Parallelizing Irregular Algorithms: A Pattern Language

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

Keshav Pingali
Institute for Computational Engineering and Sciences
The University of Texas at Austin
pingali@cs.utexas.edu

ABSTRACT

Outside of the high-performance computing domain, many applications are *irregular* in the sense that opportunities to exploit parallelism change throughout the computation, due to the use of complex, pointer-based data structures such as lists and graphs. However, the parallel programming community has relatively little experience in parallelizing irregular applications, and we presently lack a deep understanding of the structure of parallelism and locality in the algorithms that underlie these applications. In this context, irregular algorithms pose a challenging problem to current parallelization methods and techniques.

In recent years, the Galois project has proposed an approach for parallelizing irregular algorithms and applications that is based on a small set of simple abstractions. In this paper, we describe the Galois approach by means of a pattern language for parallel programming, thereby highlighting the key features of this approach, and elucidating more generally the concurrency patterns in irregular algorithms.

Categories and Subject Descriptors

D.1.3 **[Programming Techniques]** Concurrent Programming – *Parallel Programming* D.3.3 **[Programming Languages]** Language Constructs and Features – *Patterns and Frameworks*

General Terms

Algorithms, Design

Keywords

Pattern Language, Irregular Algorithms, Parallel Programming, Reverse Engineering, Object-Oriented Frameworks

1. INTRODUCTION

Parallel programming has been used for a long time in specialized application areas such as high-performance computing (HPC) and databases. Over the years, we have acquired a good understanding of the patterns of parallelism and locality in the underlying algorithms of these problem domains, which led to the creation of programming notations, and compiler and runtime technology for supporting the parallel execution of these applications.

The advent of multicore processors makes it feasible to execute applications in parallel. However, most applications outside of HPC and databases are *irregular* because they use complex, pointer-based data structures such as lists and graphs, whose patterns of computation are not statically determinable. In contrast, HPC and database applications have statically determinable patterns of computation as they use data structures such as dense arrays and relations. HPC applications like stencil computations and FFTs are amenable to forms of parallelism independent of runtime values, thus it is possible for a compiler or programmer to expose and schedule the parallelism before execution. However, parallelism in irregular applications cannot be exposed by compile-time techniques such as dependency analysis.

Dependencies between computations in irregular applications are a function of runtime entities, such as the structure of input graphs and the values on nodes and edges, so most of the work of parallelizing these applications must be performed during execution. Unfortunately, we still lack a thorough understanding of the patterns of parallelism and locality in irregular algorithms, which hampers the design of programming notations, compilers and runtime systems for supporting the parallel execution of these applications.

Recently, the Galois project has made some advances in this key area [33] by identifying the type of parallelism that best takes advantage of the dynamic structure of irregular algorithms. The Galois research shows that this generalized form of data-parallelism called *Amorphous Data Parallelism (ADP)* is ubiquitous in irregular applications.

This paper extends our previous work [49] by describing the most important aspects of the Galois approach using the language of parallel programming patterns. Nevertheless, the pattern language is independent of the details of the Galois framework and is not restricted to this system in any way. Patterns represent tangible solutions to problems in a well-defined context within a specific domain and provide support for wide reuse of well proven concepts and techniques, independent from methodology, language, paradigm and architecture [4].We aim to disseminate this knowledge to both expert and non-expert programmers through patterns, thus easing the adoption of these ideas in other systems. The patterns from this pattern language are meant to be used together, in view of providing a solution for wider and more complex problems than those tackled by a single pattern or a set of unrelated patterns. They comprise a true pattern language in that they provide a complete solution for a complex problem – the parallelization of Irregular Algorithms in this case.

The remainder of this paper is structured as follows. Section 2 provides an overview of Irregular algorithms and describes two examples in detail, which are used in subsequent sections. Section 3 provides a short overview of the Galois Framework. The Pattern Language is documented in section 4. Section 5 discusses related work and section 6 concludes the paper.

2. IRREGULAR ALGORITHMS

The domain of multicore programming uses a plethora of techniques, methods and languages to achieve efficient parallel implementations of algorithms and applications, like pthreads, OpenMP, MPI, among others. However, writing parallel code is not a trivial task and it is hard to hide the complexities of synchronization, data races, memory consistency, distribution, etc. Parallelizing compilers that use points-to and shape analysis to parallelize sequential code are especially apt for creating the needed level of abstraction from parallelism concerns. Nonetheless, these more than often fail to uncover the true potential parallelism in algorithms where data-dependencies are only known at runtime and thus no efficient schedule can be foreseen. The class of algorithms that presents such irregular dependencies is termed Irregular Algorithms [33, 36].

In the majority of irregular algorithms, data is present in the form of graphs, trees or lists with a high degree of dependency between the nodes. Thus, computations performed on a node have a high risk of interfering with computation on other nodes. As each computation occurs, the set of dependencies between nodes tends to change accordingly. Therefore, scheduling strategies cannot be uncovered at compile-time, requiring iterative reevaluation of dependencies and rescheduling of operations.

Irregular problems arise often in the scientific domain as most simulation algorithms betray irregularity. Examples of such algorithms include sparse matrix computations, computational fluid dynamics, image processing, molecular dynamics, climate modeling and optimization problems [23]. Implementing this class of algorithms on distributed-memory machines requires frequent fine-grain communications, to encompass changes in overlapping data-sets, which results in poor performance. On the other side of the spectrum, implementations of irregular algorithms on shared-memory machines alleviate these problems, but in turn require heavy cache coherence and synchronization protocols to enforce a consistent view of memory.

This implementation complexity cannot be efficiently handled by traditional approaches to parallelization, which do not account for the unpredictable run-time behavior of irregular algorithms. Therefore, efficient parallel implementations of irregular algorithms remain a challenging problem.

Next, two use-cases of widely known irregular algorithms are presented, to illustrate some of the challenges commonly associated with parallelizing this class of algorithms. The first use case presents Delaunay Triangulation [59], an algorithm to create Delaunay triangulations from a set of points, and the second is Kruskal's Minimum Spanning Tree. These irregular algorithms and many others are available for a more detailed study in the Lonestar Benchmark Suit [34].

2.1 Delaunay Triangulation

Delaunay Triangulation, also referred to as Delaunay Mesh Generation, is an irregular algorithm for generating a mesh of triangles from a given set of points [59]. When a new point is added to the mesh, its surrounding triangle is split into three new triangles, with the new point as the central vertex (see Fig 1, a-c). The new triangles must be valid according to the *Delaunay property,* which states that no point can exist inside the circumference that intersects the vertex points of a triangle (see Figure 1-d). When this property is violated, the common edge is flipped to produce a valid triangulation (Figure 1, d-f).

2.1.1 Irregularity

As points are randomly added to the mesh, its resulting structure cannot be statically predicted. Moreover, parallel addition of distinct points with the same surrounding triangle is impossible without some sort of concurrency control and ordering. Thus, concurrency control excludes parallel addition of new points to a triangle, if it is already being processed. However, parallel addition of points to non-adjacent triangles is allowed.

The random addition of points to the mesh prevents us from predicting how many triangles are ripe for processing in parallel at each step of the algorithm and thus traditional data parallelism is rendered ineffective.

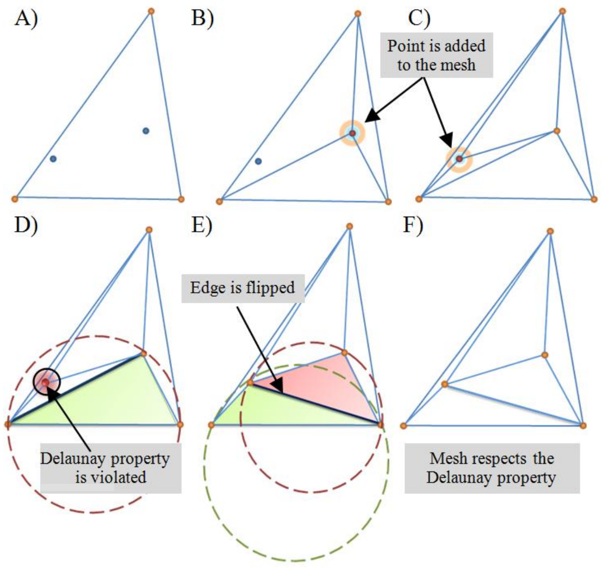
A)
B)
C)
Point is added to the mesh
D)
E)
F)
Edge is flipped
Delaunay property is violated
Mesh respects the Delaunay property

Figure 1 – Execution of Delaunay's Triangulation algorithm

2.2 Kruskal's Minimum Spanning Tree

Kruskal's MST is a well-known algorithm for calculating the minimum weight spanning-tree of a connected weighted undirected graph. This entails finding, at each step of the algorithm, the edge with the minimum weight and adding it to the MST. However, no cycles are allowed and so the algorithm must keep track of the connectedness of its sub-graphs. When a new edge is added to the MST, it checks if the MST edged deriving from both its nodes are connected. If so, the edge is discarded instead of being added to the MST, as it would introduce a cycle. This verification is usually performed through a disjoint-set data structure with Union-Find operations.

2.2.1 Irregularity

In contrast to Delaunay Triangulation, this algorithm presents an restrictive processing order, i.e. elements must be processed from the edges with the minimum weight to the edges with the maximum weight. This restricts the number of parallel computations that might occur at each step.

Edges can be added in parallel to the MST if:

- They have the same weight and that weight is the current minimum.
- If for some small n-value, an edge has weight less than or equal to *minimum+n*, and adding the edge will not lead to the creation of a cycle by a minimum weighted edge.

Thus, parallel addition of minimum weighted edges (or near to minimum) is allowed if they don't belong to the same sub-graph. Provided the addition is made in disjoint sub-graphs, edges can be added in parallel.

3. THE GALOIS FRAMEWORK

Galois is a framework for parallelization of irregular algorithms, from which the concepts expressed by our pattern language were derived, through a process of analysis and reverse engineering. The effort needed to create efficient parallel versions of this class of algorithms is not easily managed by non-expert scientific programmers, which are more accustomed to view problems in a sequential manner [52]. Thus, in the Galois approach [54], the problem of exposing and exploiting *Amorphous data Parallelism* is addressed through a small number of simple abstractions based on optimistic (or speculative) parallelization.

