The Pennsylvania State University
The Graduate School
College of Engineering

CONCURRENT ASSEMBLIES: A MODEL FOR CONCURRENT PROGRAM EXECUTION

A Dissertation in
Computer Science and Engineering
by
Roberto Lublinerman

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2012

The dissertation of Roberto Lublinerman was reviewed and approved* by the following:

Swarat Chaudhuri
Adjunct Professor of Computer Science and Engineering
Dissertation Adviser, Chair of Committee

John Hannan
Associate Professor of Computer Science and Engineering

Mahmut Kandemir
Professor of Computer Science and Engineering
Director of Graduate Affairs

Patrick Reed
Associate Professor of Civil Engineering

Raj Acharya
Professor of Computer Science and Engineering
Department Head

*Signatures are on file in the Graduate School.

Abstract

We present *Concurrent Assemblies*, an abstract model for modeling shared memory parallel programs. In particular *Concurrent Assemblies* is targeted to irregular parallel applications such as those manipulating mutable pointer-based data structures. Typically in these applications, there is no a priori bound on the available parallelism.

The *Concurrent Assemblies* abstract execution model addresses dynamism and locality through the unifying abstraction of an *concurrent assembly*: a local region in a shared data structure equipped with a short-lived, speculative thread of control. The thread of control in an assembly can only access objects within the assembly. While objects can migrate from assembly to assembly, such migration is *local*—i.e., objects only move from one assembly to a neighboring one—and does not lead to aliasing. Programming primitives include a *merge* operation, by which an assembly merges with an adjacent assembly, and a *split* operation, which splits an assembly into smaller ones. Our abstractions are race-free and inherently data-centric.

In addition to the formal definition of the *Concurrent Assemblies* abstract model, we show the usefulness of such an abstract model by presenting two different programming languages that can be modeled as instances of the *Concurrent Assemblies* abstract model. The first one is JCHORUS, a high-level parallel programming language built on top of the sequential subset of Java, suitable for irregular, heap-manipulating applications like mesh refinement and epidemic simulations. One goal of JCHORUS is to express the *dynamic* and instance-dependent patterns of memory access that are common in typical irregular applications. Its other focus is *locality of effects*: the property that in many of the same applications, typical imperative commands only affect small, local regions in the shared heap.

The second is HABANERO-CHORUS, a parallel subset of the HABANERO-JAVA

programming language. Here concurrency is expressed as asynchronous software transactions.

We present implementation of JCHORUS and HABANERO-CHORUS as well as report on the performance numbers.

Table of Contents

List of Figures **viii**

List of Tables **x**

Acknowledgments **xi**

Chapter 1
Introduction **1**
1.1 Contributions and Organization 6

Chapter 2
The *Concurrent Assemblies* Execution Model **8**
2.1 Key Features of *Concurrent Assemblies* 8
2.1.1 Heaps 8
2.1.2 *Concurrent Assemblies* 9
2.2 Formal Model 11
2.2.1 Syntax 12
2.2.2 Semantics 13
2.2.3 Operational Semantics 14

Chapter 3
The JChorus Programming Language **17**
3.1 Main Programming Abstractions in JChorus 18
3.1.1 Example: Delaunay mesh refinement 23
3.1.2 JChorus features not available in the Concurrent Assemblies abstract model 25
3.2 Formal Semantics of JChorus 27
3.2.1 Syntax 27

3.2.2 Semantics . . . 28
3.2.3 Race- and deadlock-freedom . . . 29
3.3 Case studies . . . 30
3.3.1 Delaunay Mesh Refinement . . . 30
3.3.2 Minimum-spanning tree . . . 32
3.3.3 Barnes-Hut n-body Simulation . . . 32
3.3.4 Agent-based models in epidemiology . . . 35
3.3.5 Focused community discovery . . . 37
3.4 Implementing JChorus . . . 38
3.4.1 The JCHORUS compiler and runtime system . . . 39
3.4.1.1 Divisions . . . 39
3.4.1.2 Executor threads and token-passing . . . 40
3.4.1.3 Assembly migration and load-balancing . . . 44
3.4.1.4 Use of Union-Find . . . 44
3.5 Experiments . . . 45
3.5.1 Delaunay Mesh Refinement . . . 46
3.5.2 Boruvka's algorithm for minimum spanning trees . . . 47

Chapter 4
The HABANERO-CHORUS Programming Language 51
4.1 Main Abstractions in HABANERO-CHORUS . . . 51
4.1.1 Extended example: Delaunay mesh refinement . . . 54
4.2 Formal Semantics . . . 56
4.2.1 Syntax . . . 56
4.2.2 Semantics . . . 57
4.2.3 Properties of HABANERO-CHORUS . . . 59
4.3 HABANERO-CHORUS Implementation Details . . . 60
4.4 Experimental Evaluation . . . 65
4.4.1 Benchmarks . . . 65
4.4.1.1 Boruvka minimum spanning tree . . . 65
4.4.1.2 Delaunay mesh refinement . . . 67
4.4.1.3 Hash table and Red-black tree . . . 67
4.4.2 Methodology . . . 67
4.4.3 Experimental Results . . . 68
4.4.3.1 Self relative speedups . . . 69
4.4.3.2 Execution times . . . 69
4.4.3.3 Aborted speculative work . . . 73
4.4.3.4 Overhead . . . 76
4.4.3.5 Programmability . . . 76

Chapter 5
Related Work **80**

Chapter 6
Conclusions **84**

Bibliography **86**

List of Figures

1.1 Ruppert's Delaunay mesh refinement algorithm 2
1.2 A retriangulation 3
1.3 *Concurrent Assemblies* primitives 6

2.1 A heap 9
2.2 Merging and splitting 11
2.3 Core Syntax 12
2.4 Structural operational semantics of *Concurrent Assemblies*. 15

3.1 Delaunay mesh refinement in JChorus 19
3.2 Control flow in assemblies 20
3.3 Delaunay mesh refinement: sequential algorithm 23
3.4 Snapshot of a small part of a Delaunay mesh 24
3.5 The essence of Delaunay Mesh Refinement in JChorus 25
3.6 Syntax of JChorus 27
3.7 Sematics of JChorus 29
3.8 A cavity in construction, its center triangle (in dark), the cavity triangles (light gray) and the border triangles (dotted lines). The white triangles are either part of the frontier or the destination of one of the outgoing edges. 31
3.9 Minimum spanning tree 33
3.10 n-body simulation 35
3.11 n-body simulation in JChorus 36
3.12 Focused Communities 37
3.13 Focused Communities: the core and the fringe. 38
3.14 The JChorus compiler and runtime system 39
3.15 Divisions in a heap 41
3.16 Executor thread loop for division D 42

3.17 Delaunay Mesh Refinement benchmark results. Top: Speedup results over sequential baseline. Center: Speedup over 1-thread version. Bottom: Percentage of tasks that end up in conflict (for JChorus conflicts occur when incomplete cavities are discarded. For DSTM2 and Galois conflicts occur when tasks are aborted and rolled back.) 49
3.18 Boruvka's algorithm benchmark. Top: Speedup relative to sequential baseline. Bottom: Speedup over 1-threaded version 50

4.1 A conflict 52
4.2 Delaunay mesh refinement (sequential) 53
4.3 Delaunay Refinement in the Habanero-Chorus programming model 54
4.4 Syntax of Habanero-Chorus 57
4.5 Sematics of Core Habanero-Chorus 58
4.6 Principal abstractions in the Habanero-Chorus implementation . 61
4.7 Algorithm for acquiring an object 62
4.8 Union find data structure to track object ownerships 64
4.9 Boruvka Minimum Spanning Tree pseudocode 66
4.10 HashTable and RBTree pseudocode 66
4.11 Delaunay mesh refinement on a 16-core Xeon (Note: For reasons of scale DSTM2 is left out of the plot. DSTM2 -1 thread: 962s, 16 threads: 177s) 69
4.12 Delaunay mesh refinement on a 64-way UltraSPARC T2 (Note: For reasons of scale DSTM2 is left out of the plot . DSTM2 -1 thread: 795s, 64 threads: 81s) 70
4.13 Boruvka minimum spanning tree on a 16-core Xeon (Note: For reasons of scale HJ (ISO) and DSTM2 are left out of the plot HJ (ISO)- 1 thread: 5.5s, 16 threads: 10.3s, DSTM2- 1 thread: 4.8 , 16-threads:2.7.s) 71
4.14 Boruvka minimum spanning tree on a 64-way UltraSPARC T2 (left: logarithmic scale; right: inset of the bottom 6 seconds) 72
4.15 Delaunay self relative speedup 73
4.16 Boruvka self relative speedup 74
4.17 Impact of speculative execution 75
4.18 Actual code for Delaunay Refinement in the Habanero-Chorus programming model 77
4.19 Delaunay Refinement in the Galois programming model 78

List of Tables

4.1 Benchmark results a 16-core Xeon (time in seconds). 76
4.2 Benchmark results ona 64-way UltraSPARC (time in seconds). . . . 77

Acknowledgments

I owe my deepest gratitude to my academic advisor, Swarat Chaudhuri for his support throughout this long and winding road, his enthusiasm that helped surpass all obstacles, his dedication, and the long and fruitful discussions we had during this process. I would not be here if it were not for his encouragement and advise.

I also wish to thank the committee members, John Hannan, Mahmut Kandemir and Patrick Reed for their contributions.

I would like to thank Vivek Sarkar and the Habanero group at Rice University for providing support and a very welcoming and stimulating environment.

My longtime friend and collaborator Stavros Tripakis for his support, continuous collaborations and countless hours of discussions.

I wish to thank my wife and daughters. Without their patience and understanding devoting the needed time to this endeavor would not have been possible.

Lastly, I would like to thank my parents, whose insistence on pursuing knowledge regardless of economic benefit has shaped my life and without which I would have even considered enrolling in a graduate program.

Dedication

To Araceli and Chloe who light up each and every one of my days.

To Alicia for her unwavering love and support.

To my parents for instilling in me the love for knowledge.

Chapter 1

Introduction

In his widely popular 1987 article [1], Fred Brooks states that there is no silver bullet that will bring even one order of magnitude increase in productivity nor simplicity in building software systems in one decade. Building software, he states, involves essential tasks, those inherently related to the creation of the abstract software entity and accidental tasks, those related to its representation in programming languages. The widespread adoption of multicore machines and the need to build systems that perform efficiently on these architectures has increased the complexity of accidental tasks.

This new multicore world has brought to the foreground the need for new programming models to build systems that perform efficiently on these platforms. The most popular programming models for concurrency—message passing and shared-memory with locks - provide abstractions that are too low-level, complex or error prone, and do not scale well with software complexity. Higher level abstractions, such as software transactions suffer in performance and scalability in high contention scenarios. For some applications there has been a lot of progress; algorithms that perform computations on regular data structures, such as numerical computations on matrices, are normally expressed in traditional sequential programming languages and parallelizing compilers take care of the rest by static parallelization[2, 3, 4, 5, 6]. However parallelizing the most general class of algorithms, i.e. algorithms performing computations on data structures such as mutable trees or graphs, remains for the most part an art. In this class of algorithms, which we will call *irregular*, there is no apriori knowledge that allows to

```
while ∃t. t is a bad triangle in mesh
    new_point := center of t's circumcircle;
    cavity := set of triangles whose circumcircles include new_point;
    retriangulate cavity
```

Figure 1.1. Ruppert's Delaunay mesh refinement algorithm

statically partition the data to perform the computations in parallel.

Although much attention has been given to parallelizing regular algorithms, parallelization of irregular applications remains an unsolved problem. In this work we will provide an alternative abstraction that is suited for parallelizing implementations of irregular applications.

Consider the Delaunay mesh refinement problem (DMR for short), a classic example of an irregular parallel application. Suppose we are given a set of points P in two dimensional space and a triangle mesh M that covers all points P. Additionally the mesh M satisfies the Delaunay property, i.e. no point in P falls in a circumcirle of a triangle in M. Our goal is to produce a new triangle mesh on a set of points P' that includes P that also satisfies the Delaunay property and additionally no triangle has an angle less than some quality value q (tipically q is less than 25°). We will call triangles that do not satisfy the quality constrating *bad quality* triangles or simply *bad* triangles.

We will consider the following algorithm proposed by J. Ruppert [7] whose pseudocode is show in figure 1.1. The key idea to rid the mesh of bad quality triangles, is that one can perform local operations around each bad quality triangle. Figure 1.2 illustrates the transformation that is applied to a neighborhood of a bad quality triangle: (a) a bad quality triangle is chosen, (b) the center of its circumcirle will added to the mesh, (c) all the triangles that invalidate the Delaunay property will be called a cavity, (d) the triangles in the cavity will be disposed, and finally (e) new triangles will be created to include the new mesh point while satisfying the Delaunay Property. Note that at the end of a retriangulation new bad quality triangles might be created. The process of applying this transformation iteratively to bad quality triangles is guaranteed to terminate.

Clearly the above algorithm can be embarrassingly parallel. One could choose any set of non-overlapping neighborhoods and apply the transformation to those

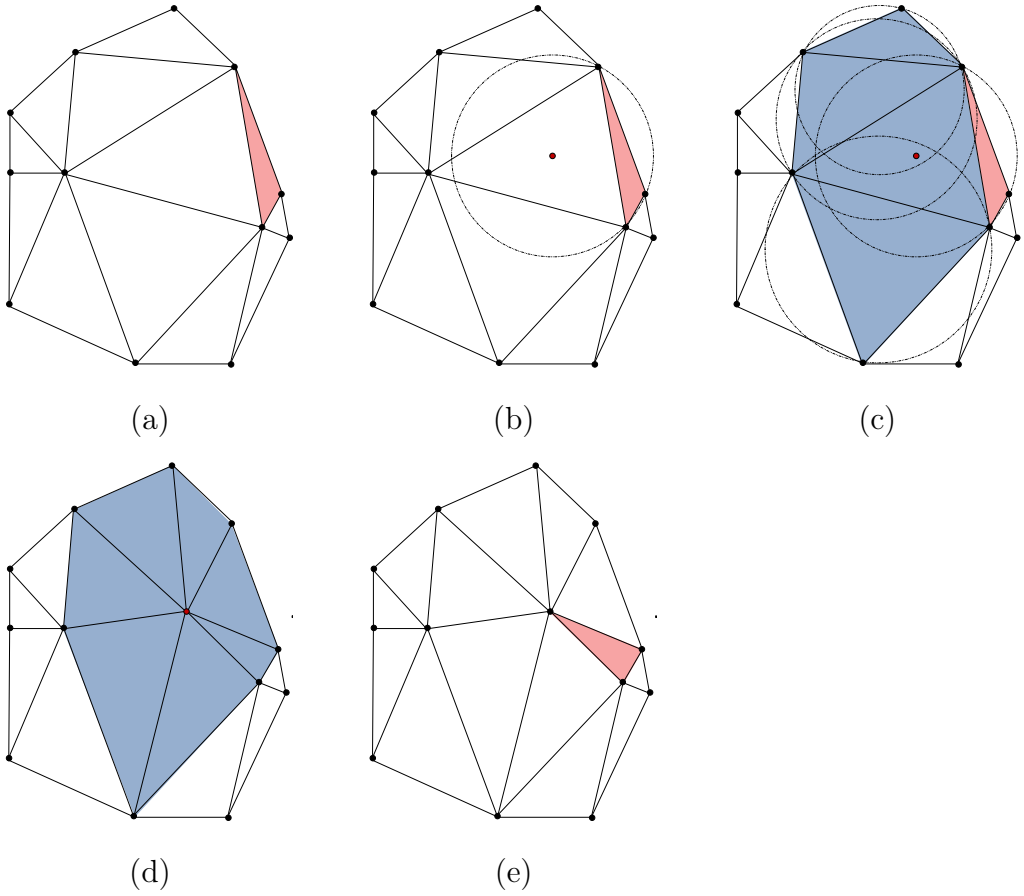


Figure 1.2. A retriangulation

neighborhoods in parallel. However there is no guarantee that cavities will not encompass a large part of the mesh, hence the amount of parallelism is highly data dependent. Moreover the number of actual cavities that can be retriangulated in parallel is completely data dependent.

Concurrent Assemblies is a parallel exucution model targeting problems like the above. *Concurrent Assemblies* offers a view of concurrency that is neither as global as Java multithreading, nor as static as traditional data partitioning, nor based on low-level message-passing like the Actor model. The key abstraction here is a *concurrent assembly*: a dynamically defined local region in the heap equipped with a short-lived thread of control.

At any point in the execution, the assemblies in the system form a *disjoint partitioning* of the heap. Typically, they are also fine-grained—in particular, an

assembly is allowed to consist of just a single object. Thus, like the Actor model, Concurrent Assemblies permits massive, object-level parallelism. Of course, assemblies are just abstractions—in any real implementation, large numbers of them would be mapped to a single hardware thread.

An assembly can perform three kinds of actions:

- It can read and write objects within itself. Notably, it cannot access objects within any other assembly, which means objects within assemblies are *isolated*.

- It can *merge* with an *adjacent* assembly, "becoming" a bigger assembly. The thread of control in the resulting assembly is a combination contributed by the two assemblies involved in the *merge*. (see Figure 1.3 (a))

- It can *split* into a collection of smaller (disjoint) assemblies, each possessing a new thread of control. (see Figure 1.3 (b))

All concurrency in our model is captured with these primitives. The number of assemblies is a proxy for the granularity of concurrency that the application permits—the greater this number, the greater the exploitable parallelism. Assemblies are of course not required to be bounded; in the worst case, they encompass the whole heap. Merges allow declarative and local *coarsening* of the granularity of parallelism in the heap, while splits let parallelism be *locally refined*. One aspect of our model is that there is no global ordering between merges and splits—e.g., merges between distinct pairs of assemblies can always happen in parallel.

Consider Delaunay mesh refinement once again. In our approach, each triangle in the initial mesh is an assembly. If a triangle discovers that it is bad, it forms a cavity (a bigger assembly) via repeated merges with its neighbors. The cavity retriangulates itself via a private update, then splits into the new triangles (each a smaller assembly). The expressed parallelism is at the finest granularity permitted by the problem instance: all triangles and cavities in the heap work in parallel, and atomicity of retriangulation is guaranteed because the data in an assembly is isolated. Thus, it captures the pattern "Own a local region, update the region, release the region," by rephrasing it as:

> "Dynamically form an assembly by repeated merges, update the assembly, split the assembly."

Note that in this scenario the merged assembly does not contribute to the resulting assembly thread of control, resulting in a speculative execution, i.e. a cavity might be in the process of being formed by successive merge operations with its neighbors only to be merged into an adjacent cavity, its thread of control discarded.

While in the previous example *merge* operations occur before the merged assembly modifies its heap, in general the *merge* operations allow for rollbacks to be performed before the merge actually occurs.

