stated that, we further observe that traditional approaches to parallelization cannot be efficiently mapped to the unpredictable run-time behavior of irregular algorithms and applications.

Irregular problems arise often in the scientific domain as most simulation algorithms betray unpredictability and irregularity. Examples of such algorithms include sparse matrix computations, computational fluid dynamics, image processing, molecular dynamics simulations, galaxy simulations, climate modeling and optimization problems [13].

3. PATTERN LANGUAGE
The patterns in this paper are part of a pattern language, a set of closely related patterns that provide a solution to the parallelization of irregular algorithms. The set of patterns described here covers a subset of the entire parallelization methods available for irregular algorithms. This paper is focused on describing patterns for the parallelization of *worklist-based irregular algorithms*, using *optimistic or speculative techniques* [14].

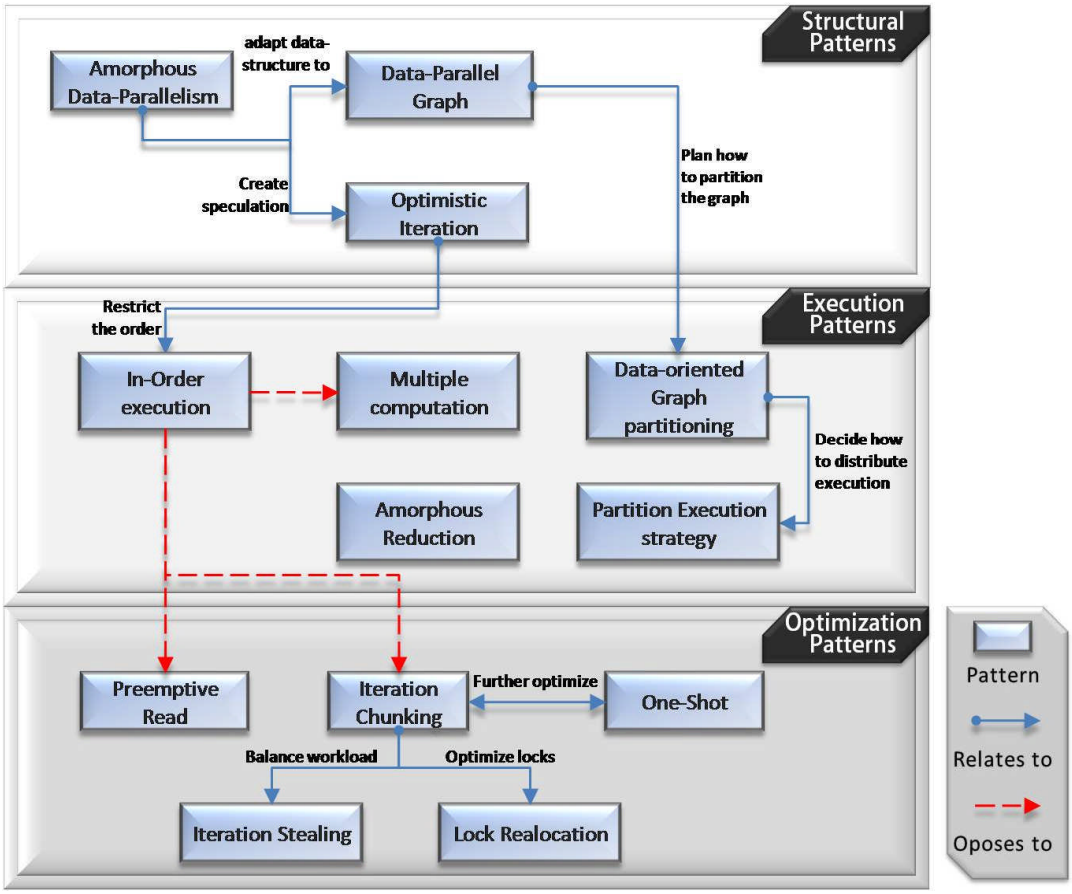

Figure 1 – Explicit relationships among patterns.

The pattern language is divided in three separate design spaces that follow a hierarchical structure representing the order in which the programmer must consider the implementation.

1. ***Structural patterns*** consider how to structure irregular algorithms in terms of their algorithmic properties and data-structures and how these will be affected by optimistic parallel execution. This set of patterns is the most important of the pattern language, since the two underlying design spaces build directly upon its properties. If the programmer cannot conform to these patterns, then using patterns from this language is discouraged.
2. ***Execution patterns*** effectively take into account how the actual execution of the algorithm is handled and how to guide the algorithm to explore the maximum amount of parallelism. Not taking these patterns into consideration may lead to lower performance benchmarks.
3. ***Optimization patterns*** are designed to present the final phase of implementation and essentially focus on some optimizations that, when applied over *structural patterns*, contribute to further increase the performance of irregular parallel algorithms.

Additional considerations as to the actual application of the patterns are worth pointing out, since there are relationships and dependences among the patterns that might provide further insight to their applicability to a particular context. **Figure 1** further describes these relationships.

Note that **Figure 1** presents a broad-brush overview of the relationships among patterns: not every relationship is explicitly represented as an arrow – the different levels of the patterns also imply dependences. Every *optimization* and *execution pattern* builds upon *structural patterns* and, while their implementation is not strictly necessary, *execution patterns* should at least be considered prior to any o*ptimization.*

This paper describes four of the patterns from the pattern language: *Amorphous Data-Parallelism* (section 3.2), Data-Parallel Graph (section 3.3), *Optimistic Iteration* (section 3.4) and *In-Order Execution* (section 3.5). The full set of patterns is documented in an upcoming MSc thesis [15] (scheduled for submission in February 2010).

3.1. Pattern-specific Terminology

An abstract terminology is used in the pattern descriptions to free the reader from algorithm and implementation specific jargon.

Available parallelism – number of iterations available for concurrent execution at any single instance in time.

Data element – a *data element* is a well-identified, describable unit of data that may be indivisible or consist of a set of data items. Additionally, data elements can be individualized from the overall data set. The identification of such data elements is algorithm-specific and usually comprises on an often repeated name whose meaning is associated to the algorithmic metaphor.

Data Set – a set of data elements or data-structure.

Iteration – an *iteration* represents the basic unit of a processing step in the solution of an algorithmic problem. Executing an algorithm entails repetitively executing the step a finite number of times, i.e., repeatedly applying the same operation over a set of data until de algorithm terminates. Execution often involves using the output of an iteration as the input of its predecessor.

Set of neighbors or neighborhood – represents the set of data elements that will be read or written by an iteration while it is being executed.

Operator – Algorithm operators can have either *read* or *write* semantics. If each iteration performs a set of operations, then the semantics of operators is defined by degree of influence:

- A Writer algorithm has at least one write operation. *Writes* are operators that introduce some sort of data-dependence by updating the data structure.
- Reader algorithms are composed of strictly *reader operations* and do not influence the set of data dependences in any way. These are infrequent in irregular algorithms and as such can be handled by regular data-parallel methods.

3.2. Amorphous Data-Parallelism

Problem

How to exploit concurrency in the presence of unpredictable data dependences?