3.1 Programming model and data structure library

Galois[1] is based on a high-level programming pattern that was abstracted from a study of large number of irregular algorithms, namely Delaunay's mesh algorithms [59], *single-source shortest-path* computations and *maxflow computations* [9].

There are three main components in Galois: (1) a simple yet powerful set of programming constructs that help non-expert programmers express key properties of irregular algorithms, (2) a library of concurrent data structures, and (3) a runtime system that uses optimistic parallel execution. The programming model is inherently sequential: in the current implementation, it is sequential Java extended with two *Galois set iterators that provide implicit parallelism*. These iterators are similar to set iterators in conventional languages like Java and C++ and iterate over unordered and ordered sets of work-items. They also allow for adding new elements to the set at runtime. The runtime system exploits the implicit parallelism in Galois iterators by executing iterations speculatively in parallel, i.e. executing in parallel assuming there is no concurrency in data access and rolling back execution if a conflict is detected. Conflict detection and recovery is handled by the library data structures and the runtime system.

[1] http://iss.ices.utexas.edu/?p=projects/galois

This approach enables application programmers to write irregular programs without having to reason explicitly about concurrency.

Figure 2 illustrates an abstraction of a typical irregular algorithm. At each point during its execution, operators are applied to certain nodes or edges in the graph (such nodes are called *active elements*). Each application of an operator is termed an *activity,* and may require reading or writing of other nodes and edges in the graph (i.e. the *neighborhood* of that activity). In Figure 2, the red colored nodes represent active nodes, and shaded regions represent the neighborhoods of those active nodes. In some algorithms, activities may *modify* the graph structure of the neighborhood by adding or removing graph elements. In general, there are many active nodes in a graph.

We distinguish two important categories of algorithms: *unordered algorithms* allow the implementation to pick *any* active node for execution (e.g., preflow-push maxflow and Delaunay mesh refinement) and can be programmed using a Galois *unordered* set iterator. In contrast, *ordered algorithms* (e.g., Prim's or Kruskal's algorithms for computing minimal spanning trees) impose a problem-dependent order for the processing of active nodes and can be programmed using a Galois *ordered* set iterator.

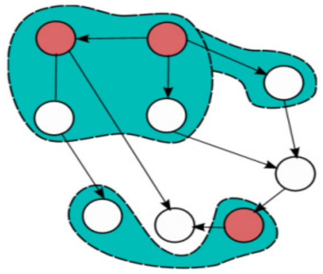

Figure 2 - Data-centric view of algorithms: red nodes are active and wait to be processed. Processing a node will potentially access and modify other nodes within the shaded neighborhoods. Parallel processing of active nodes will lead to conflicts when the neighborhoods of active nodes intersect.

3.2 Baseline parallel execution model

The Galois API of concurrent data structures covers concurrent data structures such as graphs, priority queues and sets, whose functionality is exposed through a conventional data structure API. For example, the graph API includes methods for returning the neighbors of a given node, the outgoing and incoming edges of a node, adding/deleting nodes and edges, etc. All concurrency control is implemented within the classes of this library. Figure 2 shows how opportunities for exploiting parallelism arise in graph algorithms: if there are many active elements at some point in the computation, each one is a site where a processor can perform computation, subject to neighborhood constraints. In the baseline parallel execution model, the graph is stored in shared-memory, and active nodes are processed by some number of threads. Like thread-level speculation [32] and transactional memory [25], the Galois system uses speculative parallel execution to handle the problem of dependencies that can only be elucidated at runtime. A free thread picks an arbitrary active node and speculatively applies the operator to that node, making calls to the graph class API to perform operations on the graph as needed. The neighborhood of an activity can be visualized as a blue ink-blot that begins at the active node and spreads incrementally whenever a graph API call is made that touches new nodes or edges in the graph. To enforce neighborhood constraints, each graph element has an associated exclusive *abstract lock*, which is held until the activity terminates. If a lock cannot be acquired because it is already owned by another thread, a conflict is reported to the runtime system, which rolls back one of the conflicting activities. To enable rollback, each graph API method that modifies the graph makes a copy of the data before modification. Like abstract lock manipulation, rollbacks are a service implemented by the library and runtime system. The activity terminates when the application of the operator is complete and all acquired locks are released.

Intuitively, the use of abstract locks ensures that graph API operations from concurrently executing iterations *commute* with each other, ensuring that the iterations appear to execute in some serial order as required by the semantics of the Galois set iterator. There are more sophisticated techniques for checking commutativity, but these are more complex to implement [36]. Commuting graph API operations that touch the same locations in the concrete representation must be synchronized. The programming model enables the application programmer to turn off abstract locking and conflict detection for the cases it is safe to do so. This can improve performance but may introduce race conditions if used incorrectly.

4. THE PATTERN LANGUAGE

Patterns allow the reuse of solutions and the widespread adoption of domain knowledge. A pattern language is a set of inter-dependent patterns that provide a complete solution to a complex problem [4]. The patterns presented here are meant to be used together to a solution for the parallelization of Irregular Algorithms. However, as with other patterns, each pattern might be used independently if the context for its usage coincides with the context of the pattern.

Due to the plethora of new and emergent parallel programming methods and techniques, the patterns presented here are more along the lines of conceptual patterns and don't follow the typical structure of design patterns, which are more concrete. We present patterns that provide design directions that steer the programmer to a well-parallelized Irregular algorithm, independent of the parallel programming method of his choice.

The concept underlying our pattern language was built on a hierarchical design sequence, depicted in Figure 3, which captures the reasoning steps a programmer takes to parallelize an algorithm, whether it is irregular or not.

4.1 Parallelization steps

In the ***algorithm selection*** stage, the programmer must carefully consider the existing algorithmic solutions for the problem at hand. Considerations towards the algorithms suitability for parallelization require the programmer to be aware that:

- Some algorithms are inherently sequential or are harder to parallelize.
- The parallel programming technology used might influence algorithm choice.

However, the choice of an algorithm might not be a final decision. The design of an algorithm is in essence an iterative cycle of design decisions. Thus, latter stages of the parallelization might influence the programmer to choose a different algorithm.

Having a clearly defined problem and the solution represented by the algorithm, the programmer should concern himself with ***Finding Parallelism Opportunities,*** i.e. analyzing the problem domain and the solution to identify which elements will be able to be parallelized and how they interact with the other elements in the domain. It is at this stage that we should take into account the irregularity of the algorithm.

Exploiting Parallelism entails implementing the algorithm and data structures for efficient parallel execution. The algorithm must be expressed as a program in a way that does not obscure the inherent parallelism that was found in the previous step. In addition, the programmer must consider the best data structure to supporting parallel execution.

As no efficient parallelization is completely devoid of architecture considerations, the programmer must now consider the best ***Mapping*** from the abstract algorithm definition to the actual hardware architecture.

The final step consists on ***Optimizing*** parallel execution to achieve the best possible algorithmic performance.

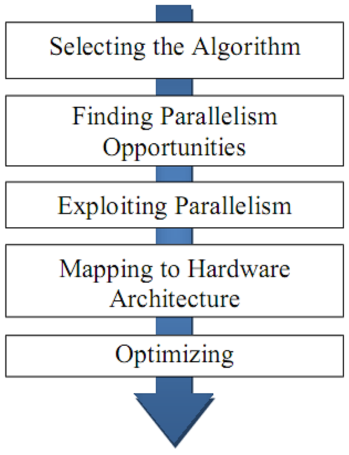
Selecting the Algorithm
Finding Parallelism Opportunities
Exploiting Parallelism
Mapping to Hardware Architecture
Optimizing

Figure 3 – Algorithm parallelization design sequence. Each stage might invalidate the solution, forcing the developer to reconsider previous stages.

Our pattern language follows this algorithm design sequence and presents a solution for the parallelization of *workpool-based irregular algorithms,* using *optimistic techniques* [22].

Figure 4 presents an overview of the dependencies and relationships among patterns. The pattern language is comprised of six design spaces that represent pattern hierarchy and structure:

1. ***Parallelism Structure***: the patterns in this design space intend to help identify what can be parallelized, i.e. what is the type of parallelism.
2. ***Data Structure***: the base data model which is used as the target of parallel operations.
3. ***Execution Structure:*** directs readers on how to drive parallel execution to take advantage of latent parallelism.
4. ***Program Structure:*** Parallel execution is an addition to the tradition of *"program = structure of algorithm + data structure"* [67] and thus this design space exists as a

composition of the ***Parallelism, Data*** and ***Execution*** structures.

5. ***Task Mapping:*** considers the scheduling of activities to the processing elements, such as processors or threads.
6. ***Data Mapping:*** considers data distribution to maximize non-concurrent access to data.

4.2 Pattern-specific Terminology

To ensure that the pattern language can be used for a wide range of irregular algorithms, we next introduce an abstract terminology for talking about irregular algorithms. This is intended to free the reader from algorithm and implementation-specific jargon and granting the reader the ability to view specific patterns individually, without the need to scrutinize the entire language.

The terminology is as follows:

Active element – An *active element* is a well-identified, describable unit of data that can be individualized from the generality. Data elements are algorithm-specific and are recognized as an often repeated name whose meaning is associated with the algorithmic metaphor. For instance, while iterating, an active element is the next element to be returned.

Neighborhood – Is the set of elements that might be read or written while an *active element* is being computed. For instance, considering matrix multiplication, the neighborhood of a result cell is the row of the first matrix and column of the second that match the cell's index.

Operator – An operator represents the operation performed on a piece of data. Algorithm operators can have either *read* or *write* semantics: *Reader* operators do not influence the data dependency set; Writer operators can be further classified as *Morph operators*, which add or remove elements from the neighborhood, or as *Local Computation Operators*, which may update the value of an element not changing the structure of the neighborhood [53].