Note that *Concurrent Assemblies* can be embedded in a high-level language in many ways. In this dissertation we present two such approaches. The first, JChorus [8], is an approach where the high-level language concepts are almost directly mapped into *Concurrent Assemblies*, allowing the programmer to express programs using a subset of the *Concurrent Assemblies* semantics. Secondly we embed *Concurrent Assemblies* into an existing parallel programming language with a transactional construct, Habanero-Chorus [9]. Here assemblies are typified with isolated asynchronous tasks. Note that *Concurrent Assemblies* can be seen as an implementation mechanism for software transactions.

Concurrent Assemblies is an abstract execution model in which higher-level programming languages can be mapped. This execution model offers a level of abstraction that is higher that shared memory with fine grained locking but lower that software transactional memory.

In this work we develop two such mappings.

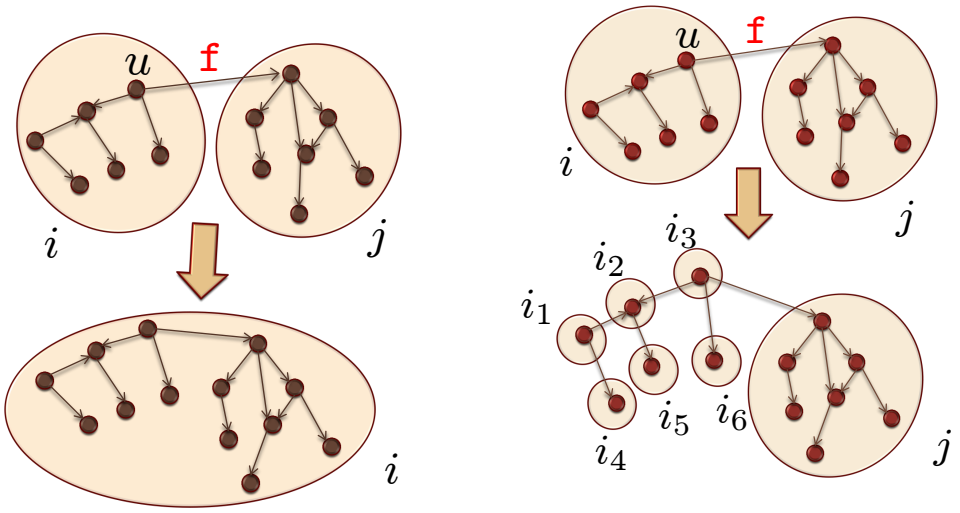


Figure 1.3. *Concurrent Assemblies* primitives

1.1 Contributions and Organization

Our concrete contributions are the following:

- We introduce and formalize the abstract *Concurrent Assemblies* model, a model suitable to express highly parallel algorithms that operate on irregular data structures.

- We present JCHORUS, a programming language that embeds our model of concurrency into the sequential subset of Java; and HABANERO-CHORUS, a programming language whose main abstraction is that of an asynchronous isolated task.

- We demonstrate the utility of JCHORUS and HABANERO-CHORUS in programming real-life applications via several case studies. In addition to mesh refinement, we consider the problems of Barnes-Hut n-body simulation [10], "Focused community" discovery in a social network [11, 12], an epidemiological simulation problem [13], and an algorithm for computing minimum spanning trees [14].

- We present a prototype compiler and runtime system for JCHORUS that uses

a many-to-one mapping of assemblies to low-level threads. The implementation exploits locality of heap operations, uses Tarjan's Union-Find data structure to maintain assemblies and a token-ring-based strategy to ensure deadlock-freedom, and performs an elementary form of load-balancing.

- We present a prototype compiler and runtime system for HABANERO-CHORUS, built on the existing infrastructure of Habanero Java, leveraging many of the aspects in the JCHORUS runtime.

- We show that these programming languages satisfy certain properties such as data race freedom and deadlock freedom by a principled mapping into the *Concurrent Assemblies* abstract model.

This dissertation is organized as follows. In Chapter 2 we present and formalize the *Concurrent Assemblies* model. In Chapter 3 we present JCHORUS, its formalization, its implementation, case studies and performance evaluation. In Chapter 4 we present HABANERO-CHORUS, its formalization, its implementation and performance evaluation. In Chapter 5 we present the related work. Lastly, in Chapter 6 we present our conclusions and future work.

Chapter 2

The *Concurrent Assemblies* Execution Model

Now we present the *Concurrent Assemblies* execution model.

2.1 Key Features of *Concurrent Assemblies*

Now we outline the basic notions of *Concurrent Assemblies*.

2.1.1 Heaps

The central structure in the *Concurrent Assemblies* execution model is the shared-memory *heap*, which maintains the state of all shared data accessed by a parallel program. We abstractly view a heap as a directed graph whose nodes are objects and edges are pointers. Pointers here are labeled with *field names*. A *region* in a heap G is a graph consisting of a subset of the nodes of G, and all edges of G that connect nodes in this subset.

For example, in Figure 2.1, u_1, u_2, u_3, and v are objects, there is a pointer from u_2 to v labeled by the field name `f`, and each shaded circle is a region. Or consider Delaunay mesh refinement. The mesh here can be modeled as a heap whose objects are triangles and whose pointers connect triangles that are neighbors in the mesh. Each cavity is a region in the heap.

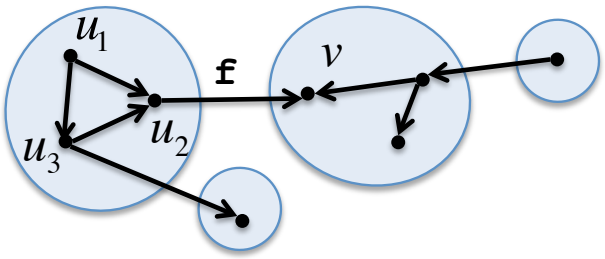


Figure 2.1. A heap

2.1.2 *Concurrent Assemblies*

A *concurrent assembly* in G is a region of G equipped with a set of local variables that map to objects in G and a sequential thread of control. The typical execution scenario has numerous assemblies executing concurrently. It is required that at each point in an execution, these assemblies form a *disjoint partitioning* of the heap—in other words, every object in the heap belongs to (the region of) an assembly, and no object belongs to two distinct assemblies.

While an assembly can update the heap, it embodies *isolation*: it has exclusive ownership of its region and can neither read nor write objects that fall outside it. This means that imperative effects are local: a heap modification by one assembly does not affect the data read by another. An assembly is allowed to merge with adjacent assemblies and can also split into a set of smaller assemblies. In typical scenarios, it is short-lived and exists to achieve a specific, local task—e.g., the retriangulation of a single cavity in Delaunay mesh refinement.

Each assembly here is equipped with a thread of control in the form of a program S and some local variables that are referenced by the program S. This set of *local variables* $\mathtt{u}, \mathtt{v}, \ldots$ are used to refer to objects (within or outside their regions. Objects can also be referenced using field expressions in the usual way: if the variable `u` refers to the object u and u has an `f`-labeled edge to v, then the name `u.f` refers to v. In order to evaluate or assign an expression of the form `u.f`, `u` must refer to an object in the assembly region.

The active behavior of an assembly is defined by its *program* S. Concurrent assemblies have complete control of when interactions between them can occur. An assembly can only be merged if it is executing a `merge` instruction. Because of this fact, freedom of data races is guaranteed by the isolated nature of local heap up-

dates. However, freedom of deadlocks can not be guaranteed for general programs in this execution model. In the sequel we will show that by doing a principled mapping we can guarantee deadlock freedom for the higher level programming languages that are mapped into Concurrent Assemblies.

`Merge` and $\overline{\texttt{merge}}$ are synchronization operations—in fact, they are our only synchronization operations. Figures 2.2-(a) and 2.2-(c) show the states of a parallel program before and after the assembly i_1 merges with i_2. Note that `merge`s *locally coarsen* the granularity of parallelism in the heap. When assembly i_1 merges with assembly i_2, these two assemblies are replaced by assembly i_3 whose region of the heap consists of the union for the regions of i_1 and i_2. The thread of control of this new assembly i_3 is determined by the original assemblies i_1 and i_2 according to a precise semantics defined in the sequel. The merge operation provides an option for rollback of the actions of either or both assemblies to some predefined previous state.

As for updates, they permit an assembly to imperatively modify its region—any expression whose evaluation requires accesses outside H returns an error value `error`. An update can also *split* an assembly into smaller ones—e.g., into assemblies containing one object each (Figures 2.2-(a) and 2.2-(b) show before-and-after scenarios for this operation). Observe that the split *locally refines* the parallelism in the system.

Importantly, merges and splits are not globally ordered: a merge between assemblies i_1 and i_2 can proceed in parallel with a merge between j_1 and j_2 (where j_1 and j_2 are distinct from i_1 and i_2), or with a split of j_1. Also, a modification within i_1 can run parallel to every action outside of i_1.

Finally, note that our object model does not allow for aliasing. At any time, an object belongs to only one assembly (we do make an exception for *read-only data* in JCHORUS—see Section 3). While an assembly can have a *reference* to an object outside its region, it cannot use this reference for reading or writing.

At any point in the execution, the assemblies in the system form a *disjoint partitioning* of the heap. They can be as fine-grained as needed—in particular, an assembly is allowed to consist of just a single object. Thus, like the Actor model, the *Concurrent Assemblies* execution model permits massive, object-level parallelism. Of course, assemblies are high level abstraction—in any real imple-

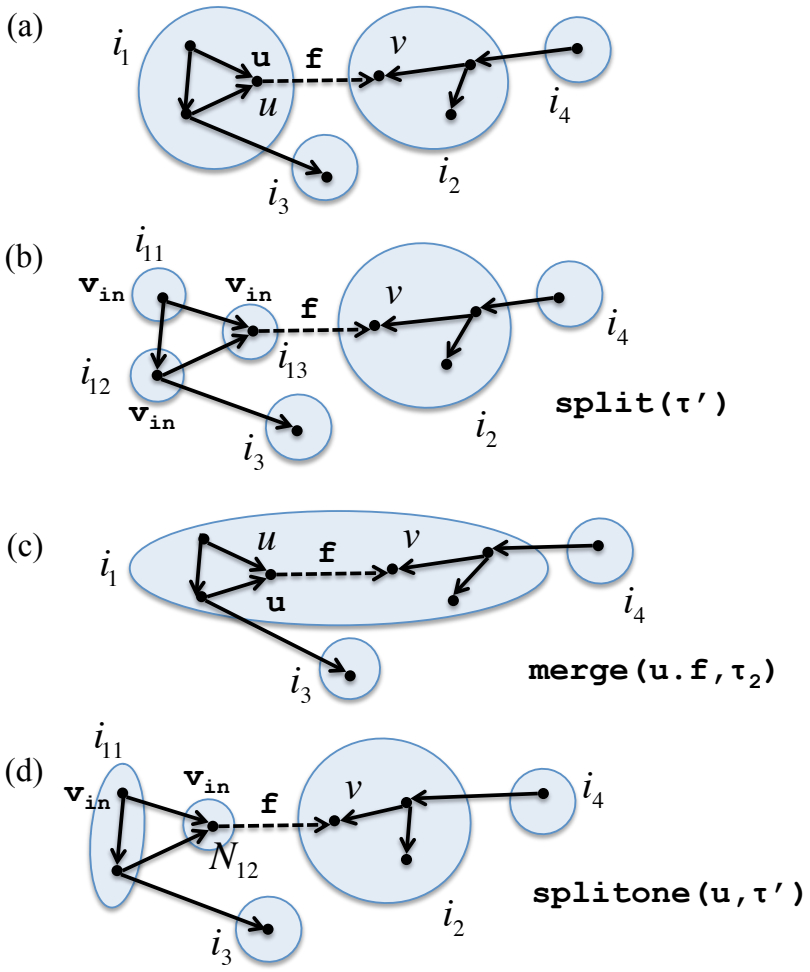


Figure 2.2. Merging and splitting

mentation, large numbers of them would be mapped to a single hardware thread.

All concurrency in our model is captured with these primitives. The number of assemblies is a proxy for the granularity of concurrency that the application permits—the greater this number, the greater the exploitable parallelism. Assemblies are of course not required to be bounded; in the worst case, they encompass the whole heap.

2.2 Formal Model

In this section we define the formal syntax and semantics for the *Concurrent Assemblies* execution model.

2.2.1 Syntax

$$
\begin{array}{rcl}
\mathit{Stmt} & ::= & \mathtt{v} := \mathtt{u} \mid \mathtt{v} := \mathtt{u.f} \mid \mathtt{v.f} := \mathtt{u} \mid \mathtt{v} := \mathtt{new} \mid \mathtt{v.f} := \epsilon \mid \\
& & \mathit{Stmt};\ \mathit{Stmt} \mid \mathit{Stmt} + \mathit{Stmt} \mid \mathit{BExp}?\mathit{Stmt} \mid \\
& & \mathtt{split}(\tau, \mathit{Stmt}) \mid \\
& & \mathtt{merge}(\mathit{var}, \tau, \mathit{Stmt}) \mid \mathtt{merge}_r(\mathit{var}, \tau, \mathit{Stmt}) \mid \\
& & \overline{\mathtt{merge}}(\mathit{Stmt}) \mid \overline{\mathtt{merge}}_r(\mathit{Stmt}) \mid \\
& & \mathtt{commit}
\end{array}
$$

where $\mathtt{v}, \mathtt{u} \in \mathit{Var}$, $\mathtt{f} \in F$, $\tau \in \mathcal{T}$ a finite set of types and *BExp* is a boolean expression from some set of suitable operators.

Figure 2.3. Core Syntax

Programs in the *Concurrent Assemblies* execution model are written according to the syntax presented in Figure 2.3. The first seven types of statements involve local graph updates and program composition operators:

`v` = `u` copies the reference in variable `u` to variable `v` (the object referred by `u` does not need to belong to the assembly region).

`v` = `u.f` copies the reference in field `u.f` to variable `v` (the object referred by `u.f` does not need to belong to the assembly region, however the object referred by `u` does).

`v.f` = `u` creates or modifies the `f`-edge of the node referred by `v` to point to the node referred by `u`.

`v.f` = `new` creates a new node and creates or modifies the `f`-edge of the node referred by `v`.

`v.f` = ϵ eliminates the `f`-edge of the node referred by `v` if it existed.

Stmt_1 ; Stmt_2 is the standard sequential composition.

Stmt_1 + Stmt_2 is the *committed* choice operator. For a nondeterministic choice of Stmt_1 (resp. *Stmt*2) to be made, it is required that Stmt_1 (resp. Stmt_2) does not get immediately stuck.

BExp?*Stmt* is the conditional operator, where *Stmt* is only executed if *BExp* evaluates to true in the present state.

The next three types of statement increase or reduce the number of concurrent assemblies executing:

> `split(`$Stmt$`)` ends the current assembly and creates one assembly for each heap node within the assembly region with $Stmt$ as its program..
>
> `merge(v,` τ`,` $Stmt_1$`)` and $\overline{\texttt{merge}}$`(`$Stmt_2$`)` are the merge request operator and merge accept operator respectively. The assembly executing a merge request statement will merge with the target assembly containing the node pointed by reference `v` in its region; provided that the target assembly is of type τ and is executing a merge accept operator. The assembly resulting from this operation will have $Stmt_1$; $Stmt_2$ as its program. The `merge`$_r$`(v,` τ`,` $Stmt_1$`)` and $\overline{\texttt{merge}}_r$`(`$Stmt_2$`)` versions of the previous commands additionally prescribe a *rollback* of the assembly actions. The new assembly will have $Stmt1$; $Stmt_2$ as its program. It is assumed that $Var(Stmt1)$ and $Var(Stmt2)$ are disjoint with no loss of generality.

Last the `commit` statement allows an assembly to "commit" its local changes so far so that they can not be rolled back.

2.2.2 Semantics

First we define heaps formally. Letting Loc be a set of *abstract locations* and F a set of *field names*, we have:

Definition 1 (Heap)**.** A *heap* is an edge-labeled directed graph $G = (O \subseteq Loc, E \subseteq O \times F \times O)$, where O is the node set and E is the F-labeled edge set, such that for each $u \in O, \texttt{f} \in F$, there is at most one edge of the form $(u, \texttt{f}, v)$ in E. Nodes and edges of G are respectively known as *objects* and *pointers*.

A heap H is a *region* in another heap G if it is a subgraph induced by a subset of the nodes of G.

We have the following definitions for assemblies and the global program state:

Definition 2 (Concurrent Assembly)**.** A *concurrent assembly* is a tuple $N = \langle \tau, S, H, \mu, (H', \mu') \rangle$, where $\tau \in \mathcal{T}$ is a type name, S is statement, H is a region

in G, $\mu : Var(S) \to (G \cup \texttt{error})$, H' is some initial state of a region and μ' is an initial mapping (we normally denote the pair (H', μ') by the letter ρ).

Two concurrent assemblies $N_1 = \langle \tau_1, S_1, H_1, \mu_1, \rho_1 \rangle$ and $N_2 = \langle \tau_2, S_2, H_2, \mu_2, \rho_2 \rangle$ are *disjoint* if H_1 and H_2 do not have nodes in common.

Intuitively, in the definition of the assembly N, H is the region on which i operates, μ provides mappings from the local variables of S to objects in G (or the "uninitialized" value `error`), and ρ provides a rollback transformation to some previous state of the local heap.

The global program state comprises the states of a set of assemblies whose regions partition the heap:

Definition 3 (Global program state)**.** A state σ of P is a tuple $\sigma = \langle G, \Gamma \rangle$, where G is a heap and Γ is a set of concurrent assemblies such that each object u in G belongs to the region of some $N \in \Gamma$, and for each $N_1, N_2 \in \Gamma$ with $N_1 \neq N_2$, we have: N_1 and N_2 are disjoint.

2.2.3 Operational Semantics

Now we present an interleaving operational semantics of Concurrent Assemblies. The semantics defines a transition relation $\longrightarrow$ between states $\langle G, \Gamma \rangle$, where G is any heap. The graph G is not required to be the complete heap of the parallel program—it can be any region in the complete heap. A transition $\langle G, \Gamma \rangle \longrightarrow \langle G', \Gamma' \rangle$ says that where G' is obtained by repeatedly rewriting G in *isolation*—i.e., without requiring information about objects outside of G. Thus, the semantics is in a sense *modular over space*.

Values in this semantics can be objects or `error`. We use the following additional notation.

- An *auxiliary term* is either a value or a term $\langle N, e \rangle$, where N is an assembly and e is a local variable `v` or a field access `v.f`. We use a transition relation $\dashrightarrow$ over auxiliary terms.