Context

Traditional data parallelism exploits the decomposition of data-structures as a way to attain concurrent behavior. This entails dividing the data structure into independent sets and distributing them among processing units in a way that allows for the parallel application of a stream of operations.

However, when dealing with irregular algorithms, the nature of data dependences is unpredictable and dynamic and the amount of parallelism that can be achieved varies according to how the algorithm changes its data dependences. As such, the decomposition of the data-structure cannot be statically defined.

Amorphous data-parallelism is a particular form of data-parallelism that arises when the underlying data-structure has no fixed shape or size, i.e. is *amorphous*, implying that the amount of available opportunities for concurrency-free parallelism is unpredictable.

How then can we *decompose* an algorithm's data in a way that allows for data-parallel execution, when:

1. The occurrence and location of data accesses can only be properly estimated at runtime.
2. Concurrent computations may modify the structure of underlying data.

Forces

- **Data Granularity**
  Coarse-grained data may imply less communication but will introduce larger computational overhead and reduce the amount of available parallelism opportunities. If on the other hand the grain is fine, communications will represent the major overhead but will introduce a greater amount of available parallelism.
- **Redundancy vs. Communication**
  In a distributed environment, it can be profitable to perform redundant calculations in each of the distribution locales, instead of relying on data communication. This can introduce scalability opportunities.
- **Sequential to Parallel Traceability**
  If the pattern is correctly applied, there must be a simple and convenient mapping between the sequential and parallel versions of an implementation. This allows programmers to easily check the correctness of their implementation.
- **Concurrent access**
  Concurrent access must be carefully considered. Coarse locks might ensure mutual exclusion but data dependences between locked elements might at the same time create deadlock opportunities.

Solution

In irregular programs, an intimate knowledge is needed of how the different parts of the program interact and what part the data plays in the overall solution design. This is the basic context for the exploitation of data parallelism. In this context however, one must consider how the concurrent behavior will *operate over the data* and how to ensure the independence of computations in the overall parallelization strategy.

As such, the general solution for this problem entails being able to, at each iteration:

1. Identify the independent sets of data suitable to be executed in parallel
2. Decide which shared data elements need to be locked to avoid concurrent access
3. Ensure that the computational cost of independent sets remain balanced

A decomposition based on *Amorphous Data-Parallelism* must ensure that:

- Data dependent computations drive parallelism.
- Computations are performed in a way that introduces opportunities for independent parallel execution over the data.

The general solution of this pattern is comprised of the following steps:

Step 1 - Determine the type of algorithm operator

The definition of the type of operator allows the programmer to understand of how each iteration changes the structure of data.

Step 2 - Define a valid data-parallel decomposition based on the concept of basic data element

The *basic data element* represents the smallest independent set of data around which the parallelism will be driven. Defining this abstraction allows the programmer to consider how to apply

locking mechanisms to ensure concurrency. Together with the operator, the definition of data element allows identify the set of independent data elements.

Step 3 - Express computations in terms of the data-structure elements.

The programmer must choose how the data-structure will be iterated and reify an abstraction of the data for the computation as a call to `DataStructure.get(index)` or some similar instruction. This step is highly influenced by the choice of parallel programming language.

Step 4 - Repeatedly apply the computation algorithm to each data block.

This means not only iterating over the data-structure and applying the computation but also checking for the constraints of *Amorphous Data-Parallelism*:

```
1  foreach element in dataStructure atomically
   do
2     dataElements = // identify neighbors
3     lock dataElements;
4     compute(dataStructure.get(dataElements));
5     unlock dataElements;
6  endForeach
```

The foreach loop iteratively traverses the dataStructure and performs the algorithm-dependent computation. The atomicity of the operation is ensured by locking mechanisms which restrict the ways in which concurrent access can invalidate the computation.

Example

Delaunay Triangulation is an irregular algorithm for generating a mesh of triangles from a given set of points [16]. In order to generate valid triangulations, every triangle in the generated mesh must fulfill the *Delaunay property*, which states that for a circumference that intersects the vertex points of a triangle, no other point belonging to the mesh is located inside the circumference. When a violation of the *Delaunay property* is detected, the common edge is flipped to produce a valid triangulation. An example of the execution of *Delaunay Triangulation* is shown in **Figure 2.**

Delaunay Triangulation takes an input set of points in 2D space and as a first step surrounds all points with a single triangle (**A**). Then, it iteratively picks a single point (***B*** and ***C***), determines its surrounding triangle and splits the triangle in three new triangles, with the selected point as focal vertex. It then follows by checking the *Delaunay property* (***D***) and flipping when needed (***E***).

The underlying problem in *Delaunay Triangulation* can be parallelized in an amorphous data-parallel way by considering each triangle as the data block that drives parallelism. The repeated application of an activity to the various triangles exposes the parallelism inherent to the algorithm.

Delaunay Triangulation is *irregular*, i.e., there is a high degree of unpredictability in data-dependences. As the actual triangle mesh is created by iteratively selecting random points to be added to the mesh, its resulting structure cannot be estimated statically. Furthermore, when a point is added to the mesh, it might invalidate the *Delaunay Property* for a number of triangles, whose edges must be flipped. A flip operation incurs in additional

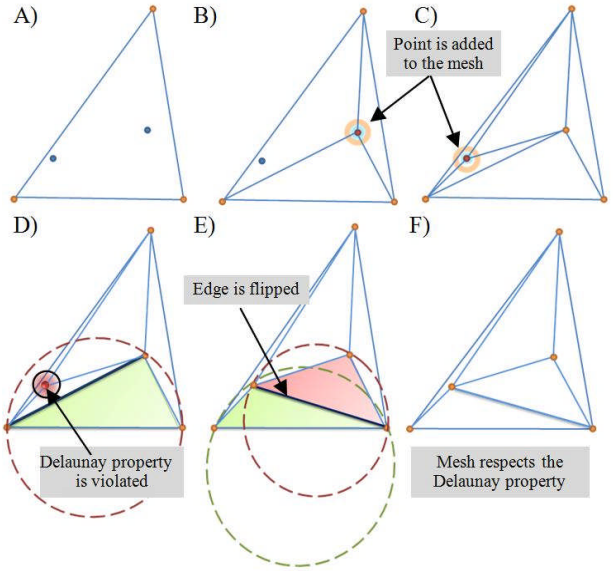

Figure 2 – Execution of the Delaunay Triangulation.

overhead, since every adjoined triangle will need to be locked.
The general *Amorphous Data-Parallell* solution to the *Delaunay Triangulation* example consists in:

Step 1 Delaunay's has two basic operations: *triangulation* and *edge flipping*, both of which change the data structure by changing the structure of data dependences. Therefore this is writer algorithm.

Step 2 On a first analysis, defining a point as the basic data element is the most straightforward choice. However, a better approach to this algorithm is to consider a triangle as the data element. This is true if considering that to ensure mutual exclusion we will always need to lock at least one triangle.