Available parallelism – Is a measure of the amount of parallelism available represented by the maximum number of independent

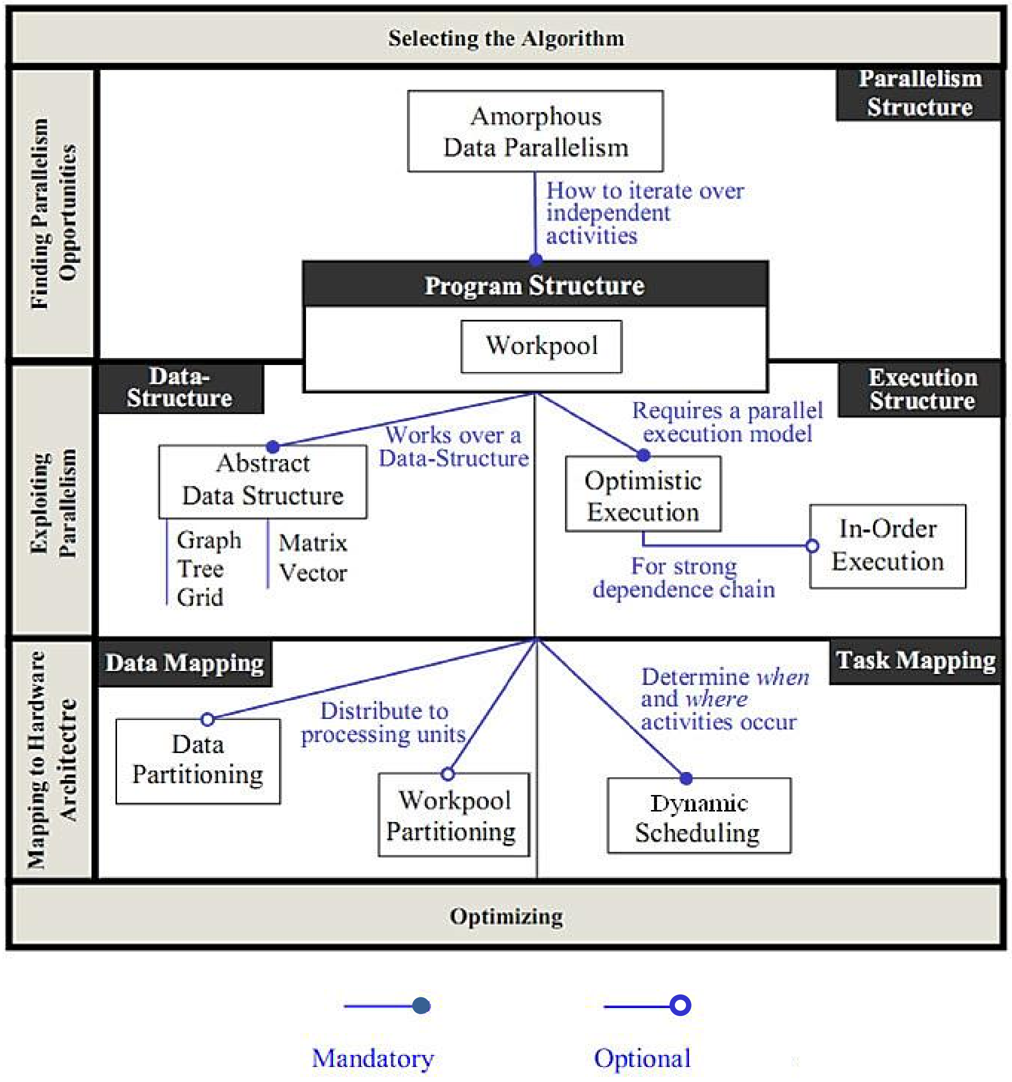

Figure 4 – Pattern Language Overview

parallel computations that can be effectively performed at a given moment.

Processing Unit (PU) – It represents either a processor core or thread, depending on the technology being used. Since our patterns consider both multi-core and multi-threaded environments this allows us to abstract from the concrete processing technology used.

4.3 Finding Parallelism Opportunities

In this section we describe the two main patterns: *Amorphous Data Parallelism* and *Workpool*. The intent of these patterns is to clarify the structure of parallelism of irregular algorithms and the overall program structure for exploiting said parallelism.

4.3.1 Amorphous Data Parallelism

Problem: How to exploit concurrency in the presence of unpredictable data dependencies.

Design space: Parallelism Structure

Context: Traditional data parallelism [29] exploits the decomposition of data structures to attain concurrent behavior, 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. This is only possible because "regular" Data-Parallelism is derived from parallel computation of iterative algorithms operating over static data structures, such as dense matrices, using index-based references.

On irregular algorithms, the nature of data dependencies is unpredictable as well as dynamic, because algorithms operate over dynamic data structures, such as graphs and trees. Thus, the amount of parallelism that can be achieved varies according to how the algorithm changes its data dependencies, hence the term "amorphous".

If the programmer is faced with an algorithm whose data structure is dynamic and the execution of an activity on an active element requires access to other elements, then he is in the presence of an Irregular algorithm, whose parallelization requires amorphous data-parallelism. This is more true if the set of elements accessed varies at runtime (i.e. if dependencies vary).

Forces

- **Overheads of Parallelization:** The benefits of parallel execution may be compromised by the overheads of parallel execution such as synchronization costs.
- **Synchronization Costs:** Coarse-grained locking has less overhead than fine-grained locking but may also exploit less parallelism.
- **Sequential to Parallel Traceability:** Clear mapping between the sequential and parallel version of an algorithm.

Solution: Exploiting data parallelism entails understanding how concurrent behavior will influence the structure of the data and how to ensure independence of computations in the overall parallelization strategy. In *Amorphous Data Parallelism*, available opportunities for concurrency-free parallelism cannot be easily predicted. Thus, to extract the maximum amount of parallelism the programmer needs to:

1. Define the *active elements* that comprise the algorithm.
2. Express computations in terms of the data structure elements.
3. Identify, at each step, which active elements are independent and which need mutual exclusion mechanisms.
4. Apply the computations iteratively to each active element.

Furthermore, an amorphous *data-parallel* decomposition should ensure that new opportunities for independent parallel execution are driven by data-dependent computations.

Galois Implementation: In the Galois implementation of this pattern the primary considerations are:

- The main loops of the algorithm must be refactored to use Galois constructs, such as the foreach loop;
- Locking is implicitly handled by the Galois Runtime.
- Galois tries to ensure a transparent traceability from sequential to parallel.

Example: Using the example of *Delaunay Triangulation*, the underlying problem in this algorithm can be parallelized in an *amorphous data-parallel* manner by considering each new point as the active element. As each new point is added to the mesh, the set of data-dependencies changes and so does the number of independently executing active elements. *Active elements* are independent if their *neighborhoods* do not overlap. The neighborhood in this case is the surrounding triangle, which can only be modified by a single *active element*.

Related Patterns

- ***Data Decomposition:*** *Amorphous Data Parallelism* is a form of *Data-parallelism* [29] or *Data Decomposition* [46].

Known Uses: The Amorphous Data-Parallel structure of irregular algorithms was first described by Kulkarni [33]. More recently, the concept of *Amorphous Data Parallelism* was used by Chorus [43], a high-level parallel programming model for irregular applications. We know of no other classification of this type of parallelism.

4.3.2 Workpool

Also Known As: Worklist, Workset

Problem: How to take advantage of parallelism when activities have runtime-dependent effects on data and active elements are created dynamically.

Design space: Program Structure

Context: When the set of dependencies varies at runtime in a way that is not statically predictable, using traditional data-driven approaches, such as data-parallelism and divide-and-conquer strategies, will not allow the programmer to extract all available parallelism. Considering *Amorphous Data Parallelism*, we must enforce a computational strategy that allows us to exploit both data and computationally dynamic dependencies.

Forces

- **Shared vs. Distributed Workpool:** shared workpools, which are globally accessible by multiple PUs, are easier to develop but might become a bottleneck from communication and concurrency overheads from access to the shared pool. Distributed workpools are decentralized, trading the bottleneck concern for a higher development cost.
- **Centralized vs. Decentralized Workpool:** A centralized workpool needs to ensure mutual exclusion, which might hinder performance. Decentralized workpools are faster but may require work-stealing methods for efficient work distribution.
- **Differentiated work:** on some algorithms, it might be useful to have more than one type of work element.

Solution: The solution is to use a *workpool*, a data structure that holds units of work or tasks. A workpool is meant to be iterated by PUs in a synchronized way. PUs then retrieve tasks and process them concurrently with other PUs. Thus, parallelism and concurrency concerns are moved from the choice of what work remains (which is dynamic) to the actual execution of activities.

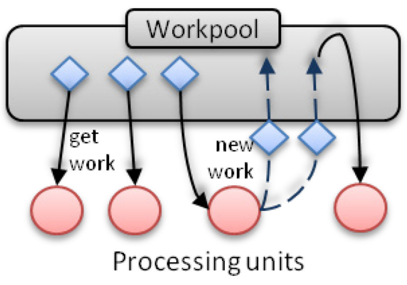

Figure 5 – Workpool model

Using a *Workpool* model entails doing the following:

1. Define work elements, consistent with the data element determined in *Amorphous Data Parallelism*.
2. Determine workpool ordering (unordered, FIFO, priority…)
3. Structure atomic mechanisms around workpool accesses.
4. Define a termination algorithm.

Processing a task may create new work units that are dynamically added to the workpool. Policies such as pseudo-LIFO, FIFO, random, chunked, etc. may be used to assign work dynamically to PU's to promote locality and reduce overheads in accessing the workpool.

While rather similar to a sequential iteration over a data structure, this parallel model requires a *termination algorithm* to ensure that the algorithm only terminates when every processing unit is idle and no more work is available on the workpool.

The general solution of this pattern is clarified with the code described in Figure 6. A loop iteratively retrieves work until the workpool is empty and every processing unit has finished computing the work it previously acquired.

Galois Implementation: The Galois framework is directed at workpool implementations of irregular algorithms, since this is the ideal way to explore available *Amorphous Data Parallelism* in this type of algorithms. Galois work elements are termed *Iterations*. The framework provides *set iterators* that act as a data-driven workpool, allowing the algorithm to concurrently iterate over a set of *Iterations*. Execution begins when a master thread starts sequential execution of the code. When it enters the Iterator construct, the thread invokes worker threads to perform the Iterations concurrently. The assignment of active elements to worker threads is the responsibility a run-time scheduler.