- Let $G = (O, E)$ be a heap with a region H; consider another region H' with the same node set as H. Then $G[H \rightsquigarrow H']$ denotes the graph obtained by removing all edges in H from G, then adding to it all edges in H'.

$$\text{(Assembly-step)}\quad \frac{N = \langle \tau, S, H, \mu, \rho\rangle \in \Gamma \qquad \langle H, \{N\}\rangle \longrightarrow \langle H', \Gamma'\rangle}{\langle G, \Gamma\rangle \longrightarrow \langle G[H \rightsquigarrow H'], \Gamma \setminus \{N\} \cup \Gamma'\rangle}$$

$$\text{(Merge)}\quad \frac{\begin{array}{c} N_1 = \langle \tau_1, \texttt{merge}(\texttt{v}, \tau_2, S_{cont_1});\ S_1, H_1, \mu_1, \rho_1\rangle \in \Gamma \qquad N_2 = \langle \tau_2, \overline{\texttt{merge}}(S_{cont_2});\ S_2, H_2, \mu_2, \rho_2\rangle \in \Gamma \\ \mu(\texttt{v}) = v \qquad v \text{ is in } H_2 \qquad \rho_1 = (H_1', \mu_1') \qquad \rho_2 = (H_1', \mu_1') \qquad dom(\mu_1) \cap dom(\mu_2) = \emptyset \end{array}}{\langle G, \Gamma\rangle \longrightarrow \langle\ G, \Gamma \setminus \{N_1, N_2\} \cup \{\langle \tau_1, S_{cont_1};\ S_{cont_2}, H_1 \sqcup_G H_2, \mu_1 \cup \mu_2, (H_1' \cup H_2', \mu_1' \cup \mu_2')\rangle\}\rangle}$$

$$\text{(Merge-rollback-left)}\quad \frac{\begin{array}{c} N_1 = \langle \tau_1, \texttt{merge}_r(\texttt{v}, \tau_2, S_{cont_1});\ S_1, H_1, \mu_1, \rho_1\rangle \in \Gamma \qquad N_2 = \langle \tau 2, \overline{\texttt{merge}}(S_{cont_2});\ S_2, H_2, \mu_2, \rho_2\rangle \in \Gamma \\ \mu(\texttt{v}) = v \qquad v \text{ is in } H_2 \qquad \rho_1 = (H_1', \mu_1') \qquad \rho_2 = (H_1', \mu_1') \qquad dom(\mu_1) \cap dom(\mu_2) = \emptyset \end{array}}{\langle G, \Gamma\rangle \longrightarrow \langle G[H_1 \rightsquigarrow H_1'], \Gamma \setminus \{N_1, N_2\} \cup \{\langle \tau_1, S_{cont_1};\ S_{cont_2}, H_1' \sqcup_G H_2, \mu_1' \cup \mu_2, (H_1' \cup H_2', \mu_1' \cup \mu_2')\rangle\}\rangle}$$

$$\text{(Merge-rollback-right)}\quad \frac{\begin{array}{c} N_1 = \langle \tau_1, \texttt{merge}(\texttt{v}, \tau_2, S_{cont_1});\ S_1, H_1, \mu_1, \rho_1\rangle \in \Gamma \qquad N_2 = \langle \tau_2, \overline{\texttt{merge}}_r(S_{cont_2});\ S_2, H_2, \mu_2, \rho_2\rangle \in \Gamma \\ \mu(\texttt{v}) = v \qquad v \text{ is in } H_2 \qquad \rho_1 = (H_1', \mu_1') \qquad \rho_2 = (H_1', \mu_1') \qquad dom(\mu_1) \cap dom(\mu_2) = \emptyset \end{array}}{\langle G, \Gamma\rangle \longrightarrow \langle G[H_2 \rightsquigarrow H_2'], \Gamma \setminus \{N_1, N_2\} \cup \{\langle \tau_1, S_{cont_1};\ S_{cont_2}, H_1 \sqcup_G H_2', \mu_1 \cup \mu_2', (H_1' \cup H_2', \mu_1' \cup \mu_2')\rangle\}\rangle}$$

$$\text{(Merge-rollback-both)}\quad \frac{\begin{array}{c} N_1 = \langle \tau_1, \texttt{merge}_r(\texttt{v}, \tau_2, S_{cont_1});\ S_1, H_1, \mu_1, \rho_1\rangle \in \Gamma \qquad N_2 = \langle \tau_2, \overline{\texttt{merge}}_r(S_{cont_2});\ S_2, H_2, \mu_2, \rho_2\rangle \in \Gamma \\ \mu(\texttt{v}) = v \qquad v \text{ is in } H_2 \qquad \rho_1 = (H_1', \mu_1') \qquad \rho_2 = (H_1', \mu_1') \qquad dom(\mu_1) \cap dom(\mu_2) = \emptyset \\ G' = G[H_1 \rightsquigarrow H_1'][H_2 \rightsquigarrow H_2'] \end{array}}{\langle G, \Gamma\rangle \longrightarrow \langle G', \Gamma \setminus \{N_1, N_2\} \cup \{\langle \tau_1, S_{cont_1};\ S_{cont_2}, H_1' \sqcup_G H_2', \mu_1' \cup \mu_2', (H_1' \cup H_2', \mu_1' \cup \mu_2')\rangle\}\rangle}$$

$$\text{(Exp-1)}\quad \frac{N = \langle \tau, S, H, \mu, \rho\rangle \qquad \mu(\texttt{v}) = u}{\langle N, \texttt{v}\rangle \dashrightarrow u} \qquad \text{(Exp-2)}\quad \frac{\begin{array}{c} N = \langle \tau, S, H, \mu, \rho\rangle \\ \langle N, e\rangle \dashrightarrow u \qquad u \xrightarrow{H,\texttt{f}} v \qquad v \text{ is in } H \end{array}}{\langle N, e.\texttt{f}\rangle \dashrightarrow v}$$

$$\text{(Exp-3)}\quad \frac{\langle N, e\rangle \dashrightarrow \texttt{error}}{\langle N, e.\texttt{f}\rangle \dashrightarrow \texttt{error}} \qquad \text{(Exp-4)}\quad \frac{\begin{array}{c} N = \langle \tau, S, H, \mu, \rho\rangle \\ \langle N, e\rangle \dashrightarrow u \qquad u \xrightarrow{H,\texttt{f}} v \qquad v \text{ is not in } H \end{array}}{\langle N, e.\texttt{f}\rangle \dashrightarrow \texttt{error}}$$

$$\text{(Assign-1)}\quad \frac{N = \langle \tau, \texttt{v} := e;\ S, H, \mu, \rho\rangle \qquad \langle N, e\rangle \dashrightarrow u}{\langle H, \{N\}\rangle \longrightarrow \langle H, \{\langle \tau, S, H, \mu[\texttt{v} \mapsto u], \rho\rangle\}\rangle}$$

$$\text{(Assign-2)}\quad \frac{N = \langle \tau, \texttt{v.f} := e;\ S, H, \mu, \rho\rangle \qquad \mu(\texttt{v}) = u \qquad \langle N, e\rangle \dashrightarrow v \qquad H' = H[(u, \texttt{f}, \_) \rightsquigarrow (u, \texttt{f}, v)]}{\langle H, \{N\}\rangle \longrightarrow \langle H', \{\langle \tau, S, H', \mu, \rho\rangle\}\rangle}$$

$$\text{(Choice-left)}\quad \frac{\langle H, \{\langle \tau, S_1;\ S, H, \mu, \rho\rangle\}\rangle \longrightarrow \langle H', \{\langle \tau', S', H', \mu', \rho'\rangle\}\rangle}{\langle H, \{\tau, \langle (S_1 + S_2);\ S, H, \mu, \rho\rangle\}\rangle \longrightarrow \langle H', \{\langle \tau', S', H', \mu', \rho'\rangle\}\rangle}$$

$$\text{(Choice-right)}\quad \frac{\langle H, \{\langle \tau, S_2;\ S, H, \mu, \rho\rangle\}\rangle \longrightarrow \langle H', \{\langle \tau', S', H', \mu', \rho'\rangle\}\rangle}{\langle H, \{\langle \tau, (S_1 + S_2);\ S, H, \mu, \rho\rangle\}\rangle \longrightarrow \langle H', \{\langle \tau', S', H', \mu', \rho'\rangle\}\rangle}$$

$$\text{(Cond)}\quad \frac{\langle H, \{\langle \tau, S_1;\ S_2, H, \mu, \rho\rangle\}\rangle \longrightarrow \langle H', \{\langle \tau', S', H', \mu', \rho'\rangle\}\rangle \qquad \langle\langle H, \{\langle \tau, S_1;\ S_2, H, \mu, \rho\rangle\}\rangle, b\rangle \dashrightarrow \texttt{true}}{\langle H, \{\langle \tau, (b?S_1);\ S_2, H, \mu, \rho\rangle\}\rangle \longrightarrow \langle H', \{\langle \tau', S', H', \mu', \rho'\rangle\}\rangle}$$

$$\text{(Split)}\quad \frac{N = \langle \tau, \texttt{split}(\tau', S');\ S, H, \mu, \rho\rangle \qquad H \text{ has objects } u_1, \ldots, u_n}{\langle H, \{N\}\rangle \longrightarrow \langle H, \{\langle \tau', S', \{u_j\}, \mu_{Init}(S', u_j), (\{u_j\}, \mu_{Init}(S', u_j))\rangle : 1 \le j \le n\}\rangle}$$

$$\text{(Commit)}\quad \frac{N = \langle \tau, \texttt{commit};\ S, H, \mu, \rho\rangle \qquad \langle N, e\rangle \dashrightarrow u}{\langle H, \{N\}\rangle \longrightarrow \langle \tau, H, \{\langle S, H, \mu, (H, \mu)\rangle\}\rangle}$$

Figure 2.4. Structural operational semantics of *Concurrent Assemblies*.

Also, $H[(u, \mathtt{f}, \_) \rightsquigarrow (u, \mathtt{f}, v)]$, for $u, v \in O'$, denotes the graph obtained by removing the outgoing edge from u labeled $\mathtt{f}$ and adding the edge $(u, \mathtt{f}, v)$ to H.

- We write $u \xrightarrow{G,\mathtt{f}} v$ if $(u, \mathtt{f}, v)$ is an edge in the heap G, and $\{u\}$ for the heap with the single node u and no edge.

- For regions $H = (O, E)$ and $H' = (O', E')$ in a heap G, $(H \sqcup_G H')$ denotes the union of H and H'—i.e., the subgraph induced on G by $(O \cup O')$. Also, $(G \setminus H)$ denotes the graph obtained by removing the subgraph H from G.

- Let μ map a set of variables V to a set of objects. We denote by $\mu[\mathtt{v} \mapsto u]$, for $\mathtt{v} \in V$, the function μ' that satisfies $\mu'(\mathtt{v}) = u$, and agrees with μ otherwise.

Our operational semantics for Concurrent Assemblies is presented in Figure 2.4). Evaluation here is parameterized by an initial heap G_{in} and begins with the state $\langle G_{in}, \{N\}\rangle$, where N is the assembly $\langle \iota, G_{in}, \mu_{Init}(\iota), (G_{in}, \mu_{Init}(\iota)\rangle$ for the program ι.

Note that updates to assemblies happen in *isolation* from the rest of the heap. The interleaving of these updates is captured by the rule ASSEMBLY-STEP, which propagates the effect of an (atomic) assembly update on the larger heap.

An *execution* of a program P from an initial state $\sigma_0 = \langle G, \Gamma\rangle$ is a sequence $\pi = \sigma_0\sigma_1 \ldots$ where for each i, $\sigma_i \longrightarrow \sigma_{i+1}$.

There are four rules that define the *merging* of two assemblies depending whether none, one or both assemblies rollback their local updates. The rule MERGE defines how to assemblies merge when none requests a rollback to its local effects, while rules MERGE-ROLLBACK-RIGHT, MERGE-ROLLBACK-LEFT and MERGE-ROLLBACK-BOTH defines how assemblies merge when either or both request rollbacks to their local effects. .

Termination. A *terminating global program state* in our model is one where no rules can be applied.

Chapter 3

The JChorus Programming Language

In this chapter we present the JCHORUS programming language [8], one of two programming languages built on top of *Concurrent Assemblies*. The JCHORUS programming language can be seen as a direct mapping into a restricted version of *Concurrent Assemblies*. The main software abstraction in JCHORUS is, as in *Concurrent Assemblies*, an assembly.

The JCHORUS language embeds a substantial subset of the *Concurrent Assemblies* abstract model into the sequential subset of Java, borrowing most of the programming constructs from Java. Assemblies and heap nodes are defined in a way that is similar to Java classes providing a unit of modularity. As in Java they can contain member variables and member methods. The programming language also borrows most of the statement syntax from Java. The goal is to provide a Java like language where all the concurrency construct found in Java are replaced by constructs found in Concurrent Assemblies.

We will present a semi-formal description of the JCHORUS programming language first, outlining the main concepts and presenting its main differences from sequential Java using examples. JCHORUS semantics will be given as a translation to *Concurrent Assemblies*.

3.1 Main Programming Abstractions in JCHORUS

Object classes. Unlike in *Concurrent Assemblies*, where objects were untyped, objects in JCHORUS are declared using Java-style class declarations. We distinguish between *shared* objects, for which assemblies contend, and objects that are private to assemblies. Accordingly, object classes are explicitly declared as shared using the keyword `shared`. Also, objects here support member method. We will use the $\mathcal{C}$ to denote an object class.

For example, consider Figure 3.1, which shows part of a JCHORUS implementation of Delaunay mesh refinement. Here, the class `TriangleObject` represents triangles in the mesh, which are shared objects. A triangle supports a method `isBad()`, which says whether it is "bad."

Assembly classes. In addition to object classes, JCHORUS allows declarations of *assembly classes*. For example, the code in Figure 3.1 declares two assembly classes `Cavity` and `Triangle`, respectively representing assemblies consisting of a cavity and a single triangle. Each assembly has a set of typed assembly variables, some private procedures, and a block demarcated by the keyword `action` that contains all the guarded updates. We will use τ to denote assembly classes.

Guarded updates in JCHORUS have the syntax *Guard* : { *Stmt* }. In addition to the forms of merges shown earlier, we also permit a syntactic sugar version, `merge(L,`$\mathcal{C}$ `x)` {*Stmt*}, where `L` is a collection (e.g., list or set) of references pointing outside the current assembly. The semantics is that the assembly can merge along any edge whose target v is contained in `L`—the precise value of v used for the merge is bound to the name `x` of shared object class $\mathcal{C}$ within the body of *Stmt*.

As for updates, they can now call methods on objects and procedures private to the assembly, and use all imperative control constructs. An access to an object outside the assembly throws an exception `NonLocalException`. Such exceptions are handled using `try-catch` blocks as in Java—see the code in Figure 3.1 for an example.

Finally, in JCHORUS we permit an assembly of class τ to change its class to τ' via a command `become(`τ'`,`$\mathtt{p}_1, \ldots, \mathtt{p}_k$`)` ($\mathtt{p}_1, \ldots, \mathtt{p}_k$ are parameters passed to the constructor of the newly created assembly). For example, in Figure 3.1, a `Triangle` becomes a `Cavity` this way.

```
 1: assembly Triangle {
 2:  Triangle(TriangleObject t) {
 3:     if (t.isBad())
  // become a Cavity
 4:         become(Cavity, t);
 5:   }
 6: } /* end Triangle */

 7: assembly Cavity {
 8:   action  { // expand cavity
 9:      merge(outgoingedges,
               TriangleObject t) : {
10:          outgoingedges.remove(t);
11:          frontier.add(t);
12:          build(); }
13:    }

14:   Set members; Set border;
         // current frontier
15:   Queue frontier;
         // outgoing edges on which
         // to merge
16:   List outgoingedges;
17:   TriangleObject initial;

18:   Cavity(TriangleObject t) {
      ... initialize data fields....
19:      frontier.enqueue(t);
20:      build(); }
      ...


21:    void build() {
22:        while (frontier.size() != 0) {
23:            TriangleObject curr = frontier.dequeue();
24:            try {
25:               if (isMember(curr)) members.add(curr);
26:               else border.add(curr);
                  // add triangles using BFS
27:               for (TriangleObject n: curr.neighbors())
28:                  if (notSeen(n)) frontier.add(n);
29:            } catch(NonLocalException e)
               {      // triangle not in assembly,
                      // add to merge list
30:                outeredges.add(e.getObject());   }
31:         }
32:         if (outeredges.isEmpty()) {
33:             retriangulate(); split(Triangle);
34:         }
35:    }

36:    void retriangulate() { ... }
37:    boolean isMember(TriangleObject t)
          {... }
38:    boolean notSeen(TriangleObject t)
          {... }
39: } /* end Cavity */

40: shared TriangleObject {
41:    Point p1, p2, p3;
42:    Triangle s1, s2, s3;

43:    Point circumCenter() {...}
44:  }

50: assembly Loader {
51:    Loader(String filename) {
52:       ...
53:        ... new Triangle(p1, p2, p3);
54:       ...
55:       split(Triangle);
56:    }
57:  } /* end Loader */
```

Figure 3.1. Delaunay mesh refinement in JCHORUS

The typical execution scenario of JCHORUS programs, as in *Concurrent Assemblies*, has numerous assemblies executing concurrently. It is required that at each point in an execution, these assemblies form a *disjoint partitioning* of the heap—in other words, every object in the heap belongs to (the region of) an assembly, and

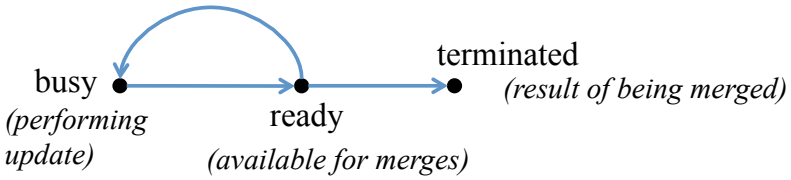


Figure 3.2. Control flow in assemblies

no object belongs to two distinct assemblies.

Also, as in the abstract model, while an assembly can update the heap, it embodies *isolation*: it has exclusive ownership of its region and can neither read nor write objects that fall outside it. This means that imperative effects are local: a heap modification by one assembly does not affect the data read by another. An assembly is allowed to merge with adjacent assemblies and can also split into a set of smaller assemblies.

A key difference between the current version of JCHORUS and *Concurrent Assemblies* is that JCHORUS assemblies are completely speculative with no provision for rollback of partial effects.

Is worth noting that for an important class of irregular applications that includes Delaunay refinement, all object acquisition happens before modification. In these cases, a need for rollback never arises (this class was identified by Pingali et al [15], who referred to such applications as *cautious applications*). JCHORUS is a programming language that is particularly suited to implement *cautious* algorithms.

The active behavior of an assembly i_1 is syntactically defined by *guarded updates* of the form

$$Guard : \{Stmt\}$$

where *Guard* is a condition that is evaluated atomically, and *Stmt* is a statement allowing imperative modification of the objects and pointers within the assembly.

Control flow in i_1 can be abstractly captured by a state machine (Figure 3.2) with three control states: *busy*, *ready*, and *terminated*. A newly created assembly starts from the *ready* state. State transitions are as follows:

- If i_1 is at the *ready* control state, then it nondeterministically chooses a guarded update, atomically evaluates its guard, and, if the guard is enabled,

moves to the *busy* control state to execute *Stmt*. As i_1 has exclusive access to its region, no extra precaution to ensure the atomicity of the update is needed.

- If i_1 is at the *busy* control state and has finished its update, then it can move back to the ready state.

- If i_1 is at the *ready* control state, then it can be *terminated*.