Step 3 In this step, we need to know how the data structure is built in order to be able to retrieve the data elements we need. For *Delaunay Triangulation*, this entails accessing the mesh and retrieving the triangle that surrounds the point we are currently triangulating.

```
mesh.getSurroundingTriangle(point);
```

Step 4 The general *amorphous data-parallel* form of *Delauney Triangulation* is consistent with the pseudo-code described in **Figure 3.**

Related Patterns

- ***Data Decomposition***
  Amorphous Data-Parallelism can be considered as a more specific form of *data-parallelism* [17] or *Data Decomposition* [18].
- ***Loop Parallelism***
  As the name implies, *Loop Parallelism* [17-18] helps uncover parallelism based in loops, which is a central part of *Amorphous Data-Parallelism*.

```
1  Set pointSet = //initialize set of points
2  TriangleMesh mesh;
3
4  foreach point in pointSet atomically do
5     Triangle tri
6     tri = mesh.surroundingTriangle(point)
7     adjacent = //get triangles adjacent to tri
8     lock adjacent and tri;
9     Triangle tri2 = triangulate(tri);
10    while tri2.isNotValid(){
11       flip(tri2);
12    }
13    mesh.addTriangle(tri2, adjacent);
14    unlock adjacent and tri;
15 endForeach
```

Figure 3 – Delaunay Triangulation with Amorphous Data-Parallelism.

Known Uses

Amorphous Data-Parallelism in irregular algorithms was first described by Kulkarni [19], although, to the best of our knowledge, we are the first to call it a pattern. We know of no other classifications of this type of parallelism. Recently, Lublinerman et al [20] presented Chorus, a high-level parallel programming model for irregular applications which uses the concept of *amorphous data-parallelism*. We know of no other parallel classification of this type of parallelism.

3.3. Data-Parallel Graph

Problem

How does a graph abstraction influence the opportunities for *Amorphous Data-parallelism* and the structure of the algorithm?

Context

On implementing an algorithm, much of the effort is spent on deciding what is the best underlying data structure on which to represent data to process and what are the characteristics that make it valuable on a concurrent environment.

In this context, we present a list of some of the reasons why graphs should be used:

1. Graphs are a generally used and accepted metaphor for describing structure and behavior. Examples of this can be as varied as state machines, flowcharts, UML diagrams, BPMN diagrams, EBNF diagrams, circuits, etc.
2. There is a large number of algorithms for graph traversal and search. The list includes *Depth-First* and *Breadth-First* traversal, *Iterative In-Order* and *Post-Order* [21], Dijkstra's *Shortest Path* [22] and Kruskal and Prim's algorithm [23], just to name a few.
3. Graphs represent structure and introduce constraints and properties such as hierarchy, connectivity, edge direction and weight, as defined in Graph Theory[24].
4. Complex graphs are composed of sub-graphs with similar structural properties. This allows for additional opportunities for divide-and-conquer strategies.
5. Graphs can be reconfigured with little or no effort, simply by loosening or tightening the connectivity constraints.
6. Graph nodes and edges can be associated with a variety of meanings and be of varying complexities.
7. Graphs can be used to represent virtually every data structure used in computation. The most common examples are:

   Trees – are a form of specialized bipartite, connected, acyclic and undirected graphs with one of its element distinguished as the root element[24]. Trees have many specialized forms (like the Binary-tree, Red-black tree, B-tree, AVL tree, etc) and can be used to represent other structures like hashtables and heaps.

   Lists – represent *path graphs*[24], acyclic graphs where every node is connected to at most 2 other nodes. Lists can be used to represent stacks, pipes and queues.

   Grids – are special distance-regular graphs that can be represented in two-dimensional space. Grids can be easily transformed into *cubes* (in three-dimensional space) or *hypercubes* (above the three-dimensional space). Grids can also be used to represent N-dimensional matrices.

Aside from the advantages stated above, programmers should take into consideration whether the graph abstraction actually benefits the implementation of the algorithm. Some data-structures, like trees and lists, are just as mature data-structures as graphs and are more attuned to some problems than others. Nonetheless, a Graph abstraction remains a perfectly good option.

Data Parallel Graph is focused on *amorphous data-parallel* graph algorithms, *i.e.*, graph algorithms that have an inherent amorphous data-parallel structure. Thus, if the underlying data in this algorithm is an irregular, pointer based data-structure, then, by all the reasons described above, a graph is the ideal choice.

The focus of this pattern is not to provide specific implementation solutions, merely to allow us to understand how graph characteristics influence irregular problems.

Forces

- **Specific vs. Reusable Implementation**
  A more specific graph implementation can provide additional performance to the algorithm but will make it inherently more difficult to implement and will hamper reusability. One must weigh the cost of implementation against the expected benefits. This force can also represent the decision of implementing a graph specifically tailored for the problem at hand or using an available graph library.
- **Optimization vs. Portability**
  If the data structure is tailored to a specific hardware, then performance will be greatly optimized but portability will be reduced by a similar proportion. This also reduces the chance of reproducing highly optimized benchmarks.
- **Update Cost vs. Performance**
  There must be a careful balance between the cost of dynamically updating the graph structure and the performance of the algorithm. If updates are

computationally expensive, then performance will be directly impacted in a negative way.

Solution

The general instantiation of a *Data-Parallel Graph* requires the following steps:

Step 1 - Identify algorithm-specific graph characteristics

A graph data-structure can have several different characteristics, which can be sorted in three distinct classes:

- **Edge characteristics**

  Direction: By default, an edge between two nodes is considered *bidirectional* or *undirected*. This means that there is a reciprocal relation between the connecting nodes and the graph can be traversed in any direction. However, there are some instances where edges can be one-way, i.e., traversing is restricted to a specific direction. In this case, the edges are said to be *directed*. An undirected graph can be represented by a directed graph where every node is connected to its neighbors by two directed edges. An example of a directed graph is a street map, since some streets are one-way and others are two-way, while a social network represents an undirected graph.

  Weight: Edges can have weights, i.e., there can be a cost associated with traversing a given edge. For instance, given a map of cities modeled as a graph where every edge has a cost associated with the distance between those same cities, we could use Dijkstra's *Shortest Path Algorithm* [22] to discover the shortest path between two cities.

  Direction and weight characteristics are completely orthogonal and we can have, for a given graph, any combination.

- **Node characteristics**

  Label: A node can have a label that distinguishes it from all other nodes. This is the case of the root node in trees or the source and sink nodes in the maximum flow problems [23].

  Value: nodes can have values that provide some contextual reference to the algorithm in question. An example is the case of boolean values that indicate whether a node has been visited before or color values, typical of graph coloring algorithms.

  Both label and value characteristics are completely orthogonal and we can have, for a given graph, any combination of the above two characteristics.

- **Structural characteristics**

  Structural characteristics of graphs infer a sense of how data is organized and help realize how special structural attributes are to be handled.

  Completeness: If every node is connected to every other node, then we say the graph is *complete*. Complete graphs are difficult to handle because they cannot be efficiently partitioned due to the absence of sub-graphs.

  Independence: A node is *independent* or *isolated* if it has no edges connecting it to other elements in the graph. A set is *independent* if it constitutes a sub-graph not connected by any edge to the main graph. The programmer must consider these characteristics when designing traversing and partitioning strategies for the algorithm.