The workpool form of an amorphous data-parallel Delaunay Triangulation is illustrated in Figure 7.

```
Workpool wp =//initialize workpool
while(algorithmRunning) do
  atomic {
    work =workpool.getWork()
  }

  if(work == null){
    do send termination to all PUs
    if(all PUs terminated){
      algorithmRunning = false;
    }
    else wait for more work or termination
  }else {
  result = processCurrentWork(work);

  if(result produced more work)
    atomic {
      workpool.addWork(result.work);
    }
  }
endWhile
```

Figure 6 – Pseudo-code of a general workpool implementation

```
TriangleMesh mesh = //initialize mesh
Workpool wp =//initialize workpool
while(algorithmRunning) do
  atomic {
    work =workpool.getWork()
  }

  if(work == null){
    do send termination to all PUs
    if(all PUs terminated){
       algorithmRunning = false;
    }
    else wait for more work or termination
  }else {

    if (work.isPoint()) {
       Triangle tri;
       tri = mesh.surroundingTriangles(work)
       result = triangulate(tri);
    }else its an invalid triangle
       result = flip(work.getInvalid());

    if(result produced more work)
       atomic {
          workpool.addWork(result.work);
       }
  }
endWhile
```

Figure 7 – Pseudo-code of Workpool for an amorphous data-parallel Delaunay Triangulation

Related Patterns

- ***Amorphous Data Parallelism***: active elements from amorphous data parallelism represent work elements in the workpool.
- ***Workpool Partitioning***: If working on a distributed environment, the workpool can be partitioned.

Known Uses: The term *workpool* was probably first used in 1989 by Lim and Johnson [40] although they refer that a similar concept had already been described by Dally [10]. A widely known use of the *Workpool* pattern is the Linda [20] model, in which processes cooperate through a shared global data space named tuple space. Since data elements are added to the tuple space, allowing other processes to access and execute it, the model functions like a *Workpool*. In [21], the authors describe a parallel *Branch and Bound* algorithm based on a pool of nodes. A workpool model for parallel programming was also devised in [30], to work independently in distributed or shared memory in a Declarative Imperative Parallel Programming model (DIPP).

4.4 Exploiting Parallelism

This section presents patterns that illustrate how to exploit latent parallelism in irregular algorithms more fully. This entails considering how to structure both data and execution. To solve the former, we present the Abstract Data Structure pattern, while the latter is addressed by patterns *Optimistic Execution* and *In-Order Execution*.

4.4.1 Abstract Data Structure

Problem: As the choice of data structure influences the complexity of algorithm design and execution, choosing an appropriate data structure is essential.

Design space: *Data Structure*

Context: When implementing an algorithm, much effort lies in deciding on the best data structure to represent data and what characteristics make it suitable for exploiting parallelism. Although data structure design is a well-researched field, we tend to view them as black boxes around which the algorithm is shaped. The choice of topology and structural representation of data affects algorithm design and therefore, prior to implementation, all aspects of data accesses and the structure of data should be subject to careful consideration and planning.

On designing a parallel algorithm, using an Abstract Data Structure requires the programmer to build mechanisms to encapsulate not only the usual behavior of the data structure but also concurrency and synchronization. If the programmer knows the set of operations defined by the Abstract Data Structure are safe, with regards to concurrency and synchronization, he can safely use it.

For Irregular algorithms, this is simplified as the set of operations available are directly related to the operators defined in the Terminology section (4.2).

```
Interface DataElement{
    getValue():Value
    setValue(Value)
    getPosition():Position
    setPosition(Position)
    getIndex():int
    setIndex(index)
    setNeighbor(DataElement)
    setNeighbor(DataElement,index)
    setNeighbors(Collection)
    getNeighbor():DataElement
    getNeighbor(index):DataElement
    getNeighbors():Collection
    removeNeighbor(Neighbor):DataElement
    removeNeighbor(index):DataElement
    isNeighbor(Neighbor):boolean
}

Interface DataStructure{
    getElementAt(position):DataElement
    setElementAt(DataElement,position)
    hasElement(DataElement):Boolean
    removeElementAt(position):DataElement
    removeElement(DataElement):Boolean
    iterate(Strategy):Iterator
}
```

Figure 8 – Structure of an Abstract Data Structure for Irregular Algorithms

Forces

- **Obliviousness vs. full knowledge:** from a programmer's point of view, using an Abstract Data Structure removes the complexity of understanding how data is accessed and stored. This in turn increases the ease with which the programmer conceptualizes the solution. However, in a parallel environment, knowledge about the concrete data structure used is important for performance. This is ever more true if using data structures from libraries. Also, algorithm-specific characteristics might influence data structure choice
- ***Iteration***: When iterating over the Data Structure, the traversal approach is of utmost importance as different algorithms have different optimal iteration strategies.
- ***Reusability***: Using an Abstract Data Structure allows programmers to change the data structure without changing the algorithm.

Solution: In order for a programmer to use an Abstract Data Structure safely and effectively in a concurrent environment he is required to:

1. Define the data type that is to be stored in the data structure. Depending on the algorithm, this can be a simple element like a string or a more complex element like a graph.
2. Define which operations need to be available to support the execution of the algorithm. In the object-oriented paradigm, this is equivalent to defining and interface which is implemented by the concrete data structure.
    2.1. Define the operations required by the data element, such as support for accessing the stored value, element positioning and indexing values (see Figure 8, the DataElement Interface). Apart from these setters and getters, which provide data access and reference, the element should also provide support for accessing the neighbors of said element. Recall that a neighbor is any data element that might be read or written when the current data element is active (see section 4.2).
    2.2. Define the operations that the data structure should provide. These are methods intended for initialization,

search and traversal of the Data Structure. (see Figure 8, the DataStructure Interface).

3. Identify the best concrete data structure with which to realize the Abstract Data Structure. This step requires the programmer to clearly understand the tradeoffs of each concrete data structure and how the algorithm influences these tradeoffs. As defined in the Terminology section (4.2), we classify algorithm operators as Morphs, Local Computations and Readers. The type of operator used by the algorithm determines if and how the structure of data is modified and how each data element is accessed. Additional insight on the tradeoffs of different concrete data structures is summarized in Table 1.

4. Design and build the concrete data structure, implementing the Abstract Data Structure methods.

Variants: For irregular algorithms, data structures are essentially organized around two types, according to topological characteristics and how they are instantiated:

- ***Pointer-based Data Structures*** have the general form of a Graph, where edges represent pointers to other nodes. Graphs can be refined into Trees, special graphs without cycles and with a root node, and Grids, where every node connects to four other nodes. Grids can also form cubes (in three-dimensional space) or hypercubes (above 3D space). From the viewpoint of irregular algorithms, we see that there is a tendency for graph-based implementations with dynamic restructuring of nodes and edges at runtime. This makes irregular graph algorithms ideal subjects for *Amorphous Data Parallelism.*
- ***Array-based Data Structures*** have the general form of a Matrix. If a Matrix has 1xN index space, it represents a vector. Matrices with additional dimensions can be used to represent cubes (NxMxT) and hypercubes (NxMx… xT). For irregular algorithms, due to their dynamic nature, matrices are usually not recommended. However, there are exceptions which might merit the usage of matrices, as they are easier to implement. Linear and Partial Differential Equation solvers for instance, are important HPC research algorithms which take the form of matrices and are nonetheless Irregular (with local computation operators) [54]. Matrix representations can also be used for algorithms with index-based references and static neighborhoods.

The above two variant data structure types can be used to implement one another – graphs are usually implemented using adjacency lists or matrices; sparse-matrices minimize the number of elements in memory if implemented as graphs. Figure 9 sums up this equivalence.

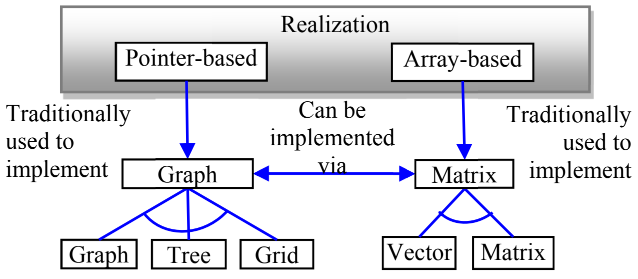

Figure 9 – Realization of abstract data structures

Galois Implementation: The Galois framework is graph-oriented and as such, supplies a few graph-based structures designed to support the aforementioned graph characteristics. These data structures are implemented around a Graph interface, providing support for directed and undirected graphs, as well as complex, simple and indexed edges**.** These can be further refined through polymorphic extension.

Table 1 – Characteristics of data structures

	Matrix	Graph
Traversal	Ability to index elements with constant time random access;	Iteration strategies vary; Nodes can only traverse to neighbor nodes;
Concurrency Control	Locked entirely or in blocks of elements;	Locks on individual nodes;
Partitioning	Straight forward in most cases;	Always requires traversal
Memory Allocation	Memory allocated as contiguous space; Good locality of references;	Per-element random memory allocation; Bad locality of references;
Growth	Predefined size; Additional elements require re-allocation and copy.	New elements are added/removed by adding/removing pointers;

For instance, for performance reasons, the Galois team implemented Compressed Row Format variations of the framework's graphs, thus storing graphs as matrices. This is shown to, in some cases, considerably improve algorithm performance, even on a dynamic Irregular algorithm as Metis[63].

Example: We can implement Kruskal's MST as a graph algorithm and, traditionally, that is the best way to proceed. However, with a small adjustment, it can be implemented with a matrix-like data structure. This is possible because the graph is undirected, which introduces sparsity in the matrix, allowing Kruskal's to be mapped to a triangular matrix without compromising efficiency (see Figure 10). In this case, we have a Graph with a Kruskal MST (a) and its matrix representation (b). As shown in Figure 10, the matrix representation can be made in terms of a triangular matrix and all remaining values can be zero. This yields a fast and simple structure for Kruskal's MST.