Unlike in other guarded-command languages, a guard here can, in addition to checking local boolean conditions, *merge* i_1 with an adjacent assembly i_2, taking i_1 to a *busy* control state and causing i_2 to *terminate*. In its new state, i_1 operates on the union of the regions previously comprising i_1 and i_2. The heap itself is not modified—e.g., no pointers are rewired. Also, the merge can happen only when i_2 is in a ready state. During the merge, i_1 can copy into its own local variables the local-variable state of i_2, thus acquiring the "work" that i_2 has already done.

As for updates, they permit an assembly to imperatively modify its region—any expression whose evaluation requires accesses outside H returns raises a programming exception. An update can also *split* an assembly into smaller ones—e.g., into assemblies containing one object each. Observe that the split *locally refines* the parallelism in the system.

Also note that for an assembly i_1 to merge with an assembly i_2, no explicit "consent" from i_2 is needed. All we require is that i_2 is not in the middle of an update at the point when the merge happens. Thus, assemblies are *speculative* entities that may not always finish the task that they set out to accomplish. At the same time, there is no notion of rollbacks in our model. Assemblies "commit" after every update, and once committed, updates are final. Therefore, applications must be coded so that the data within an assembly is in a *consistent state* at the end of each update.

Finally, note that our object model does not allow for aliasing (just as the abstract formal model). At any time, an object belongs to only one assembly (we do make an exception for *read-only data* in JCHORUS–see Section 3.1.2). While an assembly can have a *reference* to an object outside its region, it cannot use this reference for reading or writing.

Merging, splitting, and heap modification. Now we examine the concurrency related programming constructs that JCHORUS permits. Each assembly here is an instance of an *assembly class*, and programs are collections of assembly class definitions. A definition for the class τ specifies a set of guarded updates that assemblies of class τ can execute, as well as a set of *local variables* $\mathtt{u}, \mathtt{v}, \ldots$ that they use to refer to objects *within their regions*. Objects can also be referenced using field expressions in the usual way: if the variable `u` refers to the object u and u has an `f`-labeled edge to v, then the name `u.f` refers to v.

The simplest construct for merging two assemblies and executing an update on the merged assembly is

$$\texttt{merge}(\texttt{u}) : \{\mathit{Stmt}\}$$

Here `u` is a variable in the assembly i_1 executing the guarded update. Variable `u` refers to an object in a different assembly i_2. For the guard here to be *enabled*, i_2 must be in a ready state (i.e., it must not be currently executing an update). If it is, then its evaluation atomically terminates the thread of control in i_2, takes i_1 to the "busy" control state, and gives i_1 ownership of all objects previously owned by i_1 and i_2. The values of local variables of i_1 are not affected by the merge. The update *Stmt* is now executed.

Merges can also be constrained with boolean predicates. For example, we have the construct

$$\texttt{merge}(\texttt{u})\ \texttt{when}\ g : \{\mathit{Stmt}\}$$

where g is a predicate referring to objects in the region of i_1 and the local variables of i_1. The semantics is as before, except i_1 can execute the command only when g holds. The other kinds of merges are similarly generalized.

As for updates executed by an assembly i, they may *split* i into smaller assemblies, as well as imperatively modify objects in the region that i owns. For example we have the construct

$$\texttt{split}(\tau')$$

splits i at the finest possible granularity—for each object u in the region of i, a new assembly that is of class τ' and control state *ready*, and consists of the single node u, is activated. Each local variable of i_u is initialized to refer to u. The assembly i ceases to exist.

```
1: Mesh m = /* read input mesh */
2: Worklist wl = new Worklist(m.getBad());
3: foreach Triangle t in wl {
4:   Cavity c = new Cavity(t);
5:   c.expand();
6:   c.retriangulate();
7:   m.updateMesh(c);
8:   wl.add(c.getBad());  }
```

Figure 3.3. Delaunay mesh refinement: sequential algorithm

Note that assemblies never directly reference other assemblies during execution. In particular, merges happen on *object references*, not assembly references.

3.1.1 Example: Delaunay mesh refinement

Let us now go back to the Delaunay mesh refinement problem briefly mentioned in Chapter 1. Given a set of points $\mathcal{M}$, a Delaunay triangulation partitions the convex hull of $\mathcal{M}$ into a set of triangles such that: (1) the vertices of the triangles, taken together, are $\mathcal{M}$, and (2) no point in $\mathcal{M}$ lies in any triangle's circumcircle (the empty circle property). As discussed in Chapter 1, in many applications [16], there are further qualitative constraints on the resulting triangles. In order to meet these constraints, a Delaunay triangulation often needs to be refined. We blackbox the requirements and suppose there is a function that identifies "bad" triangles.

Pseudocode for a sequential algorithm for refining the mesh is in Figure 3.3. Initially, the worklist is populated by bad triangles from the original mesh. For each bad triangle t, the algorithm proceeds as follows[1]:

- A point p at the center of the circumcircle of the triangle is inserted.
- All the triangles whose circumcircle contains p are collected. These triangles form a contiguous region in the mesh called a cavity of t (Figure 3.4). As cavities are contiguous, a breadth-first search algorithm (`c.expand()`) touching only a region in the heap containing the bad triangle can be used to find a cavity.

[1]For ease of presentation, we suppose here that the bad triangle is not near the boundary of the whole mesh.

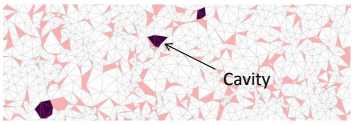


Figure 3.4. Snapshot of a small part of a Delaunay mesh

- The cavity is then retriangulated by connecting p with all the points at the boundary of the cavity (this is done by `c.retriangulate()`).

The Delaunay property (the empty circle property) is guaranteed to hold for the newly created triangles. The qualitative constraint may not hold for all the new triangles, so the size of the worklist might increase in certain steps. The algorithm, however, is guaranteed to terminate. Also, retriangulation of a cavity is a local operation, and the order in which the cavities are retriangulated is not important.

We have parallelized this application using JCHORUS (see Section 3.3). For now, we show how to capture its essence following the pseudocode in Figure 3.5. Here, the triangulation at any point is modeled by a heap whose objects are triangles, and whose pointers connect adjacent triangles. There are only one type of assembly: `Cavity`. Initially, every triangle in the heap is in its own assembly of class `Cavity`).

To simplify presentation, we assume that updates can call sequential subroutines such as `retriangulate`. We also assume that a `Triangle` can use a boolean variable called `isBad` that, at any point, is `true` iff it is bad, and that a `Cavity` can use a boolean method `isComplete()` that, at any point, is `true` iff it needs no further expansion. Finally, we let each assembly i have a local variable `v` whose value loops through the set of objects in i with an (`f`-labeled) edge to an adjacent assembly in the mesh. We abstract out the code maintaining these variables.

The code for our modeling is given in Figure 3.5. Here, each triangle (forming an assembly of class `Triangle`) checks if it is "bad." If it is, then it merges with an arbitrary neighbor to create a assembly of class `Cavity`. The expansion of a cavity—done in Figure 3.3 by the method `expand()`—is captured here by a series of merges among assemblies of class `Cavity`. Note that expansion is possible only when the cavity is not yet complete. If a cavity discovers that it is complete, it executes an update in which it first retriangulates its region, and then splits into

```
shared Triangle { ...
    Triangle n_1, n_2, n_3;
    Bool isBad;
 }

assembly Cavity { ...
  Triangle v;
  Bool isBad;

  constructor(Triangle t) { v := t.n_*; isBad := t.isBad }

  actions {
    merge (v, Cavity) when isBad:
                {if isComplete()
                          then {retriangulate(); split(Triangle)}
                          else  v = chooseAdjacentTriange()
                }
          }
  }
}
```

Figure 3.5. The essence of Delaunay Mesh Refinement in JCHORUS

its component triangles. The program terminates when the only assemblies in the mesh are `Cavitiess` containing a single "good" `Triangle`.

It is worthwhile to note the *speculative* nature of cavities in the above encoding. As cavity expansion is captured using a series of merges, a cavity may reach its ready state multiple time as it expands. Therefore, the cavity can be destroyed by another cavity before it has a chance to retriangulate itself. At the same time, such early termination of cavities does not leave the heap in an inconsistent state—the only rewriting of the heap happens within the method `retriangulate()`, which is executed atomically in one update.

3.1.2 JCHORUS features not available in the Concurrent Assemblies abstract model

JCHORUS contains several features that go beyond the *Concurrent Assemblies* model.

Sequential and parallel phases. Most realistic parallel applications contain

stages that are inherently sequential, during which the application does I/O, constructs global data structures, iterates over totally ordered sets, etc. While it is theoretically possible to model such sequential phases by global synchronization where the entire heap is merged into one assembly, such an encoding will be inefficient in practice.

Consequently, we let JCHORUS programs have both sequential and parallel phases. The former is performed by the sequential Java program into which our concurrency constructs are embedded. The sequential phase can *invoke* a parallel phase by the call `parallel`$(\tau(\mathtt{p}_1, \ldots, \mathtt{p}_k))$ which constructs a assembly of type τ ($\mathtt{p}_1, \ldots, \mathtt{p}_k$ are expressions passed as actual parameters to the constructor) whose region is the whole heap. When all assemblies in the parallel phase terminate, control returns to the sequential program once again. For a sample application for this construct, see our code for Barnes-Hut simulation (Figure 3.11).

Read-only data. Read-only data that can be simultaneously accessed by multiple threads is an important source of concurrency in many parallel applications (e.g., in Barnes-Hut n-body simulation). In *Concurrent Assemblies*, for simplicity, we did not account for such objects, which is why an object was required to belong to only one assembly at one time. In the JCHORUS programming language, however, we permit shared objects to be classified as read-only via a special type qualifier. Read-only objects can be accessed freely by multiple assemblies, and attempts to modify them cause exceptions.

Assemblies can *cast* writable objects that they own into read-only objects. For example, this is done by the call to `node.setReadOnly()` in our modeling of Barnes-Hut simulation (Figure 3.11). This is because in JCHORUS any writable object that an assembly i can access is guaranteed to be exclusively owned by i. However, the reverse, if carried out during a parallel computation, may cause races. Therefore, we permit read-only data to be cast to a writable type only in a sequential phase of the computation.

Now we present JCHORUS more formally as a translation to the *Concurrent Assemblies* abstract model. As our goal is to capture the essence of *concurrency* in *Concurrent Assemblies*, assemblies here do not call methods on objects or create new objects, and only execute straight-line code. Also, we let objects be untyped, assume that all objects are shared, and do not account for read-only data that can

be freely accessed concurrently. Finally, objects do not have distinct data fields—all data is encoded by pointers and accessed by fields, which programs are allowed to update. These restrictions are all lifted in the full language.

3.2 Formal Semantics of JCHORUS

Here we present the syntax and semantics of JCHORUS.

3.2.1 Syntax

Prog	::=	$[AssemblyDecl \mid NodeDecl]^*$
AssemblyDecl	::=	`assembly` *id* $\{UnaryConst\ MemberDef^*$ `actions` $\{\ [Guard : Stmt]^*\}\}$
NodeDecl	::=	`shared` *id* $\{MemberDef^*\}$
MemberDef	::=	*VarDef* | *FuncDef*
Guard	::=	`merge(v)` | `merge(v)` `when` *Bexp* | *Bexp*
Stmt	::=	`v` := *Exp* | `v.f` := *Exp* | `v.f` := `new` *id* | `v.f` := `null` | *Stmt*; *Stmt* | `skip` | `split`(*id*) | $\{Stmt\}$ | `if` *Bexp* `then` *Stmt* `else` *Stmt*
VarDef	::=	*id id*;
FuncDef	::=	*id id* $([[id\ id,]^*id\ id])\{Stmt\}$
UnaryConst	::=	`constructor`$(id\ id)\{Stmt\}$
Exp	::=	`v` | *Exp*`.f` | `error`
Bexp	::=	*Exp* = *Exp* | `not` *Bexp* | *Bexp* `and` *Bexp*

where
`v` $\in$ *Var* and `f` $\in F$. Syntax within square brackets is optional, and $[t]^*$ denotes zero or more occurrences of t.

Figure 3.6. Syntax of JCHORUS

Let us assume a universe *Var* of *assembly variables*, and a universe $\mathcal{T}$ of *assembly classes* that contains a designated *initial class* ι. The syntax of programs in JCHORUS is given in Figure 4.4. Here:

- *Prog* represents programs.

- *AssemblyDecl* represents declarations of assembly classes. The declaration consists of a sequence of *local variable and method declarations* and definitions of *guarded updates*.
- *NodeDecl* represents declarations of shared object classes.
- *Guard* represents *guards*; *Stmt* represents *updates* and *splits*.
- *UnaryConst* represents the constructor (used to bind local variables initially).
- *Exp* and *Bexp* respectively represent pointer expressions and boolean expressions.
- *FuncDef* and *VarDef* respectively represent member function definitions and local variables declarations.

We require that each variable used in a pointer or boolean expression is declared within the relevant assembly class, that no assembly class is declared twice, and that there is a declaration for the initial class ι. Also, we denote:

- the set of local variables in the class τ by $Var(\tau)$
- the set of guarded updates in τ by $Act(\tau)$.

3.2.2 Semantics

Here we give a formal semantics for JCHORUS. The main idea is to map *assemblies* from JCHORUS directly to *concurrent assemblies* from the abstract model. We define a single type, i.e. $\mathcal{T} = \{\tau\}$, that will be used for all assemblies. The idea is that at the top of the actions allow nondeterministically the choice of one guard or accept a `merge`.

The isolation aspect of an execution of a JCHORUS program is translated directly into the local update operations in the Concurrent Assemblies formal model semantics. For example a local update `u.f = v` in JCHORUS is translated into an update of the form `u.f := v` in Concurrent assemblies.

Let $[\![.]\!]$ be the translation function from JCHORUS into Concurrent Assemblies. In the sequel we sketch the function $[\![.]\!]$:

$$
\begin{array}{lcl}
[\![\texttt{u.f = v}]\!] & \stackrel{\text{def}}{=} & [\![\texttt{u.f}]\!]_{exp} \texttt{ = } [\![\texttt{v}]\!]_{exp} \\
[\![\texttt{u = v.f}]\!] & \stackrel{\text{def}}{=} & [\![\texttt{u}]\!]_{exp} \texttt{ = } [\![\texttt{v.f}]\!]_{exp} \\
[\![\texttt{u = new } id]\!] & \stackrel{\text{def}}{=} & [\![\texttt{u}]\!]_{exp} \texttt{ = new} \\
[\![\texttt{u = null}]\!] & \stackrel{\text{def}}{=} & [\![\texttt{u}]\!]_{exp} \texttt{ = null} \\
[\![Stmt_1\texttt{;}Stmt_2]\!] & \stackrel{\text{def}}{=} & [\![Stmt_1]\!]\texttt{;}[\![Stmt_2]\!] \\
[\![Guard_1 : Stmt_1 \cdots Guard_n : Stmt_n]\!] & \stackrel{\text{def}}{=} & ([\![G_1]\!]\texttt{;}[\![Stmt_1]\!])\texttt{+}\cdots\texttt{+}([\![G_n]\!]\texttt{;}[\![Stmt_n]\!])\texttt{+}\overline{\texttt{merge}}(\epsilon) \\
[\![\texttt{merge(u)}]\!] & \stackrel{\text{def}}{=} & \texttt{merge}([\![\texttt{u}]\!]_{exp}, \tau, [\![Stmt]\!]) \\
[\![\texttt{merge(u) when } Bexp]\!] & \stackrel{\text{def}}{=} & [\![Bexp]\!]_{bexp}\texttt{?merge}([\![\texttt{u}]\!]_{exp}, \tau, [\![Stmt]\!])
\end{array}
$$

Figure 3.7. Sematics of JCHORUS

Our implementation of JCHORUS supports termination detection (see Section 5).

3.2.3 Race- and deadlock-freedom

Race-freedom. A *data race* happens when two concurrent threads concurrently access a shared object, and at least one of these accesses is a write. Updates in JCHORUS are data-race-free as they operate on disjoint regions in the heap. As for merges, our semantics guarantees that an assembly can merge only with assemblies that are in the *ready* control state—i.e., assemblies where control is not inside an update. Thus, programs in JCHORUS are free of data races.

Deadlock-freedom. Recall the classical definition of a deadlock: a deadlock arises when a process i_1 waits for a resource from a process i_2 and vice-versa, preventing the system from progressing. To see what deadlocks mean in our setting, consider a guard g that involves a merge. This guard is *locally enabled* on the edge $(u, \texttt{f}, v)$ out of an assembly i_1, in state N, if one of the following holds: (1) g does not use a boolean constraint, and (2) g uses a boolean constraint, and this constraint evaluated to `true` in N. The guard g is *enabled* if, in addition, the assembly i_2 containing v is currently in a ready state.

We can now adapt the classical definition of deadlocks as follows: "A deadlock arises when assemblies i_1 and i_2 are *both* forbidden from further progress for the following reasons: (a) i_1 has a locally enabled merge along an edge into assembly i_2, and vice-versa. (b) Neither i_1 nor i_2 can progress otherwise—i.e., they do not have other enabled guards."

Assuming the evaluation of each update and boolean expression terminates, the above deadlock scenario is impossible in our setting. This is because in JCHORUS an assembly in a ready control state cannot prevent itself from being merged even if it has locally enabled merges by virtue of the translation defined in the previous subsection. Therefore, in a scenario as above, the runtime system nondeterministically executes one of the requested merges (say the one invoked by i_1), causing i_2 to terminate, and bringing its entire region in the possession of i_1.

Of course, the above justification, while reasonable for an interleaving semantics implemented on a uniprocessor, is unsatisfactory in the distributed context lacking a centralized runtime system, which is what we target. Deadlock-freedom in this setting can be established with a language specification closer to the implementation. See Section 3.4 for more details.

3.3 Case studies

In this section, we show how to use JCHORUS to program our flagship example—Delaunay mesh refinement—as well as Boruvka's minimum-spanning-tree algorithm [14], Barnes-Hut n-body simulation [10], "focused community" discovery in social networks [11], and a problem of agent-based epidemiological simulation [13].

3.3.1 Delaunay Mesh Refinement

Now we describe in detail the implementation of Delaunay Mesh Refinement in JCHORUS. Section 3.1.1 describes the main ideas behind this algorithm using an implementation. This example illustrates some of the constructs available in JCHORUS that are not present in the core version of the language. Figure 3.1 shows the interesting snippets.