  Connectivity: A graph is *connected* if for every two distinct nodes there is a path connecting them. This characteristic influences the amount of independence present in the graph.

  Cycles: A *cycle* exists if starting from a given node, there is a path through the graph that leads back to that same node. Most of the graphs contain cycles and this important characteristic requires the programmer to pay special attention so that the algorithm doesn't get caught in an endless loop around a cycle.

  Self-loops: A self loop happens when a node has an edge connecting to itself. This is a special instance of cycles since the algorithm can be caught in a *closed loop*, never leaving the same node. Self-loops must also be taken into account when partitioning the graph so that there is no node duplication.

Step 2 - Define the graph data-structure

The vast majority of programming languages don't provide built-in graph data-structures. However, there are a few libraries available. This is due to the fact that a generic graph library can be quite complex and can be implemented in n-number of ways (typically as adjacency lists or matrices but there are some purely object-oriented implementations available).

On deciding which implementation of graph data-structure to use, the programmer must take into account the following two factors:

- ***Reusability factor***

  On choosing or implementing a graph data-structure one must take care to identify the nature and reusability aspects of the problem at hand. If the problem is small and there is little probability that a full-fledged graph data structure will be needed, then an implementation using an adjacency list or matrix is a good option. This type of rough implementation is ideal when the cost of learning how to use a third-party library or of implementing a more generic and complex graph data-structure is considerably higher than the cost of implementing the overall algorithm.

- ***Optimization factor***

  More than the cost of learning how to use a graph library, the programmer must take care to consider if and how the algorithm can be optimized and how this optimization can be achieved with a wide-spectrum graph library.

  If the algorithm is intended to be run on a specific computational environment and is expected to achieve maximum performance in that environment, then the data-structure needs to be closely attuned to the underlying hardware configuration or operation system. This means that the data-structure should be designed with these specific characteristics and trade-offs in mind. For instance, a third-party graph library doesn't have many considerations for partitioning concerns.

  On the other hand, if an ideal performance can be achieved by fine-tuning the algorithm instead of the data structure, then probably the learning curve of using a third-party graph library has a lower cost than implementing a brand new data-structure.

Step 3 - Determining how the algorithm traverses the data-structure

This factor is important in this context because in the case of parallel implementations of algorithms, the graph traversal is what drives parallelism, i.e., the algorithm progresses through traversal of the elements of the graph and by performing the specified computations on each element. The traversal strategy is also very dependent on the way data is partitioned.

Step 4 - Determine how amorphous data-parallel computations can be composed

Graphs are ideal representations for amorphous data-parallel problems essentially because they represent a natural abstraction to the constraints of *Amorphous Data-Parallelism*. Let us consider the terminology used to describe *Amorphous Data-Parallelism*:

Data element: using a graph abstraction, the smallest independent data element is a node in the graph.

Iteration: in most irregular computations, choosing the next iteration is equivalent to retrieving the next node via one of the adjacent edges of the current node. One example is maximum-flow algorithms [23].

Operator: reader operators are equivalent to traversal operations while write operators conform to graph coarsening operations.

Neighbors: in graphs, the smallest neighborhood is composed by all nodes adjacent to the node currently being processed.

Furthermore, a graph can be seen as if composed of computational nodes connected by edges encoding computational dependences. This means that where we have a data node, we can assume that there is a corresponding computation. In addition, to every edge connecting two nodes, and therefore representing data dependences, we can assume that there is a corresponding edge in the computation that relates to computational dependences, that is, data derived from other computations.

Step 5 - Determine the need for data partitioning

At this point, it is necessary to identify if and how to partition the graph data-structure in a way that promotes further *Amorphous Data-Parallelism*. This entails *Data-Oriented Graph Partitioning*.

Example

As previously stated, using graphs as primary data-structures usually brings some useful advantages. This is the case of the *Sparse Cholesky Factorization Algorithm* a linear algebra method that transforms a matrix into a factor of a unique lower triangular matrix. The traditional Cholesky algorithms are implemented with a matrix-like data-structure. In this case however, since the matrix is sparse, that is, the majority of its elements are zero, the matrix can be mapped into a graph without compromising efficiency.

In **Figure 4,** a matrix (**a**) is mapped to a graph representation (**b**) where *r* nodes represent rows and *c* nodes represent columns. Edges between the nodes map the actual values presented in the matrix. Another, more efficient mapping (**c**) is obtained by creating one node per index value. This means that for an N*x*N matrix, there will be only N nodes. In this case, the main diagonal is represented as self-edges, while other edges are duplicated due to the matrix's symmetry. The mapping from matrix (**a**) to graph (**c**) can be accomplished by the code in **Figure 5**.

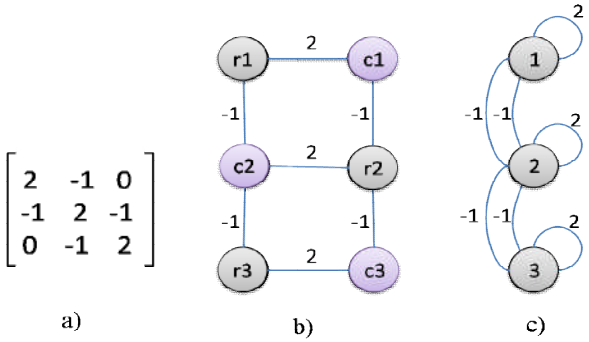

Figure 4 - Graph representations of a sparse symmetrical matrix.

```
1  Graph g;
2  Matrix [rows][columns] m;
3  for ( int col in columns){
4     for( int row in rows : row <= col){
5          if( m[row] [col] != 0){
6             Node ncol = g.addNode(col);
7             Node nrow = g.addNode(row);
8             //create edge and add its value
9             g.addEdge(nrow,ncol,m[row][col]);
10         }
11    }
12 }
```

Figure 5 – Matrix to Graph Transformation.

After this point, the programmer should adapt the algorithm to use this new graph representation of the sparse matrix.

Related Patterns

- ***Amorphous Data-Parallelism***
  Amorphous Data Parallelism is used to provide an underlying graph representation to the data required by the *Amorphous Data Parallelism pattern*.
- ***Data-Oriented Graph Partitioning***
  The graph data-structure influences partitioning which in turn will influence parallelization opportunities.

Known Uses

The RKPianGraphSort [25] is a rather recent approach to the problem of sorting a set of records in some pre-defined order. Sorting algorithms [26] are a well known and much studied set of irregular algorithms, with irregular data accesses but using non-dynamic (vectors) or semi-dynamic (lists) data-structures. This algorithm uses a graph-based sorting technique instead.

The Lonestar benchmark suite [3] offers a number of irregular applications designed with graph data-structures.

3.4. Optimistic Iteration

Also Known As

Data-Driven Speculation, Speculative Execution, Optimistic Execution

Problem

How to efficiently parallelize an algorithm that presents an amorphous data-parallel structure?

Context

When considering how to efficiently parallelize irregular algorithms, a more traditional approach using locks to synchronize concurrent accesses to data is possible, but would undoubtedly reduce the amount of available parallelism. Alternatives like static analysis techniques – such as points-to and shape analysis – or semi-static approaches – based on the inspector-executor model – fail to uncover the full set of potential parallelism, since they either only check data dependences at compile time or do not acknowledge dynamic dependence changes in data-structures.