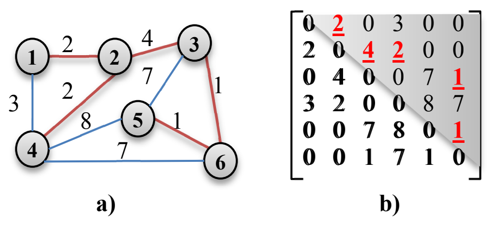

Figure 10 – Graph to Matrix mapping

The same can be applied to the Delaunay triangulation algorithm, since it also presents and high degree of sparsity. However, because the nature of the mesh is dynamic, representing the mesh as a matrix would instead be deterimental to the algorithm's performance.

Related Patterns

- *Amorphous Data parallelism:* This pattern is used to provide an underlying representation to the data required by the *Amorphous Data Parallelism* pattern.

Known uses: There are many and varied implementations of graph and matrix based data structures[15]. The earliest discussion on adjacency matrixes goes back to the 60's [31], where matrices where used to represent electrical circuits.

4.4.2 Optimistic Execution

Also Known As: Data-Driven Speculation, Speculative Execution

Problem: How to parallelize the execution of an amorphous data-parallel algorithm.

Design space: Execution Structure

Context: Parallelizing irregular algorithms is a difficult task because these are characterized by chains of inter-dependent computations. In these cases, static analysis techniques – such as points-to and shape analysis – and semi-static approaches – based on the inspector-executor model – cannot fully uncover potential parallelism, while coarse-grained locking implementations restrict the amount of available parallelism.

Forces

- **Implementation Cost vs. Benefit:** The cost of handling *Optimistic Execution* might not merit the benefit.
- **Available Parallelism vs. Number of Conflicts:** The maximum number of simultaneous independent computations should be used to define an optimal grain size.
- **Grain of Parallelism vs. Cost of Locking:** Executing multiple fine-grained computations might be computationally worse than executing a single coarser one.
- **Grain of Parallelism vs. Cost of Miss-speculation:** The cost of mis-speculation increases as the grain coarsens, as does the amount of wasted work due to rollbacks.

Solution: To allow for efficient implementations of irregular algorithms, parallelization using speculative or optimistic parallelization techniques should be adopted [33].

Speculative execution of *Amorphous Data Parallelism* implies executing the algorithm without full knowledge of how data dependencies change at runtime. Therefore, execution assumes that there is no concurrent access to data elements. The system must continually check for access violations and take appropriate corrective actions. When no violations are detected, the results can be committed and the data structure updated.

To execute an irregular algorithm optimistically, the implementation must identify active elements and handle conflicts between activities. Active elements derive from *Amorphous Data Parallelism* while conflicts depend on the computational operators and the data structures. Therefore, we must:

1. Determine how to check for neighborhood violations, and
2. Introduce transactional semantics for all data structures by using commit mechanisms and rollback operations.

Galois Implementation: Galois is an object-oriented optimistic parallelization framework for irregular algorithms and therefore, has built-in support for *Optimistic Execution*. These are provided via the three main aspects of the framework: (1) a simple yet powerful set of programming constructs that help non-expert programmers express key properties of irregular algorithms, (2) a library of concurrent data structures, and (3) a runtime system that uses optimistic parallel execution.

Galois' library provides the data structures and shared object implementations. It is the responsibility of the library and runtime system to ensure that set iterators retain sequential semantics while being optimistically executed. Library classes are also responsible for deciding which operations represent access violations and which do not. This is introduced through semantic commutativity [21]. The library also provides rollback functionality, ensured by inverse method semantics, i.e. each method that updates data has an inverse method that undoes its action.

The runtime system is responsible for checking commutativity constraints as well as enforcing rollback operations. An arbitrator checks the method's commutativity against all other executing methods. If the method commutes, there is no race condition and execution can proceed. Otherwise, the activity must rollback.

Example: For Delaunay Triangulation, an optimistic implementation would attempt to insert points concurrently using some number of Processing Units (PUs). If a PU tries to access data that is already locked by another unit, an access violation exception is thrown and caught. Otherwise the result is allowed to commit. The pseudocode for this implementation is depicted in Figure 11.

```
TriangleMesh mesh = //initialize mesh
Workpool wp =//initialize workpool
while(algorithmRunning) do
  atomic {
    work =workpool.getWork()
  }

  if(work == null){
    do send termination to all PUs
    if(all PUs terminated){
      algorithmRunning = false;
    }
    else wait until more work or termination
  }else {
    try {// optimistic execution starts
     if (work.isPoint()) {
       Triangle tri;
       tri = mesh.surroundingTriangles(work)
       result = triangulate(tri);
     }else it's an invalid triangle
       result = flip(work.getInvalid());

     if(result produced more work)
       atomic {
         workpool.addWork(result.work);
       }
    } catch (violationException ve){
       //do nothing
       //graph is only updated on commit
     }
  }
endWhile
```

Figure 11 – Pseudo-code of Optimistic implementation of Delaunay Triangulation.

Related Patterns

- ***Amorphous Data Parallelism:*** *Optimistic Execution* focuses parallel execution on independent active elements.
- ***In-Order Execution***: If the algorithm enforces a strict dependency chain, then the *In-Order Execution* pattern should be used.
- ***Workpool:*** Rolled back work can easily be re-added to the workpool until it can be safely executed.

Known uses: Optimistic parallelization techniques were introduced in the 70s as a form of branch speculation [64, 16]. In 1985, the Time Warp mechanism for the synchronization of discrete-event simulation in distributed systems was introduced by Jefferson [26]. More recently, other speculative techniques have been introduced, such as loop-based speculation [55, 22] and speculative multithreading [61, 45]. The latter enables optimistically created threads by tracking memory accesses made by loop iterations and has been introduced to a significant number of parallelization architectures [8, 51, 44]. Additionally, a wide variety of optimistic parallel implementations of irregular algorithms has been proposed by the parallel programming community [65, 34, 7] .

4.4.3 In-Order Execution

Also Known As: Ordered execution

Problem: How to increase the amount of available parallelism in algorithms with tight data dependency chains.

Design space: Execution Structure

Context: Some irregular algorithms have strict dependency chains that presuppose some type of ordering of execution. This strictness not only influences the result but also constrains correctness. On *Event-driven simulation* [11], for example, since events need to be globally ordered, using unconstrained *Optimistic Execution* would lead to a significant waste of parallelization opportunities due to frequent rollbacks. An algorithm is said to have a restricted chain of dependency when computations require the output of previous computations or when there are explicit ordering constraints, such as alphabetical or numerical order.

Forces

- **Amount of constraints vs. benefit**: If the dependency chain is restricted to a small number of iterations, then the cost of introducing *In-Order Execution* might not merit the benefit.
- **Order of rollback:** If a higher priority iteration rolls back due to a conflict with a lower priority Iteration, the algorithm would stop progressing and eventually deadlock.
- **Size of data set:** The bigger the data set, the more opportunities for independent execution exist.

Solution: The solution is to ensure that speculative execution is restricted to the order enforced by the dependency chain. Thus, speculative activities should only commit results to the data structure when all preceding activities have done so. The precedence is given by the dependency chain and is algorithm-specific.

To achieve the maximum amount of parallelism, activities must always be executed speculatively as in *Optimistic Execution.* However, they must always commit in the order they were meant to be executed, giving a deterministic characteristic to the parallel execution. This ensures that no higher priority iteration ever needs to rollback because some lower priority iteration, with whom a conflict is detected, has already committed.

To implement in-order committal we need to introduce a scheduling mechanism to keep track of which iterations are queued to be committed and what is their priority. Since iterations aren't allowed to commit if higher priority iterations exist in the scheduler, we ensure that lower priority conflicting iterations always abort and high priority iterations always commit, if valid.

Example: Kruskal's MST is a typical in-order algorithm, as edges need to be added to the MST from lower to higher weight. In this algorithm, any two edges are independent if they don't have any node in common. Independent edges can be executed concurrently if their weight is less than or equal to any other edges waiting to be processed and if the addition of both edges to the MST doesn't create a cycle. However, the possibility of creating a cycle is not explicitly handled. Instead, if a cycle is created, the operation that generated the cycle is aborted, by optimistic execution, and rolled back.

Implementation of Kruskal's by *In-Order Optimistic Execution* is shown in Figure 12**.**

Another good example is provided by *Lamport clocks* [38]. The definition of a causal order of events requires a global ordering. However, this is only enforced for events that span multiple processes. Events occurring on the same process are only required

```
Graph graph = //initial graph;
Thread worker;       //worker thread
InOrderScheduler scheduler;
Worklist worklist;
Worklist=//edges from graph ordered by weight
MST mst; //minimum spanning tree;

while(algorithmRunning) do
  atomic {
    work =workpool.getWork()
  }
  if(work == null){
    do send termination to all PUs
    if(all PUs terminated){
      algorithmRunning = false;
    }
    else wait until more work or termination
  }else {
    try {// optimistic execution starts
      Node n1 =work.getInNode();
      Node n2 =work.getOutNode();
      if(n1 and n2 arent connected in the MST)
        result=mst.add(edge);

      //Commit this Iteration if top priority
      //Else commit highest priority element
      scheduler.commitInOrder(result);

    } catch (violationException ve){
    //do nothing
    //graph is only updated on commit
     }
  }
endWhile
```

Figure 12 – Pseudo-code of In-Order Kruskal MST

to enforce local order and can occur concurrently with other local order events on other processors.

Related Patterns

- ***Optimistic Execution***: *In-Order Execution* is a specific case of *Optimistic Execution*.

Known uses: Out-of-order execution is analogous to in-order iterations, where speculative execution of processor instructions reduces the time required in future instructions [24]. Speculative parallelization Do-loops in X10 provide similar results via hardware transactional memory [66]. Safe futures may also be used to allow speculative ordered execution [50].

4.5 Mapping to Hardware Architecture

This section provides intuition on the design impact produced by different hardware architecture configurations. We introduce two patterns, *Data Partitioning* and *Workpool Partitioning*, that concerns the mapping of data to the memory model, and a *Dynamic Scheduling Pattern,* which concerns with the mapping of execution to the number of processing units.

4.5.1 Data Partitioning

Problem: To effectively parallelize an algorithm across multiple processing units the programmer must break data into small, manageable blocks, i.e. partitions, promoting locality and reducing synchronization costs.