Lines 40–44 contain the declaration for the shared object `TriangleObject`, as one would expect it contains its data and some member functions. Lines 50–56 describe the `Loader` assembly which will be the starting point for this program. The `Loader` assembly will read a file (using standard Java constructs), create all the `TriangleObject`s, and end by splitting into `Triangle`s assemblies, one for each `TriangleObject`. Lines 1–6 implement the assembly `Triangle`; if a triangle

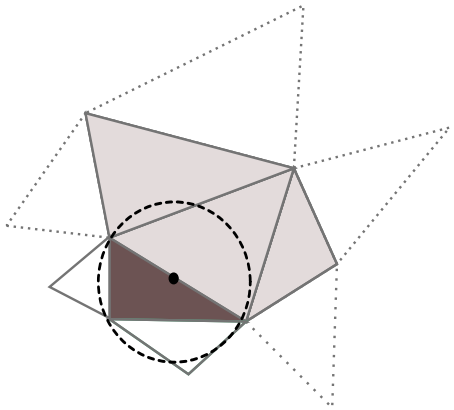


Figure 3.8. A cavity in construction, its center triangle (in dark), the cavity triangles (light gray) and the border triangles (dotted lines). The white triangles are either part of the frontier or the destination of one of the outgoing edges.

is "bad" then the `Triangle` assembly becomes a `Cavity` assembly (line 4). As is the case in with objects in object oriented programming, assemblies have constructors that are executed upon creation.

The core of the algorithm resides on Lines 7–39, the `Cavity` assembly. The `Cavity` assembly will build the cavity by traversing `TriangleObject`-s in a breadth first fashion, it starts by placing the initial "bad" triangle into the `frontier` queue (lines 18–20) and start the exploration from there.

The core construction of the cavity (lines 21–35) is done in the `build()` method. It starts by taking a `TriangleObject` object from the frontier queue (line 23), checking whether it belongs to the cavity or is a border triangle (lines 25–26), and add its neighbors to the `frontier` (lines 27–28). If the `TriangleObject` was not local to the assembly an exception is raised and catched (lines 24, 29–30), and a merge will be requested on this `TriangleObject`. If all the cavity was explored then it proceeds to retriangulate (lines 32-34) and split into `Triangle` assemblies.

Up to here we have only described the local aspects of the algorithm. The interaction code (lines 8–13) requests a merge on any of the `outgoingedges`, and upon merging with an adjacent assembly (line 9), variable `t` is instantiated to the `TriangleObject` selected for the merge, and expansion continues through a call to `build()`.

Observe that most of the code is identical to the pure sequential version. The only part needing concurrent programming are the lines 8–13.

3.3.2 Minimum-spanning tree

Now we use JCHORUS to code a classic, irregular graph algorithm well-known in the parallel programming literature [17, 18] : Boruvka's algorithm for computing the minimum spanning tree of a graph [14]. The main idea of the algorithm is to start with spanning trees that each consists of a single node, then combine these small trees into larger and larger trees. At the end we are left with a single, provably optimal spanning tree.

The input here is a weighted graph. To model edge-weights, we let each node u in the graph have a data field `outEdges`—of type "list of integers"—that contains the weights of all edges out of u. Each spanning tree obtained during the computation is captured by an assembly (initially, each such assembly contains a single node). A spanning tree maintains a minimal outgoing edge—i.e., an edge of minimal length out of those that lead to a node outside of the spanning tree.

Every spanning tree always has a merge-operation enabled along a minimal-weight edge. When a spanning tree m_1 merges with a spanning tree m_2, they form a new spanning tree m_3. The set of nodes of the spanning tree m_3 is the union of nodes of m_1 and m_2; the edge set consists of the edges of m_1 and m_2, as well as the edge on which the merge happened. The minimum outgoing edge from m_3 needs to be found as well—this is done using a method `computeNewMinEdge()`.

It can be shown that this greedy algorithm constructs an optimal minimum spanning tree. Once again, merging and local updates are the only operations needed. JCHORUS code for it is presented in Figure 3.9.

3.3.3 Barnes-Hut n-body Simulation

Now we consider the Barnes-Hut algorithm for n-body simulation [10], an application in the Lonestar benchmarks [19]. The algorithm simulates movements of and gravitational forces among N bodies (for example stars grouped in galaxies). The simulation algorithm proceeds in timesteps. Every timestep, in order to calculate the new position where each body moves, the gravitational forces they exert on each other need to be calculated. This suggests an $O(n^2)$ algorithm, which for each body directly calculates the force exerted upon it by all the other bodies.

Barnes and Hut [10] introduced an $O(n \log n)$ time based on the idea that

```
 1: shared Node {
 2:   List edges;
 3:   List weights; }

 4: assembly ComputeSpanningTree {
 5:    Tree currentTree;
    // set of outgoing edges, sorted by weight
 6:    SortedList outgoingEdges;
 7:    Edge minOutEdge;

 8:    action {
 9:       merge(minOutEdge) :
10:          computeNewMinEdge();
11:    }
    // computes the minimal edge of the merged tree
12:    void computeNewMinEdge() { ...}
   }
```

Figure 3.9. Minimum spanning tree

the forces that a group of bodies exerts on a body laying at a sufficiently far distance can be approximated by placing all mass at single point (center of gravity) representing the group.

The Barnes-Hut algorithm proceeds in timesteps, similarly to the quadratic algorithm sketched above. The computation in every time step has two phases. First, the bodies are inserted into a data structure called octree. Second, the data structure is used for efficient computation of the forces between the bodies.

An octree is a tree in which each internal node has eight children. Each node represents a cube in three-dimensional space, and the children of an internal node represent the division into eight sub-cubes. Each leaf of the octree contains exactly one body. The insertion of a body b starts at the root and proceeds towards the descendants until a leaf l representing a region where b is physically is found. By construction, l contains exactly one other body b_2. The leaf is therefore successively subdivided until b and b_2 are in different leaves. Upon construction, the octree is summarized, i.e. for each of its internal node the center of gravity is calculated.

Once the octree is constructed, the forces on each body b can be calculated. The octree data structure is traversed again. For cells C that are close to b, the force between b and all bodies in C is calculated. On the other hand, for cells D

that are sufficiently far away, a single point can be used to approximate the force - the center of gravity for the cell. Note that during the computation of the forces, that octree is read-only, that is, it is not being modified.

The pseudocode for the sequential algorithm is in Figure 3.10. For more details on the algorithm and the octree data structure, the reader is referred to [10].

The computations in different timesteps are not possible to parallelize, as the computation in a timestep depends on the results of the previous timesteps. However, every timestep is itself computationally intensive and conditions for parallelizing computation inside a timestep are much better. Both of the phases mentioned above, the insertion into the octree, and the computation of forces can be beneficially parallelized.

The JCHORUS code for the application needs to use both sequential and parallel computation. The outer loop from Figure 3.10 stays conceptually the same – timesteps are executed serially. Figure 3.11 shows the essential snippets from our JCHORUS implementation.

Lines 1-9 show the outer timestep loop. Line 7 and Line 8 are the invocation of the two parallel phases; each of which start from a single assembly but will split, each of which will end when all its assemblies become inactive.

Following the style of the implementation in the Lonestar suite, our encoding separates the construction of the octree from its summarization. The construction of the octree is done sequentially and is implemented in `OctreeNode` (line 12). The second stage, the summarization of the octree, is the most interesting in JCHORUS, where irregular parallelism is present. Each `OctreeNode` maintains whether its data has been propagated up and whether it has received updates from all its children. This second stage starts by a `ComputeMass` assembly containing each of the nodes (lines 14-22). The `ComputeMass` assembly will merge with its parent if the node is completely updated and has not been yet propagated. When a node is completed, it is made *read-only* (line 19), and can be accessed without any restriction by any assembly (a crucial fact for the third phase).

The third phase consists of computing of forces acting on each body. This is done by starting in a parallel phase where teach body sits in its own `UpdateBody` assembly (lines 23-30). Each `UpdateBody` assembly needs only traverse the octree and update the body it contains. As no merge requests are involved in the process,

```
 1: List bodylist = ...
 2: foreach timestep do {
 3:   Octree octree;
 4:   foreach Body b in bodylist {
 5:     octree.Insert(b);
 6:   }
 7:   octree.SummarizeSubtrees();
 8:   foreach Body b in bodylist {
 9:     b.ComputeForce(octree);
10:   }
11:   foreach Body b in bodylist {
12:     b.Advance(); } }
```

Figure 3.10. n-body simulation

this process is all implemented in the constructor (lines 25-29).

3.3.4 Agent-based models in epidemiology

A typical question in epidemiology is what type of vaccination distribution is effective to prevent an outbreak from developing to a full scale epidemic. It is well-known that agent-based modeling that enables different type of interactions between agents has advantages over models that assume that there is a uniform probability of any two agents meeting. A more detailed model allows capturing the fact that agents interact only with a certain number of people, those that they meet at home, at their workplace, etc. The survey [13] describes several such approaches used for modeling the spread of the smallpox virus.

We consider the model of Burke et al. [20]. It simulates how a virus can spread from a single infected person throughout a small network of towns. Each town consists of one hospital, one school, one other workplace and households of up to seven people. The model extends the interaction assumptions further and has every agent interacting with the same (up to 8) people in public places such as the schools and hospitals. During a "day," an agent interacts with all of its immediate neighbors (a fixed number of times, different for each type of community). Transmission of a virus occurs only during these interactions.

The computations necessary are thus again purely local, and can be naturally captured in JCHORUS. The agents have a fixed number of neighbors, up to eight

```
 .....
 1: void main() {
 2:   List bodylist = ...
 3:   foreach timestep do {
 4:       OctreeNode root = ...
 5:       foreach body in bodylist
 6:           root.insert(body);
 7:       parallel(ComputeMass(bodyList));
 8:       parallel(UpdateBody(bodyList)); }
 9: }

10: shared OctreeNode {
11:    ...data variables...
12:    void insert(...) ...
13: }

14: assembly ComputeMass {
15:    action {
16:       merge(node.parent)
17:          when (!node.propagated?
                    && node.complete?):{
18:             propagateUp();
19:             node.setReadOnly();
20:             split(ComputeMass); }

21:    void propagateUp() { .... }
22: }

23: assembly UpdateBody {
24:    action {}
25:    UpdateBody(OctreeNode n)
26:    {
27:         f = computeForce(n, root);
28:         n.advance(f);
29:    }
30: }
```

Figure 3.11. n-body simulation in JCHORUS

per each environment (home, school) in which they interact. An interaction is modeled by a merge, update (if one of the person is infected, a virus is probabilistically transmitted), and a subsequent split. Modeling the interactions in this way lets us express parallelism at the level of agents and their interactions. Ar-

```
core := R;
changed := true;
while changed do {
  changed := false;
  fringe := neighbors(core);
  for each v in core {
     if obj(core - {v}) < obj(core) {
       core := core - {v};
       changed := true;  } }
  for each v in fringe do {
     if obj(core union {v}) < obj(core) {
        core := core union {v}
        changed := true;
     }  }
}
```

Figure 3.12. Focused Communities

guably, this also captures the natural parallelism of real-life social networks, where information creation and propagation happen bottom-up rather than top-down.

3.3.5 Focused community discovery

A typical problem in analyzing social networks is focused community discovery. Given a person p, the task is to discover the community to which the person belongs. The community around p is intended to capture information flows in the network, thus we are interested in finding a set of people that contains p and that is robust - i.e. connection between them are stronger then their connection to the outside world. How to discover communities efficiently is a topic of current research (see [11, 12]). A data set connected to this problem is a part of the Lonestar Benchmark Suite.

We consider an algorithm for focused community discovery from [11]. Figure 3.12 has the pseudocode for the algorithm. The algorithm greedily optimizes an objective (given by the function `obj`). The algorithm keeps its current hypothesis for the community (the `core` set). The `fringe` set is the set of adjacent nodes, that is nodes that are not in the core, but are directly connected to it. Figure 3.13 has a picture of the algorithm in progress. At each step, the algorithm checks:

- For each node in the core, whether removing this node would increase the

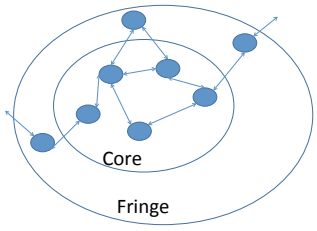


Figure 3.13. Focused Communities: the core and the fringe.

objective. If so, the node is removed.

- For each node in the fringe, whether including this node would increase the objective. If so, the node is added to the core.

The process continues until no change occurs or a cycle is detected.

Let us suppose that we are given an input stream of requests, each consisting of an individual p and an update that needs to be performed on all the members of p's community, for example about an announcement p makes. These requests can be processed in parallel. There will be an assembly called `Community` whose code will closely follow the code in Figure 3.12. The only major difference is that the set union and set difference operations need to be implemented using merges and splits.

It so happens that the code resulting from the above approach is quite similar to the pseudocode in one of the original references for this problem [11]. Yet, that paper was concerned with the algorithmic rather than the programming-language aspect of focused community discovery. We view this as evidence that JCHORUS is a natural programming model for this application.

3.4 Implementing JChorus

The assembly-level parallel exposed by JCHORUS is potentially massive. For example, running the refinement algorithm in Figure 3.3 on a mesh of over 100,000 triangles, we found the average cavity size to be only 3.75, and the maximum cavity size to be 12. Consequently, any runtime system for JCHORUS must perform a many-to-one mapping of assemblies to the limited number of processor cores. Managing such a mapping is a non-trivial task leaving room for many different

approaches to design and optimization. Now we present an implementation of JCHORUS based on one such approach.

3.4.1 The JCHORUS compiler and runtime system

Our implementation of JCHORUS consists of a prototype compiler and the corresponding runtime system. The compiler translates JCHORUS programs into Java programs running on the JCHORUS runtime system. The JCHORUS runtime system consists of a lightweight layer on top of the Java runtime system implementing scheduling, object migration and thread synchronization policies. The tool flow of the system is outlined in Figure 3.14.

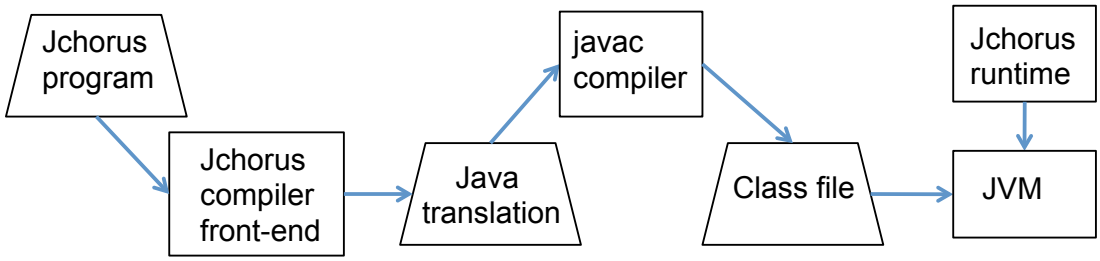


Figure 3.14. The JCHORUS compiler and runtime system

Now we outline the main features of the JCHORUS runtime system.

3.4.1.1 Divisions

The key low-level abstraction in our system is a *division*: a set of proximate assemblies in the heap that is processed as a unit by parallel processors. The job of the JCHORUS runtime is to maintain the divisions in the heap as the execution progresses, and to map them to lower-level (Java) threads. These threads are then mapped to hardware threads by the underlying Java runtime environment.

Divisions partition the heap at a coarser granularity than do assemblies—while the heap may have hundreds and thousands of assemblies at any time, it typically has only as many divisions as there are hardware threads. The abstraction is only available within the runtime system, and is invisible to programmers. Figure 3.15 shows a heap and some divisions in it. The initial divisions are roughly equal in size, and are obtained heuristically from the input dataset.

3.4.1.2 Executor threads and token-passing

The JCHORUS runtime system assigns a native thread of execution (known as the *executor*) to each division. The executor for a division D (pseudocode in Figure 3.16) maintains two queues $Workq(D)$ and $Blockq(D)$ storing the *active* assemblies that are currently in it.

In each iteration of a top-level loop (lines 1–33), the executor selects an assembly i from the *work* queue $Workq(D)$ (line 15) and tries to acquire a compare-and-swap lock on it. If it fails, it must be because an assembly in a different division is merging with i. If it succeeds, it processes a subset of the guarded updates of i in a round-robin manner, executing their guards and, if the guards are enabled, their updates. This subset of guarded updates to be executed is chosen heuristically. In some applications, better performance comes from executing all guarded updates in the assembly under consideration (we call this the *depth-first execution strategy*), while in some others, it is better to deschedule the assembly after executing one or two of them (we call this the *breadth-first strategy*).

As for the execution of guarded updates (lines 18–27), the interesting case is when the guard involves a *remote merge* by i on a reference `o` (i.e., when the object `o` is in a different division). In this case, the executor does not carry out the merge right away, instead placing the pair $(i, \texttt{o})$ in a second division-level queue $Blockq(D)$ storing assemblies *blocked* on remote merge requests. Such blocked assemblies are temporarily removed from the work queue.

Local merges (i.e., merges within the same division) are carried out immediately. Splits produce a set of assemblies that are kept local to the division. If a split happens, the new assemblies are put into $Workq(D)$ and the next top-level iteration starts. Otherwise, once the selected subset of updates in i are executed, i is unlocked, and if it is still active (this can be checked during guard evaluation), put back into $Workq(D)$.

The JCHORUS runtime uses a simple token-passing strategy to manage remote merges. Divisions are assumed to be arranged in an implicit ring in which a single global token for remote merges is passed. At the beginning of each top-level iteration, the executor thread for D checks if D currently owns the token. If so, a certain number k of remote merges are carried out in a batch (lines 5–12; see Section 3.4.1.3 for explanations on Line 10). In case the target of a remote merge

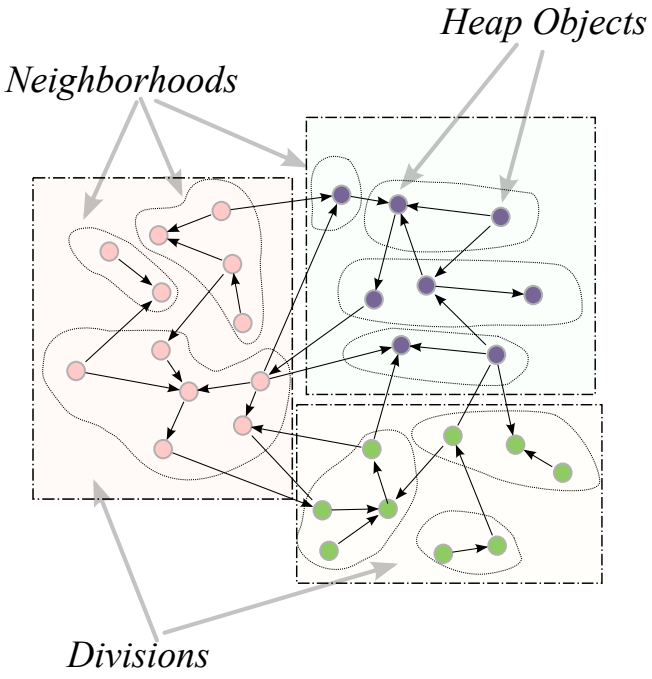


Figure 3.15. Divisions in a heap

is busy, the assembly seeking the merge is put back into $Blockq(D)$.

Note that the only synchronization needed for atomically executing an assembly's action is the lock acquisition on the assembly; we do not have locks on objects. This is because our assemblies can only access objects in their own regions.