To overcome the dependence chain under these conditions, programmers must take into account the advantages of speculative or optimistic parallelization techniques [19]. For this specific case, speculative execution of *Amorphous Data-Parallelism* implies being able to execute parts of the code without complete knowledge of the data dependences.

Forces

- **Implementation Cost vs. Benefit**
  Implementing an optimistic execution technique from scratch can be costly. The main disadvantage of these techniques lies in the complexity of handling miss-speculation problems, such as state saving and rollback actions. These can be quite challenging and if not done properly can increase the memory and computational cost of an algorithm to the point that there is no added benefit in using *Optimistic Iteration.*
- **Available Parallelism vs. Number of Conflicts**
  While finer-grained computations can yield a greater amount of available parallelism, it will also increase the likelihood of conflicts. The programmer should consider how many independent computations can take place simultaneously and find the optimal grain size.
- **Grain of Parallelism vs. Cost of Locking**
  If the cost of locking is not influenced by grain size, executing many fine-grained computations might be worse than executing a single coarser one.
- **Grain of Parallelism vs. Cost of Miss-Speculation**
  The cost of miss-speculation can be measured as the sum of the cost of corrective action with the cost of re-executing the work, added with the cost of acquiring and releasing locks for both the conflicted and re-executed iteration. Therefore, the cost of miss-speculation increases as the grain coarsens, as does the amount of wasted work due to rollbacks.

Solution

The idea behind *Optimistic Iteration* is to execute an algorithm in parallel while assuming that data dependences are never violated, *i.e.,* that there is no concurrent access to the data elements being processed. This does not mean that data is guaranteed to be independent but merely that the system must check for and take appropriate corrective actions when a dependence violation occurs. When no violations are detected, the results of iterations can be committed and the resulting data elements are added to the data dependence set.

Optimistic Iteration techniques are widely used in the parallel programming community and there are several different strategies for the implementation of speculative mechanisms. In this paper, we lack the space to describe all the different techniques in detail. For this reason, we have selected what we consider to be the main application-independent focal points of *Optimistic Iteration.*

The following steps describe how to speculatively execute an algorithm in an *Amorphous Data-Parallelism* way:

Step 1 - Determine the type of algorithm operator

As regards to the type of interaction with the data-structure, algorithm operators can have either *read* or *write* semantics. Strictly-reader algorithms don't have much to gain from *Amorphous Data-Parallelism*, since the structure of data dependences never changes.

Step 2 - Build data dependence graph

Optimistic Iteration uses the speculative execution of iterations as a way to break the highly coupled dependence chain around data elements. To create a valid mapping from data to iterations, the programmer needs to build the data dependence set for the specific algorithm under consideration. To this end, a graph can be seen as if composed of computational nodes connected by edges encoding computational dependences. Where we have a data node, we can assume there is a corresponding iteration. To every edge connecting two nodes, and therefore representing data dependences, we can assume there is a corresponding edge between iterations representing computational data dependences, i.e., data output from one iteration provides the input to another. This abstraction allows us to consider the various iterations of the algorithm as a traversal of data dependences.

Step 3 - Anticipate special ordering restrictions between iterations

The programmer must consider how strict the data dependences between the different iterations are. If iterations must be committed in a sequential-like order, then *In-Order Iteration* applies.

Step 4 - Predict the set of neighbors of each iteration

This is probably the most important and difficult step. For most irregular algorithms, the neighborhood can be predicted with a certain degree of accuracy. The prediction entails understanding what data elements will be read or written on each iteration. If the computations never change the structure of data dependences, the neighborhood can be determined in a straightforward manner. In matrix based algorithms, for example, the values of the matrix might change with every iteration but its structure remains the same. On the other hand, if the structure is dynamically changed, neighborhoods are harder to predict and while we can predict that a neighborhood is a set A of data elements, another parallel executing iteration might add a new element to the structure (say element b) which will in fact increase the neighborhood A to $A \cup \{b\}$. In this case, the neighborhood cannot be properly estimated and we are clearly in a situation where *Optimistic Iteration* is the best option.

Step 5 - Introduce locking mechanisms

The programmer must lock every neighboring data element with whichever locking mechanisms the implementation language or framework provides. Although optimistically assuming that there will be no concurrent access to data elements, it would be foolish not to lock the elements we are currently accessing. Locks are only released immediately prior to committing the iteration. This ensures atomicity in an iteration, in the sense that the data-structure always maintains a consistent state. These locks should not be all restrictive, *i.e.*, some operations should be allowed to perform concurrently while others require exclusive access to data.

Step 6 - Consider how to handle miss-speculation and rollback operations

When a conflict is detected, optimistic methods must be able to recover from the conflict without deadlocking or waiting for the locks to be released. Recovering from an illegal access requires the iteration to be reset to its initial state. There is a broad variety of methods and variations to provide this type of operation [19, 27-28]. The most frequently used methods of performing rollback are summarized next:

- **Lazy update –** changes are performed in cache and are only moved to main memory after the iteration commits successfully. This is equivalent to *Shadow copies*, where all operations are performed on a copy of the data element that then replaces the original, if the iteration commits successfully.
- **Reverse operations –** all operations are stored in an undo log. Rolling back an iteration is just a matter of applying reverse methods in a *Last-In First-Out* manner.
- **Snapshot –** prior to any change, a snapshot of the data is saved and all changes are performed on the original data-structure. In case of a rollback, the snapshot is recovered and replaces the modified data, restoring it to its original state.

After rollback, the iteration either is allowed to try again immediately or waits to be processed later.

Step 7 - Release all locks

Whether the iteration is able to commit or has to rollback, the final step is to release all locks that the iteration acquired and proceed to the next iteration.

Example

A general example of the implementation of a worklist-based irregular algorithm is shown in **Figure 6**. Here, a worker thread starts an unbounded while loop (line 5) and requests a new iteration from the scheduler (line 8). Each iteration then performs its corresponding computation and if it produces more work, it is added to the worklist (line 13-14) and the thread iterates again. If there is a conflict between iterations, an exception is thrown (line 18), otherwise the iteration is allowed to commit (line 16). This example assumes the existence of a *scheduler*, who is responsible for providing iterations to threads and to keep supplying work.

Picking up the example of *Delaunay Triangulation*, we can elaborate on the previous implementation and create a rough optimistic version of the algorithm (**Figure 7**). In this implementation, a worker thread starts an unbounded while loop (line 5) and asks for a new iteration from the scheduler (line 8). Each iteration then creates a new triangulation and, if that triangulation is invalid, it is added to the worklist (line 14-15) and the thread iterates again to correct the problem. If there is some conflict between iterations, an exception is thrown (line 21), otherwise the iteration is allowed to commit (line 19).