Design space: *Data Mapping*

Context: In order to efficiently parallelize algorithms in multi/many-core environments it becomes essential to separate as much as possible the number of shared resources, while at the same time taking advantage of multiple PUs and maximizing data and task locality.

Partitioning therefore becomes a key factor for large, complex algorithms with stringent performance requirements.

Forces

- **Partition size vs. independence**: Larger partitions decrease the likelihood that neighborhoods overlap multiple partitions, but this may reduce concurrency if all active elements within a given partition are processed by the same PU.
- **Partition size:** Smaller partitions and in greater number than that of PUs allow for better distribution of work.
- **Cost of dynamic partitioning:** The overhead of constant repartitioning might reduce the benefit.
- **Underlying data structure:** Partitioning should be handled in an efficient way, avoiding computational costs as much as possible.
- **Partition Data:** Each partition should ideally be comprised of data elements that share common traits and have tighter dependencies with data in the same partition than with others.
- **Data Structure Obliviousness:** The user should be unaware of the actual data structure being partitioned, i.e. partitioning should have a similar effect independently of the actual data structure.

Solution: A partitioned environment needs to reduce the cost of accessing shared elements, thus reducing synchronization To reduce the amount of concurrent access, the workpool should differentiate work items according to their assigned partition, thus avoiding having to decide which work elements go to which PU. To achieve this, the programmer must:

1. Define the number of partitions as a function of the number of processing units ($\boldsymbol{N\ Part} = \boldsymbol{nPU}, \boldsymbol{n} \in \mathbb{Z}^{+}$)
2. Determine the type of partitioning required by the algorithm: Static or Dynamic
3. Choose a partitioning algorithm.
4. Handle *Amorphous Data Parallelism*.
   4.1. Decide the granularity of synchronization: i.e., whether locks are associated with data structure elements or with entire partitions.
   4.2. Decide how to handle neighborhoods that span multiple partitions.
   4.3. Determine the update strategy, i.e., to which partition should newly created data structure elements be added.

Galois Implementation: One of the most important ideas behind data partitioning in Galois is that the client code should not need to change radically when instantiating data partitioning. To enable dynamic load-balancing, data structures are over-decomposed so that each PU has multiple partitions to work on and can steal partitions from other PUs if it runs out of work.

Partitionable graphs implement the *Partitionable* interface. Nodes and edges in partition graphs must implement the *PartitionObject* interface, which allows the programmer to access information about the partition to which the object belongs. Additionally, partitioning a graph entails assigning a Partitioner to the graph class. Galois currently supports *Graph Bisection*, where the graph is traversed breadth-first from an arbitrary boundary node until half the nodes have been traversed, and one based on *Metis* [28].

Example: In Delaunay Triangulation partitions can't be statically determined, as the mesh is generated dynamically. Therefore dynamic methods are required to efficiently partition and distribute data. The usual solution is to start with a single partition and as the number of data elements increases, the data-set is repartitioned and distributed to the PUs. As each partition is assigned to a single PU, the graph can be seen in a more abstract way as if dependencies between nodes (Figure 13-a) were in fact dependencies between PUs (Figure 13-b).

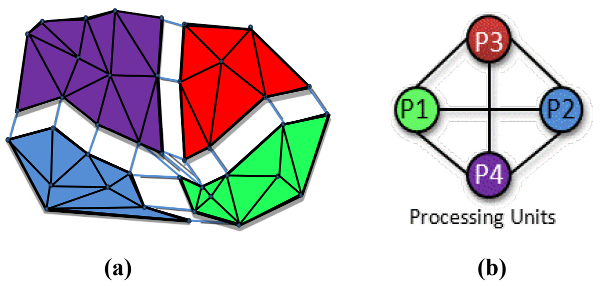

Figure 13 – Partitioned Delaunay mesh

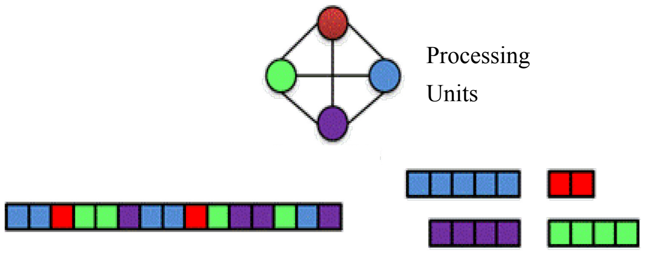

Figure 14 – Workpool partitioning

As regards Kruskal's MST, since the goal is to produce a sub-graph of a pre-existing graph, we can statically partition the input graph per the PUs. Thus, the partitioning algorithm can have a higher computational cost, providing optimal distribution of nodes per PUs. However, we need to pay special attention to bordering nodes and the possibility of cycles. When a border node is added to the MST in a partition, we can only know if a cycle is created if the partitions exchange MST information among themselves.

Related Patterns

- ***Geometric Decomposition:*** decomposes data structure based on its geometric properties, for distribution purposes [46].
- ***Workpool Partitioning:*** The workpool can be partitioned so as to mimic the partition of the data structure.

Known Uses: There are many algorithms available for graph partitioning. Some studies of partitioning methods are well known to the parallel programming community: Karypis and Kumar [28] provide an analysis of current partitioning techniques for irregular algorithms; Wider surveys of graph partitioning algorithms are described by Fjallstrom [17] and Elsner [14].

The concept of supporting partitioning in languages and frameworks is around since the Ada language [27]. Recent approaches to high performance computing, such as High Performance Fortran (HPF) [42], Threaded Building Blocks (TBB) [56] or Chapel [12], also provide partitioning strategies. HPF focuses on the partitioning of arrays to distributed memory computers, while TBB only supports static partitioning with work stealing. Chapel belongs to a group of Partitioned Global Address Space (PGAS) languages which have a partitioned memory model [48]. On these languages, a data structure is accessed as if it were local though it is in fact distributed. Chapel supports traditional data distributions as part of its class library and allows programmers to implement application specific distributions if needed.

4.5.2 Workpool Partitioning

Problem: How to minimize the cost of accessing the *Workpool*.

Design space: Data Mapping

Context: An efficient distribution of an algorithm is not guaranteed simply by partitioning data structures. To achieve a proper division of work per processing unit, programmers should reduce access to non-partitioned data structures, which create synchronization bottlenecks and reduce concurrency. On workpool-based irregular parallel algorithms, this bottleneck derives from synchronized accesses to the workpool. Its high cost can severely degrade the performance of the parallel algorithm.

Forces

- ***Communication vs. Computation:*** If the workpool remains centralized, the scheduler needs to be aware of which partition the work is assigned. However, this increases computational cost of scheduling procedures. On distributed workpools, the scheduler has to keep track of multiple local workpools, which is communication intensive.
- ***Work Distribution:*** Due to the dynamic characteristics of Irregular Algorithms, the amount of work produced by partitions might be unbalanced, thus requiring work-stealing or work-sharing methods for efficient work distribution.

Solution: Not having to decide which work needs to be processed by which partition reduces the amount of synchronization needed, thus reducing the cost of accessing the shared workpool. The workpool should be partitioned to reflect the partition of the data structure, enhancing the locality of the algorithms and reducing the need for synchronization.

The solution however is non-trivial as it depends on the specific characteristics of the algorithm being implemented.

The overall solution is as follows:

1. Define the number of partitions as the exact number of processing units.
2. Define how work elements are assigned to each processing unit.
   - 2.1. Define how work elements will be marked as belonging to a PU.
   - 2.2. Determine which partition gets each new work element. When a PU adds work to the workpool, partitioning ensures that the same PU will eventually process the work it produced.
3. Determine how to move work elements to and from the PU.
   - 3.1. Use an asynchronous fetch mechanism to get work from the PUs and add it to the workpool. However, this mechanism needs to prioritize PUs, reducing the possibility of starvation and maximizing parallel work.
   - 3.2. Use a synchronous mechanism for PUs to receive work elements from the workpool. PUs should not get work elements directly by accessing the workpool as this introduces high synchronization costs and cause PUs to waste resources waiting for work. Instead, work should be assigned to the PUs preemptively (see *Dynamic Scheduling*).

Variants: We can consider two main variant forms of *workpool partitioning* (see Figure 14):

Partitioned Global workpool – The workpool remains globally accessible although work elements are partition-aware, i.e. each PU only receives work for the partitions it currently holds. Having work elements marked with the partition they belong to enables the workpool to be implemented as if it were composed of *n* workpools, one for each partition. Thus, there is little access contention and the cost of performing workpool operations is reduced. This form of workpool is more effective if there is a need for a centralized management resource, whose role can be assumed by the workpool. This is the case of *In-Order Execution*, as the global workpool can localize the knowledge of the execution order, or work scheduling.

Partitioned Local workpool – The workpool itself is partitioned and each PU maintains its own local workpool. This implementation allows the PUs to increase data locality since new work produced is placed on the local workpool. This solution works better if work elements are added to the workpool dynamically, as having local work items reduces both the cost of updating the workpool and that of retrieving new work elements.

Galois Implementation: In Galois, the assignment of work to PUs is performed in a partition-sensitive manner. The workpool itself remains global. The programmer must first instruct the runtime system to recognize the different partitions. Contrary to normal workpool elements, partitionable elements know which PU currently holds the partition they belong to. Iteration Coalescing, an optimization of the Galois framework [47] adds local workpools to improve locality of references.

Example: On *Delaunay Triangulation*, if we consider that no new element is added to the workpool, then using a simple shared global workpool presents advantages because a measure of locality is offered by the workpool partitioning. In addition, using a global workpool allows the algorithm to concentrate its computational resources on triangulating the mesh, instead of coordinating the multiple workpools.

If we instead consider that every bad triangle produced when a point is added to the mesh is inserted in the workpool as a new work item, then using a local workpool provides locality advantages. Triangles that needed to be re-triangulated would be added to the workpool of the PU that originated the bad triangles.

Related Patterns

- ***Partitioning:*** the data structure must be partitioned.
- ***Dynamic Scheduling:*** scheduling can be used to preemptively assign work elements to PUs.