A simple *termination detection* strategy (lines 3–4) is woven into the remote access phase of the executor. Here, we let the global token track the *last active division* that owned it. If D finds that its work and blocked queues are empty (i.e., all assemblies in it are inactive) and it is the last division to have accessed the token, then the current parallel phase of the program (recall that JCHORUS allows parallel and sequential phases) has reached a terminating state.

Deadlock-freedom. Recall that by our definition, a deadlock arises when two assemblies i_1 and i_2 both have locally enabled merges along an edge, are unable to proceed, and yet are not terminated by a merge.

If i_1 and i_2 are in the same division D and i_1 is picked by the executor of D, then i_1 can merge with i_2 *unless* an assembly in a different division is merging with

```
1: while (true)
2:     if (ownsToken())
3:         if (token.last = D and Blockq(D) = empty
                  and Workq(D) = empty)
4:            terminate;
5:         repeat k times
6:            (i,o) := dequeue (Blockq(D));
7:            if Assembly(o) is busy
8:               enqueue (Blockq(D), (i,o));
9:            else
10:              migrate assemblies;
11:              performMerge(i,o);
12:              enqueue (Workq(D), i);
13:        update token.last;
14:        passToken(D.next());
15:    i := dequeue (Workq(D));
16:    if (locking i fails)
17:        continue;
       /* We have assembly i locked */
18:    for act in i.action.guardedupdates()
19:        if act.guard is false
20:            continue;
21:        if act.guard involves merge with
                                remote object o
22:            enqueue (Blockq(D),(i,o));
23:              break;
24:        else if act.guard involves merge with
                                local object o
25:            performMerge(i,o);
26:        execute act.update();
27:        if i was split then break;
28:    unlock i;
29:    if i was a split
30:        enqueue all children in Workq(D);
31:    else
32:        if i was not found to be inactive or
               added to Blockq(D)
33:        enqueue (Workq(D),i);
```

Figure 3.16. Executor thread loop for division D

i_2 when i_1 attempts the merge. In the latter scenario, i_2 is nonexistent from this point on, so that there is no deadlock.

Now suppose i_1 and i_2 are respectively in distinct divisions D_1 and D_2, and D_1 possesses the token. One scenario is that i_2 is in the blocked queue of D_2. In this case, it is not in the work queue of D_2, which means that it cannot be currently locked by D_2 (since D_2 does not have the token, it cannot be executing Lines 6–8 using i_2 either). In this case, the merge goes through.

The other scenario is that i_2 is currently in the work queue of D_2. In this case, it is possible that i_2 is currently locked by D_2. However, in this case the executor of D_2 will eventually discover that i_2 has an enabled remote merge, and will send it to its blocked queue, and the earlier scenario will take effect.

Simple scheduling. As an alternative to the above execution strategy, we allow for a *simple scheduler* that does not offer the guarantee of deadlock-freedom, but has lower overheads. Here, a division does not carry a queue of blocked assemblies. Instead, all assemblies in it are kept in its work queue, and are selected and executed in a round-robin manner.

In one of the two applications on which we ran experiments (Boruvka's algorithm), this strategy led to significantly better performance due to lower overheads. At the same time, this scheduler has the potential of livelock, as two divisions can try to repeatedly, and simultaneously, merge assemblies that have edges into each other (as each assembly is locked, neither of these merges succeeds). Of course, as divisions typically contain numerous active assemblies, such livelocks perhaps extremely unlikely. In particular, we did not face livelocks in our experiments with this scheduler.

Perhaps this scheduler can also benefit from *randomized* strategies for livelock avoidance. Here, after an unsuccessful attempt to merge along an edge, a division waits for an increasing, randomly chosen time span before locking the assembly in question again. The purpose is to *break the symmetry* between attempts by different divisions to synchronize. We leave this direction for future research.

3.4.1.3 Assembly migration and load-balancing

To cope with the dynamic nature of irregular applications, divisions in our setting need to be *dynamically reconfigurable*. To minimize communication overheads, our goal is to maintain the divisions so that the ratio of remote accesses to local accesses is as low as possible. Therefore, we try to ensure that for most assemblies i, if i is in a division D, then so are most of its neighbors.

We achieve the above goal through a simple heuristic (invoked at Line 10 of Figure 3.16). Consider a merge between a assembly i and an assembly i', respectively on divisions D and D' (where $D \neq D'$) along an edge $(u, \mathtt{f}, v)$ in the heap. Every time such a merge happens, the assembly i', as well as all assemblies in D' reachable from v by a certain number m of pointer indirections, are *migrated* to D. The intuition is that just as updates in our applications happen in local regions in the heap, merges typically happen in local regions in the assembly graph. The heuristic dynamically adapts divisions to exploit this locality. The quantity m is a parameter for the runtime system, and is known as the *indirection depth*.

In fact, we enrich the above heuristic with a simple form of *load-balancing*. Observe that in the above scenario involving i and i', there is no inherent reason why the assembly migration has to be from D' to D and not the other way round. This observation leads to a load-balancing heuristic. Each division now keeps track of the number of assemblies in it. During remote merges, migration always happens from the division with a greater number of assemblies.

3.4.1.4 Use of Union-Find

The Concurrent Assemblies abstract model requires that we keep track of the relationship between objects and object assemblies. Given a reference to an object, the runtime needs to determine the assembly to which it belongs. This information is needed for merges, which are requested on object references. It is also needed whenever an assembly dereferences an object to perform some operation, as the runtime must then determine if the object belongs to the assembly. Additionally, assemblies need to determine the complete set of objects that they own. This information is necessary when an assembly ends its execution by splitting, where each owned object becomes the single object of a new assembly.

The choice of data structures for maintain ownership information can impact the performance of the runtime in a significant way. The operations that are at the heart our model, namely split, merge and object dereference, impose different constraints. Consider, first, the merge and object-dereference operations. Given the fact that assemblies constitute disjoint sets of objects, these operations can be supported by an ownership directory represented using Tarjan's Union-Find data structure. Our implementation allows the optional use of this structure.

Here, given a number of disjoint sets, only two operations are performed on them: (1) set union (needed for merges), and (2) set membership test (needed for ownership lookups). The problem has a complexity lower bound of $\Omega(n + m\alpha(m + n, n))$ for a sequence consisting of $n - 1$ unions and m finds; where α is an inverse of Ackermann's function [21]. If we consider single operations, there is a tradeoff between the cost of the union and that of the find. This lower bound is achieved by representing each set as a tree, where each element has a pointer to its parent (and no pointers to its children), and implementing the operation using two techniques: (1) union by rank and (2) path compression [21].

In order to implement the split operation efficiently, objects are embedded in a linked-list structure. This allows to perform the join of two list in $O(1)$ thus not affecting the cost of the merge. The object list must be traversed to perform the split.

3.5 Experiments

We have performed a preliminary experimental evaluation of our approach using the Delaunay mesh refinement and Minimum Spanning Tree applications. These applications were chosen because they are non-trivial and of practical importance, and because they produce results that can be examined for correctness (the last of these is not true, for example, for the Focused Communities application).

We wrote complete implementations of these two applications in JCHORUS. The baseline for performance comparisons was sequential Java. One of the competing approaches used hand-coded, fine-grained locking. In addition, we wanted to compare our performance with that of a well-established, publicly available, Java-based system for high-level parallel programming. We settled on DSTM2,

a state-of-the-art library of software transactions supporting dynamic-sized data structures [22, 23].

The experiments were run on an 8-core (dual quad-core) Intel Xeon X5550 with 2.66Ghz CPU speed per core, 24GB memory, a 64-bit Linux operating system (kernel version 2.6.18), and Java version 1.6.0_14, 64-bit.[2]

3.5.1 Delaunay Mesh Refinement

The first set of experiments involved the Delaunay mesh refinement application. Three datasets for this application are available in the Lonestar benchmark suite [19]—we selected the largest one. The dataset defines a 2-dimensional input mesh containing 100,364 triangles, approximately half of which do not satisfy the desirability criteria.

The JChorus implementation used in the comparisons was as in Section 3.3.1 and Figure 3.1. The initial partitioning into divisions was spatial—i.e., the mesh resides in a unit square and is recursively subdivided in four as many times as needed to equal or exceed the number of threads. The token-based scheduler in Figure 3.16 was used, with a depth-first execution strategy. Assembly migration was performed with an indirection depth of 4. The union-find data structure was not used. For comparison purposes, we defined a *conflict* to be a cavity that started executing, but could not finish its work due to being merged by adjacent ones.

For the sequential implementation, we started with an implementation available in the Lonestar benchmarks and hand-optimized it, removing some inefficiencies. In the fine-grained-locking-based implementation, locks were acquired at the triangle level. Each thread tried to acquire locks as it built a cavity in a non-blocking fashion. If it could not acquire a lock on a triangle, then it released all triangles it possessed, placed the bad triangle at the end of the worklist, and picked a new bad triangle to work with. Each such incident was considered to be a conflict.

Finally, the DSTM2 implementation was derived from the lock-based implementation in the most natural way possible. As DSTM2 permits a number of different transaction managers, the one with the best performance in this application was chosen. A conflict here is an aborted transaction.

[2]The source code of JChorus and the datasets used in the experiments are available at `http://www.cse.psu.edu/~swarat/chorus`.

As we worked on an 8-core system, we ran these competing approaches varying the number of threads from 1 to 8. Figure 3.17 depicts these comparisons. The first set of measurements compared the speedup over the baseline sequential implementation. We observe that we only start getting speedups over the sequential code at 8 cores (though DSTM does far worse). We ascribe this to the fact that JCHORUS works as a layer over Java, does not have direct access to the JVM or hardware threads, and makes expensive method calls for various runtime checks. In future, lower-level implementations, these overheads will hopefully come down.

The next set of measurements compared the *self-relative speedup* of the competing approaches—i.e, the ratio of the performance of an approach at n threads, to that of the *same approach* at 1 thread. This measure tries to eliminate the effects of constant overheads in a runtime. We observe that the self-relative speedups of DSTM and our approach are comparable.

The third set of measurements compared the amount of contention in the two systems as the number of threads increased. Here, we used the definition of conflicts given earlier, and plotted the change in the percentage of conflicts to the total number of initiated retriangulation tasks (cavities) as the number of threads increased.

3.5.2 Boruvka's algorithm for minimum spanning trees

Our experiments with the minimum spanning tree application used a graph dataset from the 9th Dimacs implementation challenge for shortest paths. The graph represents a road network of the Western USA United States and has 6,262,104 nodes and 15,248,146 edges.

The JCHORUS implementation used was as in Figure 3.9. The initial partition of the graph into divisions was random. The simple scheduler, with a breadth-first strategy, was used.

The fine-grained-locking implementation locked the graph at the level of components (recall that Boruvka's algorithm is based on repeatedly merging the components in a graph). If a component tried to acquire the lock for the component to merge and could not, then it placed the current component at the end of the worklist, released its lock, and proceeded to the next component. The DSTM2

version was built from this implementation.

In this case, the DSTM2 implementation aborted due to lack of memory. Figure 3.18 shows the speedup over the baseline sequential implementation as well as the self-relative speedup. Note the bizarre behavior of fine-grained locking at 8 threads. We suspect that this behavior is due to the garbage collector; however, as we do not have a performance model of this version of Java, we are unsure of its exact cause.

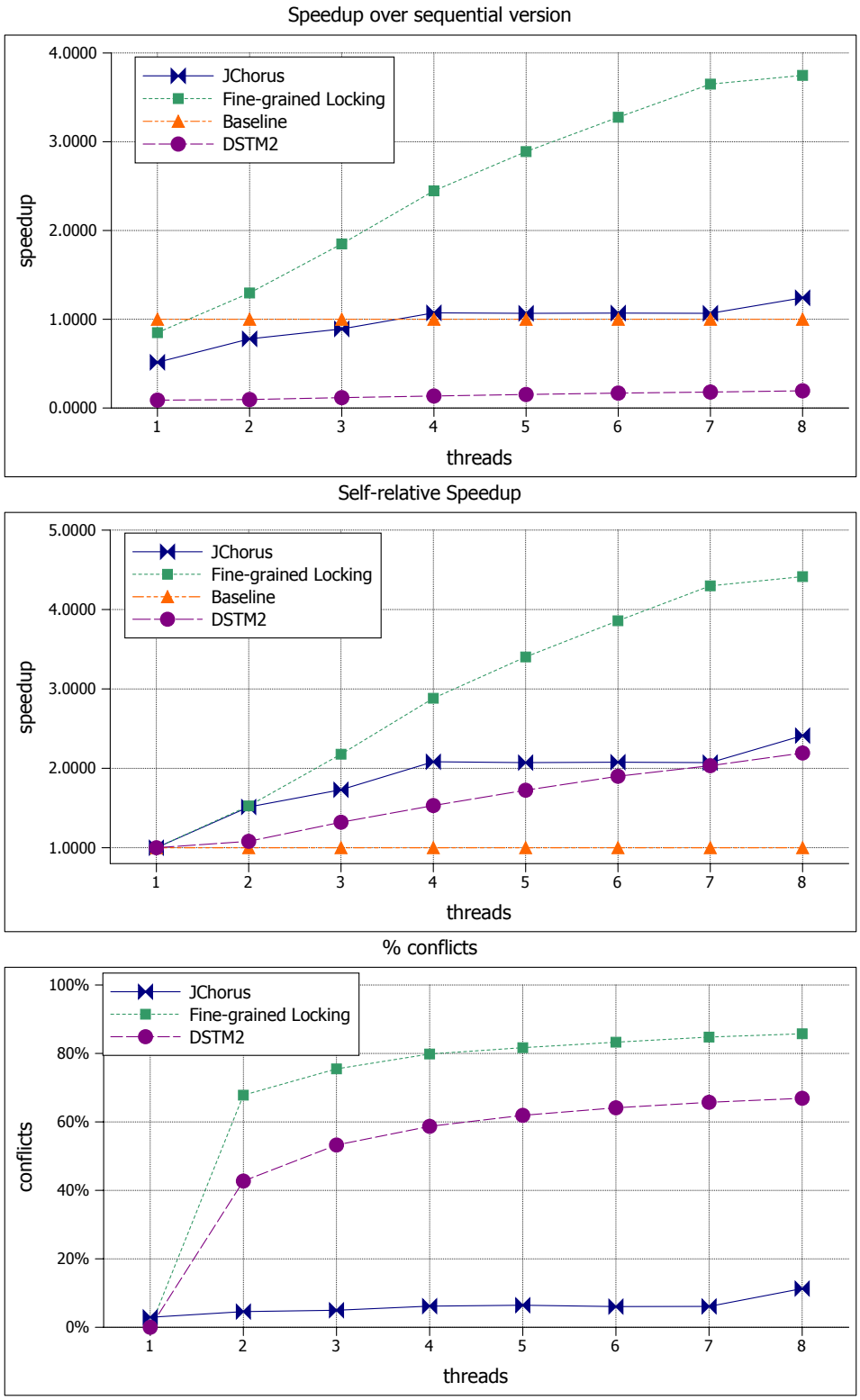


Figure 3.17. Delaunay Mesh Refinement benchmark results. Top: Speedup results over sequential baseline. Center: Speedup over 1-thread version. Bottom: Percentage of tasks that end up in conflict (for JCHORUS conflicts occur when incomplete cavities are discarded. For DSTM2 and Galois conflicts occur when tasks are aborted and rolled back.)

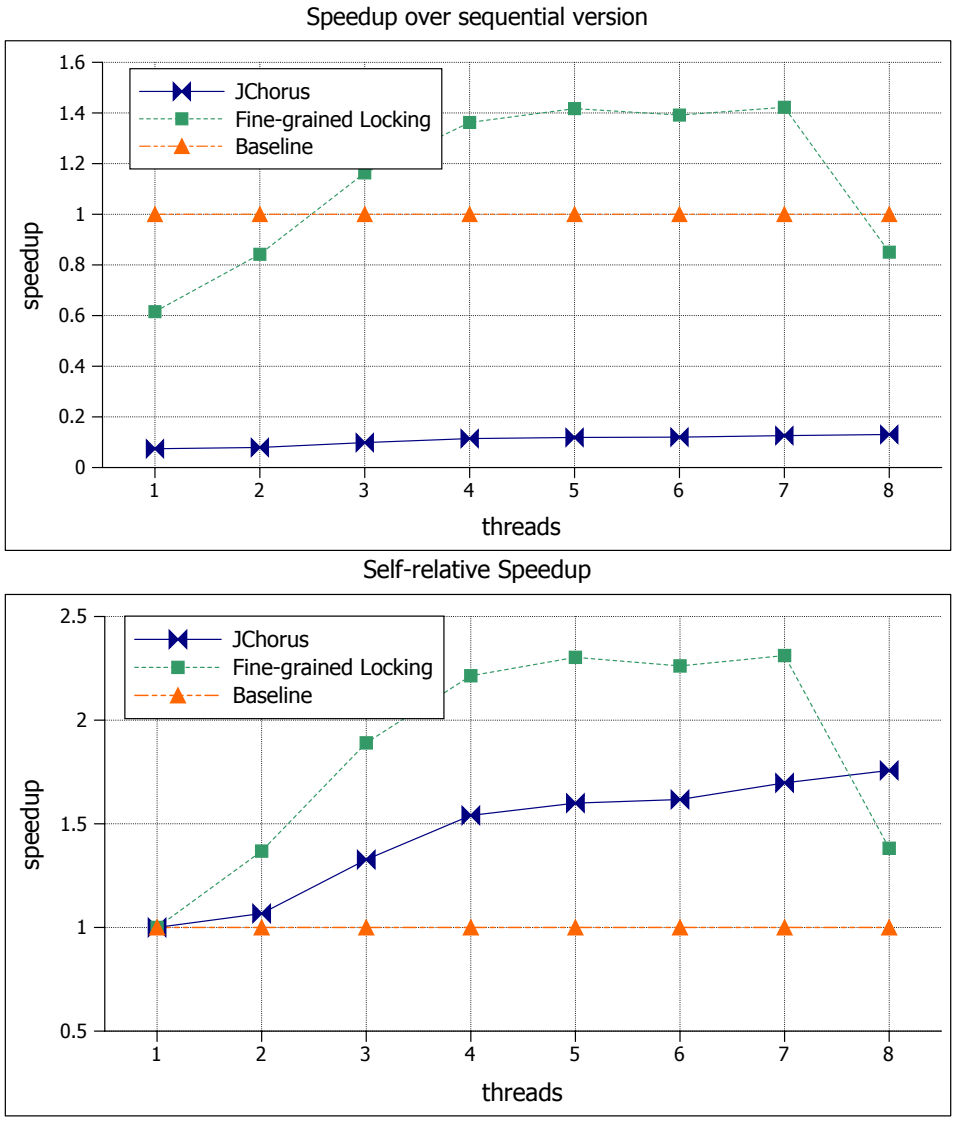


Figure 3.18. Boruvka's algorithm benchmark. Top: Speedup relative to sequential baseline. Bottom: Speedup over 1-threaded version

Chapter 4

The Habanero-Chorus Programming Language

In this chapter we present the second programming model built on top of Concurrent Assemblies, Habanero-Chorus. The main programming abstraction in Habanero-Chorus is an asynchronous isolated task, i.e. an asynchronous software transaction. In a sense programming in Habanero-Chorus is no different than programming Java with software transactions.