A wide variety of different optimistic parallel implementations of irregular algorithms have been proposed by the parallel programming community [3, 29-32].

```
1  Graph graph;
2  Worker worker;      //worker thread
3  Scheduler scheduler;
4
5  while (true){
6     try{
7          Iteration it;
8          it = scheduler.newIteration(worker);
9
10         scheduler.nextElement(it);
11         <result,work> = compute(graph,it);
12         graph.replaceSubgraph(it, result);
13         if(work.isNotNull())
14              scheduler.addWork(work);
15
16         scheduler.commitIteration(it);
17
18    }catch (violationException ve){
19         //do nothing
20         //graph is only updated on commit
21    }
22    //check for termination
23 }
```

Figure 6 – General optimistic implementation of a irregular algorithm.

```
1  Graph mesh;
2  Worker worker;      //worker thread
3  Scheduler scheduler;
4
5  while (true){
6     try{
7          Iteration it;
8          it = scheduler.newIteration(worker);
9          do {
10            scheduler.nextElement(it);
11            Triangle tri;
12            tri = triangulateOrFlip(mesh,it);
13            graph.replaceSubgraph(it, tri);
14            if(tri.isInvalid())
15               scheduler.addWork(it, tri);
16
17         } while(it.workLeft());
18
19         scheduler.commitIteration(it);
20
21    }catch (violationException ve)
22         //do nothing
23         //graph is only updated on commit
24    }
25    //Check for terminations
26 }
```

Figure 7 – Optimistic implementation of Delauney Triangulation.

Related Patterns

- ***Amorphous Data-Parallelism***
  The best way to handle *Amorphous Data-Parallelism* is by *Optimistic Iteration.*
- ***Data-Parallel Graph***
  The graph data-structure provides an appropriate data-structure for *Amorphous Data-Parallelism*, since it provides an ideal abstraction for the dependence graph.
- ***In-Order Iteration***
  If iterations have a restrict scheduling order, then *In-Order Iteration* applies.

Known uses

The first examples of optimistic parallelization were introduced in the 70s as a form of branch speculation [33-34]. Years later, in 1985, Jefferson presented one of the most well known optimistic methods: the *Time Warp mechanism* [35]. This mechanism implemented a method for transparently synchronize discrete-event simulation in distributed systems. Other well known optimistic techniques relate to loop speculation [14, 36]. Recently hardware techniques have enabled optimistically created parallel threads by tracking dependences by monitoring memory accesses made by loop iterations [37-41]. This technique, known either as *Thread level speculation* or *Speculative Multithreading*, proves to be quite useful to optimistically parallelize many applications and has been introduced in a significant number of parallelization architectures [42-46]. The *Galois framework* [19] is a recent approach to the parallelization of irregular algorithms whose execution model is based on optimistic execution.

3.5. In-Order Iteration

Also Known As

Ordered execution

Problem

How to find available *Amorphous Data-Parallelism* when tightly inter-dependent iterations constrain execution to a sequential iteration order?

Context

In most irregular algorithms, the order in which iterations are processed doesn't constrain the actual outcome. The end result is the same in whichever order the iterations are processed, as is the case of *Maxflow algorithms* [23] which always find the maximum flow, independently on the order in which nodes are processed. Others have different outputs according to the order of iteration but the correctness of the algorithm is not compromised. For example, in *Delaunay Triangulation* and *Refinement algorithms* [16], different orderings might produce different meshes, but the output will always be a mesh on which every triangle respects the *Delaunay Property*.

However, in some algorithms the order of iteration not only influences the end result, but is the sole order that ensures correctness. This is the case of *Event-driven simulation* [47], where events must be processed in global time order, and *Kruskal's minimum spanning tree* [23], where edges must be processed by increasing weight.

When dealing with optimistic parallelization of irregular algorithms, there is a good chance that the programmer will eventually be confronted with a restrictive ordering of execution that in theory would invalidate the advantages of speculation. Ordering is enforced when (1) Iterations depend on data previously computed in other iterations or (2) Iterations must follow data properties that enforce ordering constraints, like alphabetical or numerical order.

Matching the execution order of iterations to this sequential order can be achieved statically. The problem is how to extract *Amorphous Data-Parallelism* using *Optimistic Iteration* in such cases*?*

Forces

- **Amount of constraints vs. benefit**
  If ordering constrains only a very small set of iterations, then probably the cost of introducing *In-Order Iteration* doesn't cover the benefits in performance.
- **Order of rollback**
  The order of rollback of conflicting iterations could lead to deadlocks. If a higher priority iteration keeps rolling back due to conflicts with a lesser priority iteration, the algorithm would stop progressing and eventually might not terminate. A timeout mechanism could be an efficient way to check for priority errors.
- **Size of data set**
  The size of the data set influences the distribution of iterations and therefore, the bigger the data set, the more opportunities for independent execution exist.

Solution

The solution entails finding a way to extract a useful amount of *Amorphous Data-Parallelism* without disregarding the complexity of the ordering constraints. The steps to achieve this are described next:

Step 1 - Check for partial ordering

The majority of irregular algorithms enforce only *partial ordering*, i.e., only relatively small sets of iterations must meet ordering constraints. Independence between constrained sets is nevertheless possible. To illustrate this, let us consider the case where two iterations, A and B, are geometrically distant in that they don't share the same data elements. Nevertheless, some ordering is enforced – say alphabetical ordering – meaning that iteration A would always have to be executed before iteration B. Between these two iterations there is no available optimistic parallelism because executing B before A would lead to a conflict. However, this represents only a partial ordering. There is always a possibility that two A iterations could be executed concurrently. The same concept is applied to *minimum spanning tree algorithms* like *Kruskal's MST* [23] where usually more than one edge has the same or approximate weight – and *event based algorithms* with logic clocks – *Lamport clocks* [48] have causal order of events, yet a global ordering is only enforced for events that trigger actions on different processes. Same process events

have only to comply with local order and can occur concurrently with other local order events on other processors.

If the amount of iterations able to execute concurrently is significant, there might be no need to further refine the implementation to better explore *Optimistic Iteration.* However, the amount required for efficient performance is very algorithm-dependent and therefore requires experimentation to obtain reliable estimates.

Step 2 - Consider committal order

If there isn't enough available parallelism and performance is constrained, another solution is to consider that when algorithms have partial ordering constraints, that order needs only be enforced when iterations commit. The state that is observed by the system must remain consistent at all times, but consistency is only ensured after committal. When iterations are executing speculatively the state remains consistent and conforms to the order in which iterations should execute, *i.e.*, iterations should be allowed to execute in any order but committal order should be enforced.

One way to enable the above model of optimistic execution is to assign priorities to iterations and ensure that higher priority iterations always commit before the lower priority ones, while allowing lower priority iterations to execute speculatively. This way, state consistency is ensured. Uncommitted iterations should be stored in a heap-like data-structure and only allowed to commit when at the top of the heap. This implementation nevertheless leaves the programmer with the task of ensuring that when committing the root of the heap, that iteration has the highest priority and that no other higher priority iteration will occur in the future.

Example

Following *Kruskal's MST algorithm*, any two edges are independent if they don't have any node in common. Given this, the algorithm can process independent edges concurrently if their weight is less than or equal to any other edges waiting to be processed. Implementation of this algorithm by *In-Order Optimistic Execution* is shown in **Figure 8.**

Related Patterns

- ***Optimistic Iteration***
  In-Order Iteration is a special case of *Optimistic Iteration* where iterations have ordering constraints.