Known uses: A similar approach to workpool partitioning is proposed by Chandra et al [6]. Their dynamic partitioning strategy named Dispatch builds processor-local workpools, which are then used to reconstruct the global work distribution lists. A similar approach was used by Bai et al [2] to developed a software transactional memory executor that partitions transactions among processors by grouping them based on their search keys. The Chapel programming language [12] uses an asynchronous partitioned global address space programming model that provides virtual partitioning of data structures in memory spaces. This is an analogous yet different approach to partitioning. In Chapel, each processor node retrieves tasks from a task pool but can also invoke work on other processor nodes using *On* clauses. These force computations to occur in the processor node that holds the object in memory. Processor nodes can also fetch data from remote locations.

4.5.3 Dynamic Scheduling

Problem: How to dynamically assign work to processing units.

Design space: Task Mapping

Context: When considering parallel implementations of algorithms, the programmer must always take care to create an efficient mapping between the tasks (or work elements) to be executed and the processing units that will eventually execute them. Scheduling essentially entail predicting at runtime how work elements should be assigned to PUs so that it can be done preemptively, without PUs having to wait for new work elements to process. This mapping, or scheduling, has concrete effects on the algorithm's performance, essentially aiming to optimize concurrency, locality and load-balancing.

There are a multitude of scheduling techniques for static, semi-static and dynamic scheduling [39, 60]. For irregular algorithms, static and semi-static scheduling techniques fail to introduce valid and efficient schedules that would allow the algorithm to fully exploit of its potential parallelism, since dependencies are only known at runtime [33].

Forces

- ***Assignment Overhead:*** If computations are too fine-grained, the cost of scheduling might not justify the benefits.
- ***Scheduler overhead:*** There is a high computational overhead on arbitrating conflicts for strict dependencies.
- ***Know the domain:*** There needs to be a tight understanding of the neighborhood of the algorithm and how that neighborhood is influenced by computations. If the neighborhood of an active element remains the same throughout execution then the programmer should use that fact to cluster sets of closely dependent computations to be processed by a single PU. If the neighborhood is dynamic, then the mechanism to assign computations to PUs needs also be dynamic.
- ***Know the architecture:*** Knowing the underlying hardware architecture, how many and what type of cores exist and how memory is managed, allows us to understand how to best maximize the number of parallel computations. As computations are assigned to PUs dynamically, the number of active PUs may vary throughout the execution.
- ***Know the dependencies:*** Computational dependencies ultimately define the order of processing on each PU. It is essential to understand how new work elements influence the existing schedule and the locality of resources.

Solution:

To create a valid and ideal schedule configuration between computations and PUs, the programmer must:

1. Define a way to predict how distinct work elements are needed by each PU. This prediction is tuned by the programmer for each specific algorithm according to its characteristics and has essentially two forms:

    1.1. There is a well-defined computational path and each work element processed causes the PU to process work elements that access neighboring nodes. This is the case of algorithms like maxflow computations and sparse matrix computations, where processing follows a fixed-step sequential path, although that path is not known at compile time. It is worth to mention that this is not the same as *In-Order Execution*, as the order in which neighborhood elements are processed might not matter.

    With these characteristics, the schedule should try to cluster sets of closely dependent computations to be processed by the same PU.

    1.2. There is no defined computational path and scheduling can be random or it can follow the structure of data partitioning, thus taking advantage of data locality. This is best for non-deterministic algorithms like *Delaunay mesh generation* and *refinement*.

2. Reduce the number of collisions between PUs by trying not to simultaneously assign work elements to different partitions if processing those work elements will cause the PUs to access the same data. Recall that, by *Optimistic Execution,* when a collision is raised the offending PU is forced to abort and rollback, thus wasting resources.
3. Add workload balancing by defining mechanisms that will allow the scheduler to override the assignment of work elements to PUs based on execution schedule and allocate work based on load balancing concerns. There are two main reasons to allow this:
   3.1. A PU might be starving but there are work elements still left to process, although ideally those work elements should be process by another PU.
   3.2. A PU might have too much potential work queuing to be processed, while other PUs have little to no work available.

On implementing these scheduling mechanisms, the programmer should take a special care to make them as light as possible. If scheduling wastes resources then it might be best not to have scheduling. Also, to ensure correctness, the schedule achieved needs to be able to reduce to a sequential implementation, thus ensuring that all PUs have a consistent view of the system state.

Galois Implementation: The runtime system has a *scheduler* that is responsible for fetching work from the set iterators and creating optimistic parallel iterations. The scheduler depends on three scheduling functions to schedule computations to the available PUs efficiently:

- A *Clustering function* groups closely inter-dependent work items. Clusters may be of different sizes.
- A *Labeling function* maps clusters to PUs. Each cluster is assigned to a single PU but a PU can have multiple clusters. Labeling can be performed on demand, as each PU fetches work from the *Workpool*.
- An *Ordering function* finds the sequential order in which each cluster's work items are within a PU.

Figure 15 presents the conceptual scheduling mechanics of the Galois Framework.

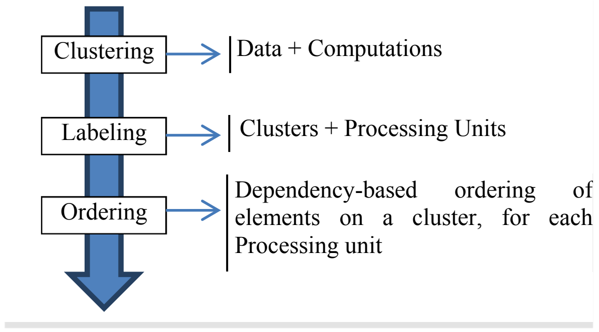
Clustering
Data + Computations
Labeling
Clusters + Processing Units
Ordering
Dependency-based ordering of elements on a cluster, for each Processing unit

Figure 15 – Scheduling in the Galois Framework

A number of preset scheduling functions are provided by Galois [35] but the programmer has the option to implement their own scheduling functions in order to adapt Galois to the specifics of the algorithm.

Example: Considering the *Delaunay Triangulation* algorithm, if at a given step in the algorithm we have an initial mesh and a given number of points that still need to be added to the mesh, a scheduling on such conditions would:

1. Cluster the remaining points according to the data partition where they will be inserted. This activity provides clustering based on interdependencies, since neighborhoods of points on the same partition have an added probability of interfering with each other. (see Figure 16);
2. As clusters are built partition-wise, they are executed by the PU that holds the data partition.
3. As can be seen in Figure 17**,** the sequential ordering of work items within a cluster (a) can be performed by using a dependency graph (b)**.**

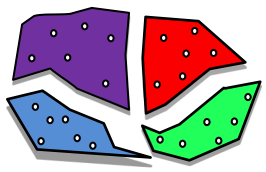

Figure 16 – Partition-wise work clustering.

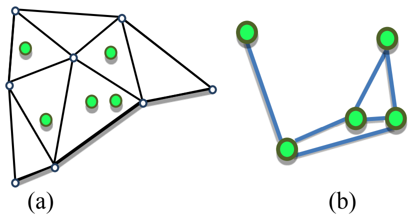

Figure 17 – Inter-cluster dependency graph

Related Patterns

- ***Partitioning:*** Partitioning might help cluster computations.

Known uses: There are myriads of scheduling techniques and algorithms for parallel processing. Programming languages such as HPF [42] and ZPL [5] Schedule computations along with data structures to improve locality. X10 allows user defined scheduling of computations to cores [66]. Gramps, a programming model for graphic pipelines uses multi-level scheduling to minimize on-chip cache support fort intermediate pipelining results [62]. Carbon [37] is a purely hardware scheduler that allows task queuing and scheduling, although it lacks customization of scheduling strategies. The dynamic scheduling of parallel computations in multiprocessor systems with identical parallel processors is tackled by Liu [41]. In its approach, dynamism in scheduling is a function of the number of available processors can vary in time.

5. RELATED WORK

The tradition of using patterns as tools for documentation and reusability was made popular by the Gang of Four design patterns. However, their book provides solutions based on object-oriented concepts such as inheritance and polymorphism and, although adaptable, each pattern presents precise classes, operations and hierarchies that the programmer should follow to achieve the intended solution [19]. Our patterns are different in that they discuss problems in terms of abstract principles and leave the task of deciding the actual implementation to the programmer, i.e. the Pattern Language presents advice and considerations about how a programmer should introduce the solution and why.

We identify three main pattern languages and catalogues focusing on parallel programming. Schmidt et al [58] present a set of patterns for concurrency and networking that does not focus on semantics and domain-dependent concepts and does in fact represent a pattern language. However, as they acknowledge, each pattern is self-contained and independently described. For this reason, we do not consider this as a fully-fledged pattern language, but rather a pattern catalogue with some inter-pattern dependencies. Their patterns represent specific parallelization constructs, while we focus on parallelization methods. It should also be noted that Schmidt et al use the JAWS web server as a basis for their patterns, similarly to Galois in our pattern language.

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 [29] and the pattern language of Mattson *et al* [46]. 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. In this paper, we instead 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 [13].

Aside from patterns, there are other approaches that describe higher level strategies for irregular algorithms: Fonlupt *et al* [18] describes a set of load balancing redistribution strategies, illustrating several algorithm formulations. Biswas et al [3] describe computing strategies in relation to specific hardware architecture. Rünger and Schwind [57] describe parallelization strategies for algorithms that contain both regular and irregular characteristics. Ansejo et al [1] present general use optimization strategies. These strategies are not as high-level as patterns but present pattern mining opportunities for future work.

6. CONCLUSIONS

This paper describes a pattern language for the parallelization of irregular algorithms. This class of algorithms is mainly used in the scientific community but not much work has been to identify and document abstractions that simplify the parallelization of such complex problems.

The patterns documented here result from a reverse engineering effort of the Galois System [34]. Other frameworks and languages have considerably different methodologies for handling irregularity. In future, we intend to explore these alternatives as well, and relate them to the patterns described here to enrich and mature the language and enhance its potential applicability to cover a broader set of techniques and methods targeting parallel irregular algorithms.

ACKNOWLEDGMENTS

We would like to thank Robert Hanmer, whose comments have been incredibly helpful and insightful. His shepherding helped us significantly improve the quality of our paper. We would also like to thank our program committee member Hironori Washizaki.

This work was partially supported by project PRIA – *Parallel Refinements for Irregular Applications* (UTAustin/CA/0056/2008) funded by Portuguese FCT/MCTES and FEDER.