4.1 Main Abstractions in Habanero-Chorus

In Chapter 3 we showed that JChorus was a programming language suitable for cautious algorithms, i.e. algorithms where the main concurrent tasks access all necessary objects before performing any modification to the heap. However programming more general algorithms in JChorus might prove challenging. In this chapter we present a high level programming language Habanero-Chorus which is suitable for programming general algorithms.

The main abstraction in Habanero-Chorus is that of an *asynchronous isolated task*, i.e. tasks that may run concurrently but that operate in isolation. *Asynchronous isolated tasks* are similar to software transactions, where either all the effects of a task are visible to the others or none.

Consider, then, a scenario that is unavoidable in any dynamic approach to isolation: tasks A and B, executing isolated code and each owning parts of the

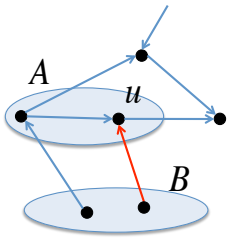


Figure 4.1. A conflict

heap, conflict over an object u in the shared heap. (See Fig. 4.1; here nodes represent shared objects, edges represent pointers between objects, the shaded ovals represent the regions of objects owned by the two tasks.) Clearly, only one of the two tasks—say A—should be allowed to access u. But what happens to B? Letting B block while maintaining ownership of some data does not seem to be a good idea: not only will tasks seeking B's data now have to wait, in the worst case, we will have deadlocks. Another option is to let the task manager retry B. If we retry B too soon, we will have unnecessary contention that will degrade performance; in the worst case, we will have livelocks. If we do not retry B soon enough, do we release all objects held by B? If so, what if a task C acquires some of this data before B is retried?

We note that the "right" strategy for concurrent execution here depends to a large extent on the structure of the data, and it is imperative in these settings that task and data management are somehow "coupled." Unfortunately, this is not the case in many traditional dynamic approaches to isolation. Rather than using an operational semantics that relates tasks with data in a principled manner, they often resolve conflicts using heuristics like exponential back-off [24] that do not offer solid guarantees, are divorced from high-level programming models, and depend on ad hoc parameters.

In contrast, the HABANERO-CHORUS execution model uses *Concurrent Assemblies* to implement isolated asynchronous tasks. Each such task is modeled as an assembly. If multiple tasks are trying to operate in the same region of the shared heap in an isolated manner, then concurrency among them is not serving any useful purpose, and we can do better if a single task takes over all the work. This task has explicit ownership of a local region in the shared heap, and can only access objects in that region. As in *Concurrent Assemblies*, such a task is called a

```
1: Mesh m = /* read input mesh */
2: List wl = new List(m.getBad());
3: foreach Triangle t in wl {
4:   Cavity c = new Cavity(t);
5:   c.expand();
6:   c.retriangulate();
7:   m.updateMesh(c);
8:   wl.add(c.getBad());  }
```

Figure 4.2. Delaunay mesh refinement (sequential)

(concurrent) assembly.

In case of conflicts among concurrently executing assemblies the merge operation over the conflicting assemblies occurs. In particular, consider the scenario in Figure 4.1: A and B are assemblies, and the shaded ovals depict their owned regions. Now suppose B attempts to access the object u via a pointer. In Habanero-Chorus, this conflict scenario is handled by letting B *delegate* itself and its owned region to A. In essence, A and B are now "merged," and from now on, A: (1) owns the region that B passed to A; and (2) has the responsibility of completing the task that B set out to do. We can see this as a form of *delegation*, where a task that detects a conflict *delegates* its work to the task owning the objects that caused the conflict.

At a high level of abstraction *merging as above is the only communication primitive in* Habanero-Chorus, as `merge` is the only communication primitive for *Concurrent Assemblies*. This delegation mechanism is transparent to the user and is accessed by the programmer using a simple programming model similar to that of software transactions. The core model contains only two parallel constructs: a two-word keyword "`async isolated`" that forks an asynchronous task that is to be executed in an isolated manner, and a keyword "`finish`" for synchronization. This Cilk [25]-style framework for scoped fork-join parallelism allows creation and joining of isolated tasks. While the "`finish`" construct (akin to "`sync`" in Cilk) is a form of task synchronization, there is no data communication involved in it and it does not introduce issues such as deadlocks or livelocks. Semantically, any two tasks that join at the same program point and operate on different parts of the shared data structure are causally independent, and can be executed in any order.

We observe that the above programming model is quite general and can be inte-

```
 1: main () {
 2:    for (t in initial set of bad triangles)
 3:         processTriangle(t); }

 4: processTriangle(t) {
 5:    async isolated {
 6:       if (t in mesh) {
 7:          Cavity c = new Cavity(t);
 8:          c.expand();
 9:          c.retriangulate();
10:          for (s in c.badTriangles())
11:               processTriangle(s);  }
       }
    }
```

Figure 4.3. Delaunay Refinement in the HABANERO-CHORUS programming model

grated with any imperative parallel language; in particular, we have implemented it on top of the HABANERO-JAVA language [26]. Note that the underlying programming language may have other primitives for task creation that offer no guarantees of isolation—for example, in HABANERO-JAVA, there is a keyword "`async`" for creating such tasks. Such tasks do not fall within the ambit of HABANERO-CHORUS.

Provided access to shared memory only happens from within blocks declared "`async isolated`," isolation and race-freedom are guaranteed in HABANERO-CHORUS. This is because the only objects accessed by an assembly A are those it owns. Deadlocks and livelocks are ruled out because when two assemblies conflict, one of them always delegates to the other, and the "merged" assembly continues to progress.

4.1.1 Extended example: Delaunay mesh refinement

To see how the HABANERO-CHORUS programming and execution model work, let us consider again the classic irregular application *Delaunay mesh refinement* [16, 27] discussed in Chapters 1 and 3. Given a set of points in the euclidean space, a Delaunay triangulation partitions their convex hull into a set of triangles such that no point lies in any triangle's circumcircle. As before [16], there are further qualitative constraints on the resulting triangles. To meet these constraints, a Delaunay triangulation often needs to be *refined*; triangles that do not satisfy this

property are "bad." The *mesh refinement* problem is to retriangulate the mesh so that there are no bad triangles left.

In a parallel version of the sequential algorithm presented in Section 3.1.1, the isolation requirement is that `c.retriangulate` executes atomically—logically, a cavity is a unit of isolation. In practice, cavities are almost always small, which means that a typical instance of the problem has a significant amount of available parallelism. At the same time, there is no a priori way here to tell if two parallel tasks carrying out retriangulations in parallel will *conflict*—i.e., if during cavity expansion, one task will demand access to a triangle under use by another. Conflict management, therefore, has to be dynamic.

In the HABANERO-CHORUS approach to the problem, the programmer codes the application as in Figure 4.3. The main function is a sequential loop that calls the method `processTriangle` on each bad triangle. While executing a call to the method `processTriangle` on input t, the HABANERO-CHORUS runtime notices a block of code `K` marked as "`async isolated`". Such a task is executed by the HABANERO-CHORUS runtime in an isolated manner—i.e., as an assembly. Therefore, in the current scenario, a new assembly B is created.

As in *Concurrent Assemblies* an assembly contains a region of the heap and a thread of control. The thread of control in HABANERO-CHORUS assemblies may contain code for the current `async isolated` tasks as well as for all the tasks that have been delegated to it. This code is represented by sequential composition.

The code `K` is executed following standard sequential semantics—only, any new triangle accessed by `K` is first added to its region by merging. Objects that are not owned by any `async isolated` tasks are represented in *Concurrent Assemblies* as belonging to "inert" assemblies (which have a fixed type ϵ). In the program on Figure 4.3, the first triangle added to the assembly region will be t. *This way, the assembly can only access objects in its owned region.* Unless there is a *conflict* (see below), B completes the execution of `K` and proceeds to execute the code for the delegated `async isolated` tasks if any. The challenging scenario is when `K` tries to access a triangle in the owned region of some other assembly A (or vice versa). In this *conflict* scenario, B *delegates* its work (all the code for itself and its delegated tasks. to A. Thus, delegation *coarsens* the granularity of parallelism in the heap, in a local and data-centric way.

Note that the code that caused a conflict must be executed *from the beginning* at a subsequent time. As our language is imperative, this code may, in general, already have performed modifications to the heap; therefore, HABANERO-CHORUS rolls back its effects by relying in the underlying mechanism of *Concurrent Assemblies*.

As mentioned in chapter 3 an important class of irregular applications are *cautious* algorithms. In principle, we could use a static analysis [28] that identifies an application or assembly as cautious—given such an analysis, we could eliminate the runtime overhead of maintaining rollback logs in many cases (including every assembly created in Delaunay refinement). The current implementation of HABANERO-CHORUS does not use any such optimization.

Also, the retriangulation triangle t may have been removed from the mesh by the time A gets to executing the delegated retriangulation (Line 6, Figure 4.3). In this case, A simply skips it, and moves on to the next closure in Q_A.

Finally, note that a block marked "`async isolated`" is in a sense an *asynchronously executed* transaction. The asynchrony entails that the recursive call inside the method `processTriangles` is not really a nested transaction; instead, it is a new asynchronously executed assembly.

4.2 Formal Semantics

Now we formalize a foundational subset of HABANERO-CHORUS, henceforth referred to as *Core* HABANERO-CHORUS.

Core HABANERO-CHORUS uses the same concurrency constructs as our implementation of HABANERO-CHORUS; however, the former makes several simplifying assumptions about the sequential language underlying the model. For example, statements in Core HABANERO-CHORUS do not call methods on objects, create new objects, or declare new local variables. Also, we let objects be untyped and assume that all objects are shared.

4.2.1 Syntax

HABANERO-CHORUS is implemented on top of a framework for fork-join paral-

lelism. In Core HABANERO-CHORUS as well as our implementation, the programmer creates assemblies by enclosing imperative code blocks within the construct "`async isolated {...}`". The construct "`finish {...}`" defines a scope at the end of which all assemblies created within the scope must join. Note that "`finish`"-blocks and "`async isolated`" blocks can be arbitrarily nested.

$$
\begin{array}{lcl}
\mathit{Prog} & ::= & \mathit{Prog};\ \mathit{Prog} \mid \texttt{finish}\,\{\mathit{Prog}\} \mid \\
 & & \texttt{async isolated}\,\{\mathit{Prog}\} \\
 & & \texttt{async isolated}\,\{\mathit{Stmt}\} \mid \texttt{finish}\,\{\mathit{Stmt}\} \\
\mathit{Stmt} & ::= & \texttt{v} := \texttt{u.f} \mid \texttt{v.f} := \texttt{u} \mid \mathit{Stmt};\ \mathit{Stmt}
\end{array}
$$

where $\texttt{v}, \texttt{u} \in \mathit{Var}$ and $\texttt{f} \in F$.

Figure 4.4. Syntax of HABANERO-CHORUS

Formally, let us assume a universe *Var* of *variables* and a universe F of *field names*; also, let us assume a standard syntax for side-effect-free arithmetic expressions *Exp* and boolean expressions *Bexp*. The syntax of programs *Prog* in Core HABANERO-CHORUS is shown in Fig. 4.4. We denote by $\mathit{Var}(P)$ the set of variables appearing in a program P—these are assumed to be implicitly declared at the beginning of P.

4.2.2 Semantics

The semantics of HABANERO-CHORUS will be given as a translation to *Concurrent Assemblies*. The main idea of the translation is to map `async isolated` tasks to *concurrent assemblies* of type τ and "free" objects as belonging to "inert" assemblies of type ϵ. Each access to a reference is performed by nondeterministically choosing to:

- `merge` along the edge with an assembly of type ϵ. This choice represent the "acquisition" of a "free" object.
- $\texttt{merge}_r$ along the edge with an assembly of type τ. This choice represents a "conflict", as the required object is owned by an active assembly. This operation will result in a rollback and delegation.
- an access to the specific object, `u.f =` $\cdots$. This choice represents a local operation.

- $\overline{\texttt{merge}}$ to accept incoming delegations. This choice represent a conflict, where the assembly that detected the conflict will merge and delegate with the assembly accepting the `merge`.

The isolation aspect of an execution of a Core HABANERO-CHORUS program is translated directly into the local update operations in the *Concurrent Assemblies* formal model semantics.

Let $[\![.]\!]$ by the translation function from HABANERO-CHORUS into *Concurrent Assemblies*. In the sequel we sketch the function $[\![.]\!]$:

$$
\begin{array}{lcl}
[\![\texttt{u.f = v}]\!] & \stackrel{\text{def}}{=} & \texttt{merge}([\![\texttt{u}]\!]_{exp}, \epsilon, [\![\texttt{u.f}]\!]_{exp} \texttt{ = } [\![\texttt{v}]\!]_{exp}\texttt{;}\cdots)\texttt{+} \\
 & & \texttt{merge}_r([\![\texttt{u}]\!]_{exp}, \tau)\texttt{+} \\
 & & [\![\texttt{u.f}]\!]_{exp} \texttt{ = } [\![\texttt{v}]\!]_{exp}\texttt{+} \\
 & & \overline{\texttt{merge}}([\![\texttt{u.f}]\!]_{exp} \texttt{ = } [\![\texttt{v}]\!]_{exp}\texttt{;}\cdots) \\
[\![Stmt_1\texttt{;}Stmt_2]\!] & \stackrel{\text{def}}{=} & [\![Stmt_1]\!]\texttt{;}[\![Stmt_2]\!]
\end{array}
$$

Figure 4.5. Sematics of Core HABANERO-CHORUS

At each reference access, for example `u.f`, one of three things can happen:

- either `u` is a reference to a local object, in which case the program proceeds to dereference `u` and continue or
- either `u` is not a reference to a local object, in which case the object may be owned by another `async isolated` task or the object is "free" or
- some other assembly merges with the current assembly in which case the combined assembly will continue to execute the code for the current assembly followed by the code in the merging assembly.

Note that in the above semantics, we assume that delegation happens atomically. At the level of the implementation, this atomic delegation step could have been built using straightforward, coarse-grained locking. This would make the atomicity of delegation obvious. However, to achieve greater performance, our implementation of delegation uses a more sophisticated fine-grained locking approach.

Section 4.3 describes this implementation and explains in detail how delegation is handled in all conflict scenarios. The key points are that all assemblies in the

system are ordered and can be locked, and locks on assemblies are obtained in order. Therefore, if assembly A tries to delegate to assembly B just as B is trying to delegate to A, both A and B will attempt to obtain locks on A and B in the same order, and only one will succeed.

4.2.3 Properties of Habanero-Chorus

Now we discuss the safety and liveness guarantees that Habanero-Chorus offers.

Isolation. The property of isolation demands that a concurrent task read or write shared-memory objects without interference from other tasks. In Habanero-Chorus, an assembly can only read or write objects in its own region; also, if A_1 delegates work to A_2, the ability of A_2 to read or write its objects is not compromised. Therefore, Core Habanero-Chorus guarantees isolation. In our implementation of the model, the guarantee of isolation holds for all memory accesses performed from within `async isolated` blocks; as in transactional memory systems, we do not offer guarantees on memory accesses made outside blocks declared to be isolated.

Deadlock-freedom. There are only two synchronization operations in Habanero-Chorus: scoped joining of assemblies and delegation. As the former operation does not depend on data at all, it clearly does not introduce deadlocks. As for delegation, the only plausible deadlock scenario involving *two* assemblies is the following: "Assembly A tries to delegate to assembly B, B tries to delegate to A, and neither can progress." This scenario, however, is impossible in Habanero-Chorus. If A and B try to simultaneously delegate to each other, then one of the two requests (let us say the one from B) will be nondeterministically selected and honored. Let u be the object that A needed to access in the statement that triggered the delegation request from A to B. After B delegates to A, A will have all objects that B previously owned. Therefore, the request from A will no longer be a conflict—the request from A to access u will succeed, and A will be able to progress.

It is easy to generalize the above argument to cyclic deadlocks between n assemblies. Of course, while our semantics for Core Habanero-Chorus is deadlock-free, we have to ensure that our *implementation* of Habanero-Chorus semantics

is deadlock-free as well. We do so in Sec. 4.3.

Livelock-freedom. In HABANERO-CHORUS two assemblies A_1 and A_2 would be livelocked if they constantly try to delegate to each other, none of them progressing. However, in such a scenario, HABANERO-CHORUS would destroy one of the two assemblies, delegating its work to the other—the other assembly would then be able to progress. This is in contrast to transactional memory systems where there is always a non-zero probability of a livelock.

Bound on conflicts/commit ratio Finally, an extremely interesting property of Core HABANERO-CHORUS is that in any execution, the number of conflicts is bounded by the number of commits. This property works as a sort of performance guarantee in high contention-scenarios, where, in many state-of-the-art systems, there may be too many aborted tasks and too few commits. As we show in Section 4.4, this property lets us achieve better performance in high-contention scenarios than other approaches to irregular parallelism, such as GALOIS and software transactional memory (with DSTM2 as an example).

4.3 HABANERO-CHORUS Implementation Details

A key aspect of our model, that presents challenges not present in other concurrency models, is the fact that ownership changes operate on sets of objects rather than on the individual objects themselves. This is illustrated in a conflict scenario, where an assembly will delegate its code as well as transfer its region of ownership to the assembly it has a conflict with. We use a parallel version of a union-find data structure for disjoint sets [21, 29] to implement this ownership transfer efficiently.

Figure 4.6 summarizes the main abstractions in the HABANERO-CHORUS runtime: shared objects, assemblies, and isolated code.