Known uses

On processing algorithms subject to ordering constraints, static approaches tend to provide more efficient implementations of algorithms. In cases where data dependences are available only at run-time, more careful hand-coded concurrent implementations using coarse locking mechanisms are usually preferred due to the small amount of parallelism available. Therefore, the number of speculative parallelization approaches that provide support for ordering is reduced.

The SETL language for set theory [49] and Galois [19] have similar ordered set iterators, but contrary to Galois, SETL doesn't have unbounded sets, and neither is a parallel programming language. An analogous use is that of out-of-order execution, where speculative execution of processor instructions is used to reduce the amount of time for required for future instructions [50]. Tomasulo's *reorder buffer* [33] is an approach to a Commit pool structure. Another approach adds speculative parallelization to FORTRAN-style DO-loops in X10, with resource to hardware transactional memory [51]. Safe futures are a related form of allowing for speculative ordered execution [52].

```
1  Graph graph;
2  Worker worker;      //worker thread
3  InOrderScheduler scheduler;
4  scheduler = //add iterations from graph
5  MST mst; //minimum spanning tree;
6
7  while (true){
8     try{
9          Iteration it;
10         it = scheduler.newIteration(worker);
11         do {
12              scheduler.nextElement(it);
13              inNode=it.getEdgeIn();
14              OutNode=it.getEdgeOut()
15              tree=//if valid path, create MST
16              mst.replaceSubgraph(it, tree);
17         } while(it.workLeft());
18    //Commit this iteration if top priority.
19    //If not, commit the top of the heap.
20         scheduler.commitInOrder(it);
21
22    }catch (violationException ve){
23         //do nothing
24         //graph is only updated on commit
25    }
26    //check for termination
27 }
```

Figure 8 – In-Order implementation of Kruskal's MST.

4. RELATED WORK

Pattern catalogs and languages for software design represent a widely prolific area of development, partly due to the renowned Gang of Four catalog of object-oriented design patterns [53]. From this first approach, patterns became popular in the field of reusable design, branching different application areas such as object-oriented programming [54], aspect-oriented programming [55] framework design [56-57], software architecture [58-59], components [60], machine learning [61-62] and even patterns about patterns [63-64].

The pattern language proposed here has close relations to some of the pattern languages for parallel processing proposed by the software pattern community – such is the case of pattern repository of the Hillside group [17], the pattern language of Mattson *et al* [18]. However, our view is that most pattern languages and catalogs mostly represent solutions for regular problems and handle irregularity as special cases, in which case the solution needs to conform to a different set of characteristics. Our pattern language contrasts with this view and is specifically focused on irregular problems, which are considerably more complex. We propose instead to classify the solution to regular problems as a subset of the solution of irregular problems. There are nonetheless some pattern languages designed for specific irregular algorithms, as is the case of Dig *et al* pattern language for N-Body methods [65].

5. CONCLUSIONS AND FUTURE WORK

This paper describes a pattern language for the parallelization of irregular algorithms. Currently, none of the three design spaces (Structure, Execution and Optimization) present a high degree of maturity considering that they don't express the full set of solutions for the parallelization of irregular algorithms. However, when considering optimistic approaches, these patterns represent a well developed and mature body of knowledge.

In future, we plan to further refine this pattern language, as well as to produce a set of case studies to validate our approach. Efforts for developing a collection of aspect-oriented implementations of the patterns are about to begin. The majority of the patterns shown here were developed using the Galois Framework [19] as case study. Other frameworks and languages have considerably different methodologies for handling irregularity. We hope to explore these alternatives as well, and relate them to the patterns described here, enriching and maturing the language, as well as enhancing its potential applicability to cover a broader set of techniques and methods targeting parallel irregular algorithms.

6. ACKNOWLEDGMENTS

For accompanying us through the development of this pattern language, we owe our thanks to Keshav Pingali and Mario Méndez-Lojo, of the Institute for Computational Engineering and Sciences of the University of Texas at Austin, and to João L. Sobral of Minho University, Portugal.

This paper builds upon the work produced at the University of Texas in Austin, namely the Galois framework, and was supported by the project Parallel Refinements for Irregular Applications (UTAustin/CA/0056/2008) funded by FCT-MCTES and European funds (FEDER).

7. REFERENCES

[1] J. Gustafson, "Reevaluating Amdahl's Law," *Communications of the ACM*, vol. 31, no. 5, 1988.

[2] R. Biswas, L. Oliker, and H. Shan, "Parallel computing strategies for irregular algorithms," *Annual Review of Scalable Computing*, 2003.

[3] M. Kulkarni, M. Burtscher, K. Pingali *et al.*, "Lonestar: A suite of parallel irregular programs." pp. 65-76.

[4] D. Schmidt, and F. Buschmann, "Patterns, frameworks, and middleware: their synergistic relationships." pp. 694-704.

[5] R. Asenjo, F. Corbera, E. Gutiérrez *et al.*, "Optimization techniques for irregular and pointer-based programs." pp. 2-13.

[6] R. Das, M. Uysal, J. Saltz *et al.*, "Communication optimizations for irregular scientific computations on distributed memory architectures," *Journal of Parallel and Distributed Computing*, vol. 22, no. 3, pp. 462-478, 1994.

[7] L. Capretz, "A brief history of the object-oriented approach," *ACM SIGSOFT Software Engineering Notes*, vol. 28, no. 2, 2003.

[8] K. Pingali, M. Kulkarni, D. Nguyen *et al.*, "Amorphous Data-parallelism in Irregular Algorithms."

[9] Z. Zhang, and J. Torrellas, "Speeding up irregular applications in shared-memory multiprocessors: Memory binding and group prefetching," *ACM SIGARCH Computer Architecture News*, vol. 23, no. 2, pp. 188-199, 1995.

[10] M. Hermenegildo, "Parallelizing irregular and pointer-based computations automatically: Perspectives from logic and constraint programming," *Parallel Computing*, vol. 26, no. 13-14, pp. 1685-1708, 2000.

[11] D. Nikolopoulos, C. Polychronopoulos, and E. Ayguadé, "Scaling irregular parallel codes with minimal programming effort." p. 16.

[12] S. Taylor, J. Watts, M. Rieffel *et al.*, "The concurrent graph: basic technology for irregular problems," *IEEE Parallel and Distributed Technology*, pp. 15-25, 1996.

[13] E. Gutierrez, R. Asenjo, O. Plata *et al.*, "Automatic parallelization of irregular applications," *Parallel Computing*, vol. 26, no. 13-14, pp. 1709-1738, 2000.

[14] M. Gupta, and R. Nim, "Techniques for speculative run-time parallelization of loops." p. 12.

[15] P. Monteiro, "Identification of Concurrency Concerns in the Galois Framework," CITI - Departamento de Informática, Faculdade de Ciências e Técnologia da Universidade Nova de Lisboa, Almada, 2010.