REFERENCES

[1] Asenjo, R., Corbera, F., Gutiérrez, E.*, et al.*, Optimization techniques for irregular and pointer-based programs. in, (2004), 2-13.

[2] Bai, T., Shen, X., Zhang, C.*, et al.* A key-based adaptive transactional memory executor *International Parallel & Distributed Processing Symposium, IPDPS 2007*, Long Beach, CA, 2007, 1-8.

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

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

[5] Chamberlain, B., Choi, S., Lewis, E.*, et al.* ZPL: A machine independent programming language for parallel computers. *IEEE T. Software Eng.*, *26* (3). 197.

[6] Chandra, S., Parashar, M. and Ray, J. Dynamic structured partitioning for parallel scientific applications with pointwise varying workloads *Proc. 20th IEEE/ACM International Parallel and Distributed Processing Symposium*, Rhodes Island, Greece, 2006.

[7] Chrisochoides, N., Lee, C. and Lowekamp, B. Mesh generation and optimistic computation on the grid. in *Performance analysis and grid computing*, Kluwer Academic Publishers, 2004, 231-250.

[8] Codrescu, L., Wills, D. and Meindl, J. Architecture of the Atlas chip-multiprocessor: Dynamically parallelizing irregular applications. *IEEE T. Comput.*, *50* (1). 67-82.

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

[10] Dally, W. *A VLSI architecture for concurrent data structures*. Kluwer Academic Publishers, 1987.

[11] Das, S., Adaptive protocols for parallel discrete event simulation. in, (1996), IEEE Computer Society Washington, DC, USA, 186-193.

[12] Diaconescu, R. and Zima, H. An approach to data distributions in Chapel. *Int. J. High. Perform. C*, *21* (3). 313.

[13] N-Body Pattern Language, http://parlab.eecs.berkeley.edu/wiki/patterns/n-body_methods, February, 2010

[14] Elsner, U. Graph partitioning: a survey *Tech. Rep. 97-27, Technische Universität Chemnitz*, Chemnitz, Germany, 1997.

[15] Even, S. *Graph algorithms*. WH Freeman & Co. New York, NY, USA, 1979.

[16] Fisher, J.A. Very Long Instruction Word architectures and

the ELI-512 *Proc. 10th annual international symposium on Computer architecture*, ACM, Stockholm, Sweden, 1983, 140-150.

[17] Fjallstrom, P. Algorithms for graph partitioning: A survey. *Computer and Information Science*, *3* (10).

[18] Fonlupt, C., Marquet, P. and Dekeyser, J. Data-parallel load balancing strategies. *Parallel Computing*, *24* (11). 1665-1684.

[19] Gamma, E., Helm, R., Johnson, R*., et al.* Design Patterns: Elements of Reusable Object-Oriented.

[20] Gelernter, D. Generative communication in Linda. *ACM T. Progr. Lang. Sys.*, *7* (1). 80-112.

[21] Gendron, B. and Crainic, T. Parallel branch-and-bound algorithms: survey and synthesis. *Operations Research*, *42* (6). 1042-1066.

[22] Gupta, M. and Nim, R. Techniques for speculative run-time parallelization of loops *Proc. ACM/IEEE conference on Supercomputing*, IEEE Computer Society, San Jose, CA, 1998, 1-12.

[23] Gutierrez, E., Asenjo, R., Plata, O*., et al.* Automatic parallelization of irregular applications. *Parallel Comp.*, *26* (13-14). 1709-1738.

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

[25] Herlihy, M. and Moss, J., Transactional memory: Architectural support for lock-free data structures. in, (1993), ACM, 300.

[26] Jefferson, D. Virtual time. *ACM T. Progr. Lang. Sys.*, *7* (3). 425.

[27] Jha, R., Kamrad, J. and Cornhill, D. Ada program partitioning language: A notation for distributing Ada programs. *IEEE T. Software Eng.*, *15* (3). 271-280.

[28] Karypis, G. and Kumar, V. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, *20* (1). 359-392.

[29] Keutzer, K. and Mattson, T. Our Pattern Language (OPL): A Design Pattern Language for Engineering (Parallel) Software *ParaPLoP*, 2009.

[30] Knopp, J. and Reich, M., A Workpool Model for Parallel Computing. in *First International Workshop on High Level Programming Models and Supportive Environments (HIPS)*, (Honolulu, HI, USA, 1996), IEEE Computer Press.

[31] Koffman, E. and Wolfgang, P. Objects, abstraction, data structures and design using C++.

[32] Krishnan, V. and Torrellas, J. A chip-multiprocessor architecture with speculative multithreading. *Computers, IEEE Transactions on*, *48* (9). 866-880.

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

[34] Kulkarni, M., Burtscher, M., Pingali, K*., et al.*, Lonestar: A suite of parallel irregular programs. in *IEEE International Symposium on Performance Analysis of Systems and Software*, (2009), 65-76.

[35] Kulkarni, M., Carribault, P., Pingali, K*., et al.*, Scheduling strategies for optimistic parallel execution of irregular programs. in, (2008), ACM, 217-228.

[36] Kulkarni, M., Pingali, K., Walter, B*., et al.* Optimistic parallelism requires abstractions. *Commun. ACM*, *52* (9). 89-97.

[37] Kumar, S., Hughes, C.J. and Nguyen, A. Carbon: architectural support for fine-grained parallelism on chip multiprocessors *Proc. 34th International Symposium on Computer architecture*, ACM, San Diego, California, USA, 2007, 162-173.

[38] Lamport, L. Time, clocks, and the ordering of events in a distributed system. in *Commun. ACM*, ACM, 1978, 558-565.

[39] Leung, J. *Handbook of scheduling: algorithms, models, and performance analysis*. Chapman & Hall, 2004.

[40] Lim, J. and Johnson, R. The heart of object-oriented concurrent programming. *ACM SIGPLAN Notices*, *24* (4). 167.

[41] Liu, Z. Dynamic scheduling of parallel computations. *Theor. Comput. Sci.*, *246* (1-2). 239-252.

[42] Loveman, D.B. High Performance Fortran. *IEEE Parallel Distrib. Technol.*, *1* (1). 25-42.

[43] Lublinerman, R., Chaudhuri, S. and Cerny, P. Parallel programming with object assemblies *Proc. 24th ACM SIGPLAN Conference on Object Oriented programming systems languages and applications*, ACM, Orlando, Florida, USA, 2009, 61-80.

[44] Marcuello, P. and González, A., Control and data dependence speculation in multithreaded processors. in *Proc. Workshop on Multithreaded Execution, Architecture and Compilation*, (1998), 98-102.

[45] Marcuello, P. and González, A. A Quantitative Assessment of Thread-Level Speculation Techniques *Proc.14th International Symposium on Parallel and Distributed Processing*, IEEE Computer Society, 2000, 595.

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

[47] Méndez-Lojo, M., Nguyen, D., Prountzos, D*., et al.* Structure-driven optimizations for amorphous data-parallel programs *Proc. 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ACM, Bangalore, India, 2010, 3-14.

[48] Model, S. Programming in the Partitioned Global Address Space Model. *Tutorial at Supercomputing*.

[49] Monteiro, P. and Monteiro, M. A Pattern Language for Parallelizing Irregular Algorithms *2nd Annual Conference on Parallel Programming Patterns (ParaPLoP)*, Carefree, Arizona, 2010.

[50] Navabi, A., Zhang, X. and Jagannathan, S. Quasi-static scheduling for safe futures *Proc. 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, ACM, Salt Lake City, UT, USA, 2008, 23-32.

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

[52] Pancake, C. and Bergmark, D. Do parallel languages respond to the needs of scientific programmers? *IEEE Computer*, *23* (12). 13-23.

[53] Pingali, K., Kulkarni, M., Nguyen, D*., et al.* Amorphous Data-parallelism in Irregular Algorithms, The University of Texas at Austin, Department of Computer Sciences, Austin, TX, USA, 2009.

[54] Pingali, K., Nguyen, D., Kulkarni, M*., et al.* The tao of parallelism in algorithms. *SIGPLAN Not.*, *46* (6). 12-25.

[55] Rauchwerger, L. and Padua, D., The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. in *Proc. Programming Language*

Design and Implementation, (1995), ACM New York, NY, USA, 218-232.

[56] Reinders, J. Intel Threaded Building Blocks, O'Reilly Press, 2007.

[57] Rünger, G. and Schwind, M., Parallelization Strategies for Mixed Regular-Irregular Applications on Multicore-Systems. in *Proc. 8th International Symposium on Advanced Parallel Processing Technologies*, (Rapperswil, Switzerland, 2009), Springer-Verlag, 375-388.

[58] Schmidt, D., Stal, M., Rohnert, H*., et al. Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley, 2000.

[59] Shewchuk, J. Delaunay refinement algorithms for triangular mesh generation. *Comp. Geom.-Theor. Appl.*, *22* (1-3). 21-74.

[60] Sinnen, O. *Task scheduling for parallel systems*. Wiley-Blackwell, 2007.

[61] Steffan, J., Colohan, C., Zhai, A*., et al.* A scalable approach to thread-level speculation. *ACM Comp. Ar.*, *28* (2). 1-12.

[62] Sugerman, J., Fatahalian, K., Boulos, S*., et al.* GRAMPS: A programming model for graphics pipelines. *ACM T. Graphic*, *28* (1). 4.

[63] Sui, X., Nguyen, D., Burtscher, M*., et al.* Parallel graph partitioning on multicore architectures. *Languages and Compilers for Parallel Computing*. 246-260.

[64] Tomasulo, R. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Res. Dev.*, *11* (1). 25-33.

[65] Verma, C. Multithreaded Delaunay Triangulation. *College of William and Mary, Williamsburg, VA*.

[66] von Praun, C., Ceze, L. and Cascaval, C. Implicit parallelism with ordered transactions *Proc. ACM SIGPLAN symposium on Principles and practice of parallel programming*, ACM, San Jose, California, USA, 2007, 79-89.

[67] Wirth, N. *Algorithms + Data Structures=programs*. Prentice Hall Englewood Cliffs, New Jersey, 1985.