[16] J. Shewchuk, "Delaunay refinement algorithms for triangular mesh generation," *Computational Geometry: Theory and Applications*, vol. 22, no. 1-3, pp. 21-74, 2002.

[17] K. Keutzer, and T. Mattson, "Our Pattern Language (OPL): A Design Pattern Language for Engineering (Parallel) Software."

[18] T. Mattson, B. Sanders, and B. Massingill, *Patterns for parallel programming*: Addison-Wesley Professional, 2004.

[19] M. Kulkarni, "The Galois System: Optimistic Parallelization of Irregular Programs," Cornell University, 2008.

[20] R. Lublinerman, S. Chaudhuri, and P. Cerny, "Parallel programming with object assemblies." pp. 61-80.

[21] J. Launchbury, "Graph algorithms with a functional flavour," *Lecture Notes in Computer Science*, vol. 925, pp. 308, 1995.

[22] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269-271, 1959.

[23] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to algorithms*, Cambridge,Mass. ; London: MIT Press, 1990.

[24] J. Gross, and J. Yellen, *Graph theory and its applications*: CRC press, 2006.

[25] R. Pal, "RKPianGraphSort: a graph based sorting algorithm," *Ubiquity*, vol. 2007, no. October, pp. 16, 2007.

[26] S. Akl, *Parallel sorting algorithms*: Academic Press, Inc. Orlando, FL, USA, 1990.

[27] C. Carothers, K. Perumalla, and R. Fujimoto, "The effect of state-saving in optimistic simulation on a cache-coherent non-uniform memory access architecture." pp. 1624-1633.

[28] W. Dieter, and J. Lumpp, "A user-level checkpointing library for POSIX threads programs." pp. 224-227.

[29] C. Antonopoulos, X. Ding, A. Chernikov *et al.*, "Multigrain parallel delaunay mesh generation: challenges and opportunities for multithreaded architectures." p. 376.

[30] N. Chrisochoides, C. Lee, and B. Lowekamp, "Mesh generation and optimistic computation on the grid."

[31] I. Kolingerová, and J. Kohout, "Optimistic parallel Delaunay triangulation," *The Visual Computer,* vol. 18, no. 8, pp. 511-529, 2002.

[32] C. Verma, "Multithreaded Delaunay Triangulation."

[33] R. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of research and Development,* vol. 11, no. 1, pp. 25-33, 1967.

[34] J. Fisher, "Very long instruction word architectures and the ELI-512." pp. 140-150.

[35] D. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems (TOPLAS),* vol. 7, no. 3, pp. 425, 1985.

[36] L. Rauchwerger, and D. Padua, "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization." pp. 218-232.

[37] P. Marcuello, A. González, and D. de Computadors, "A quantitative assessment of thread-level speculation techniques." pp. 595–604.

[38] J. Steffan, C. Colohan, A. Zhai *et al.*, "A scalable approach to thread-level speculation," *ACM SIGARCH Computer Architecture News,* vol. 28, no. 2, pp. 1-12, 2000.

[39] S. Wang, X. Dai, K. Yellajyosula *et al.*, "Loop selection for thread-level speculation," *Lecture Notes in Computer Science,* vol. 4339, pp. 289, 2006.

[40] M. Prabhu, and K. Olukotun, "Using thread-level speculation to simplify manual parallelization." p. 12.

[41] Y. Zhang, L. Rauchwerger, and J. Torrellas, "Hardware for speculative run-time parallelization in distributed shared-memory multiprocessors." p. 162.

[42] L. Codrescu, D. Wills, and J. Meindl, "Architecture of the Atlas chip-multiprocessor: Dynamically parallelizing irregular applications," *IEEE Transactions on Computers,* vol. 50, no. 1, pp. 67-82, 2001.

[43] J. Oplinger, D. Heine, S. Liao *et al.*, "Software and hardware for exploiting speculative parallelism with a multiprocessor," *Computer Systems Laboratory Technical Report CSL-TR-97-715, Stanford University,* 1997.

[44] M. Dorojevets, and V. Oklobdzija, "Multithreaded decoupled architecture," *International Journal of High Speed Computing,* vol. 7, no. 3, pp. 465, 1995.

[45] J. Tsai, and P. Yew, "The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation," *Urbana,* vol. 51, pp. 61801-1351.

[46] P. Marcuello, and A. González, "Control and data dependence speculation in multithreaded processors." pp. 98-102.

[47] S. Das, "Adaptive protocols for parallel discrete event simulation." pp. 186-193.

[48] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," 1978.

[49] J. Schwartz, R. Dewar, E. Schonberg *et al.*, *Programming with sets; an introduction to SETL*: Springer-Verlag New York, Inc. New York, NY, USA, 1986.

[50] J. Hennessy, D. Patterson, D. Goldberg *et al.*, *Computer architecture: a quantitative approach*: Morgan Kaufmann, 2003.

[51] C. von Praun, L. Ceze, and C. Ca caval, "Implicit parallelism with ordered transactions." p. 89.

[52] A. Navabi, X. Zhang, and S. Jagannathan, "Quasi-static scheduling for safe futures." pp. 23-32.

[53] E. Gamma, R. Helm, R. Johnson *et al.*, "DESIGN PATTERNS-Elements of Reusable Object-Oriented Software."

[54] J. Noble, and A. Sydney, "Towards a pattern language for object oriented design," *Proc. of Technology of Object-Oriented Languages and Systems (TOOLS Pacific),* vol. 28, pp. 2-13.

[55] U. Zdun, "Pattern language for the design of aspect languages and aspect composition frameworks," *IEE Proceedings-Software,* vol. 151, no. 2, pp. 67-84, 2004.

[56] D. Roberts, and R. Johnson, "Evolving frameworks: A pattern language for developing object-oriented frameworks," *Pattern Languages of Program Design,* vol. 3, pp. 471–486, 1998.

[57] K. Wolf, and C. Liu, "New clients with old servers: A pattern language for client/server frameworks," *Pattern Languages of Program Design*, pp. 51–64, 1995.

[58] F. Buschmann, R. Meunier, H. Rohnert *et al.*, "A system of patterns: Pattern-oriented software architecture," Wiley New York, 1996.

[59] P. Avgeriou, and U. Zdun, "Architectural patterns revisited–a pattern language." pp. 1-39.

[60] D. Spinellis, and K. Raptis, "Component mining: A process and its pattern language," *Information and Software Technology,* vol. 42, no. 9, pp. 609-617, 2000.

[61] P. Avgeriou, A. Papasalouros, S. Retalis *et al.*, "Towards a pattern language for learning management systems," *Educational Technology & Society,* vol. 6, no. 2, pp. 11-24, 2003.

[62] P. Goodyear, P. Avgeriou, R. Baggetun *et al.*, "Towards a pattern language for networked learning." pp. 449-455.

[63] G. Meszaros, and J. Doble, "Metapatterns: A pattern language for pattern writing."

[64] J. Coplien, and B. Woolf, "A pattern language for writers' workshops," *C PLUS PLUS REPORT,* vol. 9, pp. 51-60, 1997.

[65] D. Dig, R. Johnson, and M. Snir, "N-Body Pattern Language."