Here, the "`Assembly`" objects represent a collection of "`IsolatedCode`" blocks that represent all the tasks for whose execution the assembly is responsible. When created using the HABANERO-CHORUS "`async isolated`" construct, the "`Assembly`" object will contain an empty region and a single task in the queue. As the program progresses, assemblies either accumulate data and tasks, finish their execution and terminate, or delegate their data and tasks to another assembly and terminate.

```
class hj.lang.Object {
    // owner is only changed using CAS
    private Assembly owner;

    // unowned objects are acquired
    // using this method
    public boolean acquireObject(
                        Assembly newOwner);

    // called on the first access to this
    //object by the current owner
    public void acquire();

    // get the owner by navigating
    // the union−find data structure
    public Assembly owner();

    // methods for rollback and restore
    public Copy copyValues();
    public void restoreValues(Copy v);
}


class hj.runtime.wsh.IsolatedCode {
    // points to an Assembly in the
    // union find structure
    private Assembly owner;

    // get the owner by navigating
    // the union find data structure
    public Assembly owner();

    // holds information on how to
    // restore objects on roll back
    private List<Copy> rollbackLog;

    // actual code from the user program
    // is run through this method
    public void run();

    // the commit action, done
    // in constant time
    public void commit();

    // restore all accessed object to
    // their pre−execution state
    public void rollback();
}
```

```
class hj.runtime.wsh.Assembly {
    // parent pointer in the union−find tree
    private Assembly parent;

    // for union by rank
    long rank;

    // a unique assembly id
    private long id;

    // a queue of delegated isolated code
    private Queue<IsolatedCode>
                delegatedCodeQueue;

    // find the representative assembly
    public Assembly find();

    // An assembly that finished the execution
    // of all its tasks. Free objects are represented
    // by belonging to a dead assembly
    public boolean dead();

    // implements the delegation through union
    public void delegateTo(Assembly to);

    // next IsolatedCode to run
    public IsolatedCode next();

    // acquire assemblies a1 and a2 in order
    static Locks orderedLock(Assembly a1,
                             Assembly a2);
}
```

Figure 4.6. Principal abstractions in the HABANERO-CHORUS implementation

```
1  Method Acquire, Input   : Object obj
2    objAssembly = obj.owner();
3    currAssembly = this.owner();
4    if objAssembly == currAssembly then go to 13;
5    if  acquireObject(obj, currAssembly) then go to 13
     // Conflict!  Delegate to other assembly
6    locks = orderedLock (currAssembly, objAssembly);
     // If anything changed, retry
7    if (objAssembly ≠ obj.owner() ∨ objAssembly.dead() ∨ currAssembly ≠
     this.owner()) then
8      locks.release(); go to 2;
9    rollback (); // rollback the assembly's state
10   delegateTo (objAssembly);
11   locks.release();
12   die ();
13   if (write operation ∧ obj ∉ writeSet) then
14     rollbackLog.add(obj.copyValues());
15     writeSet = writeSet ∪ {obj}
```

Figure 4.7. Algorithm for acquiring an object

During the execution of an "`async isolated`" block, the runtime needs to perform ownership checks when accessing shared objects. These checks are performed by calling "`Object.acquire()`" whose pseudocode is shown in Figure 4.7. Note that a given "`async isolated`" block needs to acquire a particular shared object only once, the first time the object is accessed.

The algorithm for "`Object.acquire`" ensures that the check for object ownership and any delegation is done *atomically* with respect to the other assemblies in the system. Lines 6 and 7, discussed below, are critical for guaranteeing this atomicity.

"`Object.acquire`" first checks if the assembly that is executing the code already owns the object, in which case it can continue execution (adding the object to its write set if necessary) on line 13. The assembly then tries to acquire the object (line 5), which will succeed if the object is not currently owned by any other assembly. If successful, the assembly can continue (line 13).

Otherwise, there is a conflict and the assembly needs to delegate itself to the one that holds the ownership of the object. It first acquires the locks on the

current assembly and on the assembly that owns the desired object (line 6). To avoid deadlock, all assemblies that are created in the program have a unique ID and the locks are always acquired in order.

Upon acquiring the locks, the assembly checks on line 7 if anything has changed in the meantime. The assembly that owned the object might have delegated its tasks to another assembly and transferred the ownership of its objects, the assembly that owned the object might have finished the execution of all its activities and died (transferring the ownership of its objects to the unowned object pool) or the representative for the current assembly might also have changed (a consequence of a requirement for the performance guarantees in the disjoint set data structure, delegations can result in the representative for the current assembly to change). If any of that has happened both assemblies are released and the process of acquiring the object is retried.

If nothing has changed, the assembly proceeds with the delegation process. It first rolls back all the changes it has made to the objects in its *writeSet*, then performs the delegation, which copies all the `IsolatedCode` tasks from the currently assembly's queue to the queue of the object's owner. This is done in constant time by concatenating the delegation queues together. The delegation process also transfers the ownership of all the objects owned by the current assembly to the assembly it is delegation to. The assembly then releases the locks and dies (returns the execution to the runtime).

As already mentioned, if all the accesses to the shared memory in the program are done only inside the isolated blocks, our implementation of the isolated construct using assembly delegation as described in the algorithm in Figure 4.7 guarantees isolation and race freedom.

We use a union-find data structure to track object ownerships. This allows us to perform near-constant time object ownership checks, and constant time ownership transfers when an assembly delegates itself to another one, as well as constant time releases of all the objects owned by the assembly when the assembly finishes its execution.

Figure 4.8 show the union-find data structure we use to track the ownership of objects. Each object has an *owner* reference, which points to the assembly that owns it. Assemblies merged by delegation are represented as a tree with the root

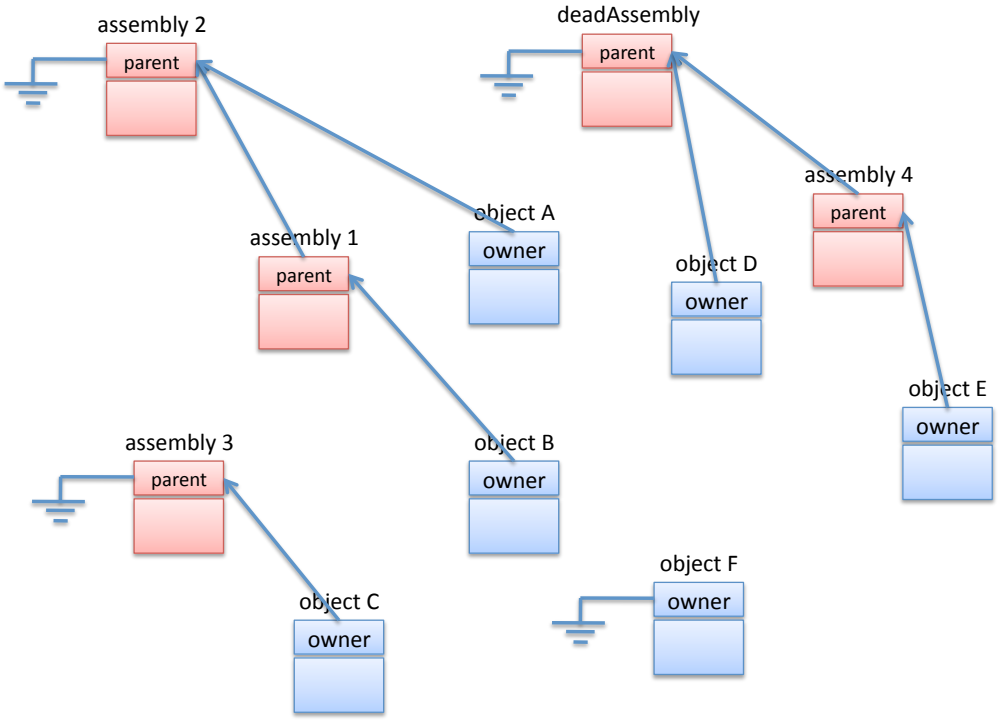


Figure 4.8. Union find data structure to track object ownerships

being the representative assembly. Assemblies also have a *parent* reference that points to their parent in this tree. On Figure 4.8, object C is owned by assembly 3, while objects A and B are owned by assembly 2. When an assembly delegates to another assembly, it transfers the ownership of its objects, which will result in setting either its *parent* reference to point to the assembly it delegated to or vice-versa, depending on the shapes of their union-find trees.

To efficiently keep track of all the objects that have been released, assemblies that have finished execution are marked as dead. An object is free if its *owner* reference is *NULL* (which will be the case if it has never been acquired by anyone),

or if the *find* algorithm determines that its owner is dead (which will be the case if it has been acquired by some assembly and then the assembly or its delegate has finished execution). In Figure 4.8, objects F, D and E are not owned by anyone.

Assemblies that are not at the root of the union-find tree are eventually garbage collected when no objects point to them. References to these assemblies are removed by $find$ since it is performed with path compression. Assembly 1 in Figure 4.8 will become unreferenced when, for example, a *find* is performed on object B.

4.4 Experimental Evaluation

We have used two platforms for our experimental evaluation: (a) a 64-way (single-socket, 8 cores, 8 threads per core) 1.2 GHz UltraSPARC T2 (Niagara 2) with 32 GB main memory running Solaris 10 and Sun JDK 1.6 64-bit version; (b) 16-way (quad-socket, quad-core per socket) Intel Xeon 2.4GHz system with 30GB of memory and running Red Hat Linux (RHEL 5) and Sun JDK 1.6 64-bit version. All benchmark were allocated 3.5 gigabytes in the JVM heap with the exception of DSTM2 that needed a 15G heap to complete. Each benchmark was run 7 times in each platform/programming model. The result reported are the average excluding the first run to factor out just-in-time compilation effects.

4.4.1 Benchmarks

We evaluate our approach on a set of four different benchmarks described below. The performance on each benchmark is measured only for the parallel stage of the algorithms .

4.4.1.1 Boruvka minimum spanning tree

We consider a classical algorithm for computing the minimum spanning tree of an undirected graph [14]. The main idea is to proceed by fusing nodes through their minimal weight edge, maintaining for each fused set of nodes the minimal spanning tree. As the algorithm progresses it builds larger and larger local minimum spanning trees until only one node remains. Initially a parallel task is spawned for

```
 1:  start(nodes){
 2:    finish {
 3:      for(n in nodes)
 4:        mergeMinEdge(n);
 5:    }
 6:  }

 7:  mergeMinEdge(c) {
 8:    async isolated {
 9:      c' = c.getMinEdge();

10:      // merge c into c'
11:      c'.addEdges(c);

12:      // change x -> c into x -> c'
13:      // change c -> x into c'-> x
14:      // remove c -> c' and c -> c'
15:      c'.fixEdges(c);
16:    }
17:  }
```

Figure 4.9. Boruvka Minimum Spanning Tree pseudocode

```
 1:  start(nodes){
 2:    finish {
 3:      for 0...40000
 4:         task()
 5:    }
 6:  }

 7:  task() {
 8:    async isolated {
 9:      for 0..20 //  20 operations per task

10:        v = *;
11:        do *  => insert(v)   //  5%
12:           *  => delete(v)   //  5%
13:           *  => contains(v) // 90%
16:    }
17:  }
```

Figure 4.10. HashTable and RBTree pseudocode

each node in the graph. As they grow their local minimal spanning trees, the tasks need to merge their local tree with the tree of their minimal weight edge neighbor. This algorithm starts with the maximum parallelism available and the parallelism decreases as the tree is built. For evaluation we used a graph representing the road network for the state of New York from [30]. The road network graph consists of 264,346 nodes and 733,846 edges. A pseudocode for the algorithm is shown in figure 4.9. This benchmark is part of the Lonestar Benchmark Suite [19].

4.4.1.2 Delaunay mesh refinement

As mentioned earlier, this algorithm refines a 2D Delaunay mesh so that each of the triangles satisfy a quality constraint. In our evaluation, we start with a triangle mesh consisting of 100,770 triangles of which 47,768 do not satisfy the quality constraint. This benchmark and the dataset used are part of the Lonestar Benchmark Suite [19].

4.4.1.3 Hash table and Red-black tree

Whereas all other benchmarks are parallel implementations of some algorithm to exploit the inherent data parallelism, these two benchmarks illustrate a scenario where several clients access shared data structures concurrently. In these benchmarks we model 40,000 concurrent tasks performing 20 operations each in an isolated manner. Operations are chosen at random with a 90% chance of an operation being a query, 5% being an insert and 5% being an update. The values are also chosen at random. In the RBTree benchmark operations are performed on a red-black tree while in HashTable they are performed on a 256-bucket hash table. A pseudocode for the algorithms is shown in figure 4.10. These benchmarks are based on the red-black tree and hashtable benchmark implementations in DSTM2 [22].

4.4.2 Methodology

We compare our approach with several alternate parallel programming models :

- **JChorus**: This programming model described in detail in chapter 3

- **DSTM2:** DSTM2 [22] provides an implementation of dynamic software transactional memory. The red-black tree and hashtable benchmarks are provided in the DSTM2 distribution. .

- **HJ (SEQ)**: Habanero-Java [26, 31] sequential. The sequential baseline versions of all benchmarks are implemented using the Habanero-Chorus data structures and runtime and a sequential version of the code.

- **HJ (ISO):** Habanero-Java with isolation provided by the Habanero-Java "`isolated`" construct. This multithreaded version of the benchmarks were implemented using the same code and data structures as in Habanero-Chorus. The `isolated` construct in Habanero-Java is implemented as a coarse-grained lock.

- **Java (FGL)** The fine-grained locking implementation of the Delaunay and Boruvka benchmarks. Several worker threads obtain work units from a global first-in-first-out worklist, they attempt to acquire all needed shared objects and if failed the work unit is returned to the worklist.

- **Galois**: In this programming model [32], algorithms are expressed as sequential operations on an unordered worklist. The runtime executes the loop bodies in parallel and through annotations the programmer can specify locking and logging policies as well as many policy options for the implementation of the abstract worklist. The Delaunay mesh refinement and Boruvka minimum spanning tree benchmarks are included in the Galois distribution. The red-black tree and hashtable benchmarks where implemented by us for this paper using the graph data structures from the Galois distribution.

4.4.3 Experimental Results

A comprehensive collection of all timing results is shown on Table 4.2. We have also extracted some of the most interesting results and plotted them on graphs discussed below.

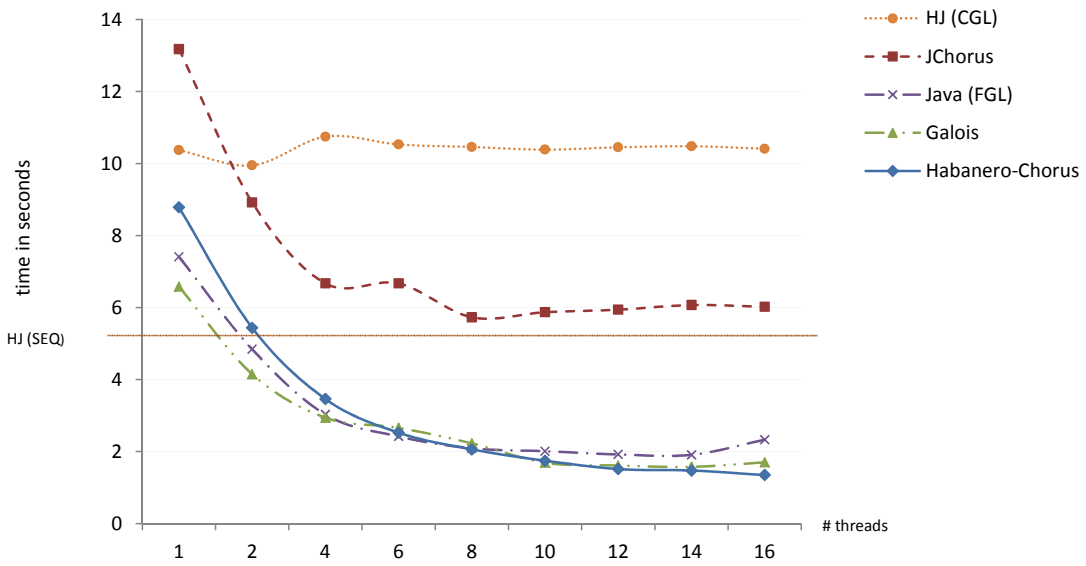


Figure 4.11. Delaunay mesh refinement on a 16-core Xeon (Note: For reasons of scale DSTM2 is left out of the plot. DSTM2 -1 thread: 962s, 16 threads: 177s)

4.4.3.1 Self relative speedups

Figures 4.15 and 4.16 show the speed up with respect to the 1 threaded execution for Delaunay mesh refinement and Boruvka minimum spanning tree respectively. Self relative speedup measures the scalability of the different approaches factoring out their respective overheads. As shows in the figures HABANERO-CHORUS show the most consistent speedup compared to all other approaches.

4.4.3.2 Execution times

Figures 4.11 and 4.12 show the execution times for Delaunay mesh refinement on the 16-core Xeon and the 64-way UltraSPARC T2 respectively. HABANERO-CHORUS shows a speedup of 5.4× over sequential HJ at 16 threads compared to the next best approach which is Galois at 3.3× over sequential HJ at 14 threads

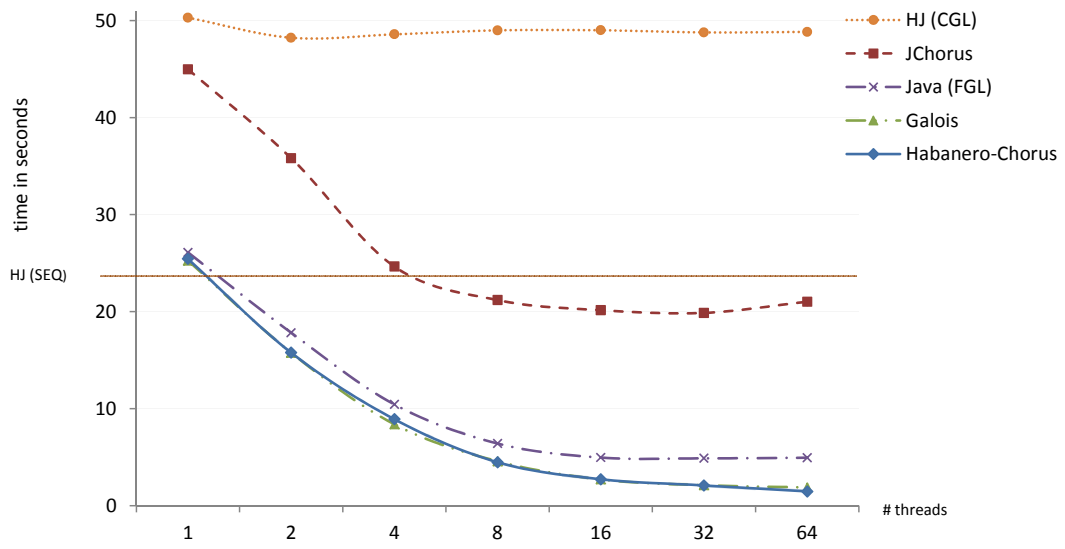


Figure 4.12. Delaunay mesh refinement on a 64-way UltraSPARC T2 (Note: For reasons of scale DSTM2 is left out of the plot . DSTM2 -1 thread: 795s, 64 threads: 81s)

(where Galois has a maximum speedup) on the Xeon architecture. The self-relative speedups are 7.8× for HABANERO-CHORUS and 4.1× for Galois. On the UltraSPARC T2 architecture, HABANERO-CHORUS peaks at 11.9× speedup against sequential HJ at 32 threads whereas Galois again is the next best and peaks at 11.5× with 64 threads.

Figures 4.13 and 4.14 show the performance numbers for Boruvka minimum spanning tree on the on the 16-core Xeon and the 64-way UltraSPARC T2 respectively. HABANERO-CHORUS shows a speedup of 1.16× over sequential HJ at 16 threads being the third best approach bettered by Java fine-grained locking at 1.37× (at 12 threads) and also by Galois at 1.36× (at 14 threads). The self-relative speedups are 3.6× for HABANERO-CHORUS, 2.4× for Java FGL and 2.3× for Galois. On the UltraSPARC T2 architecture, on the other hand, HABANERO-CHORUS peaks at 2× speedup against sequential HJ at 16 threads whereas Galois

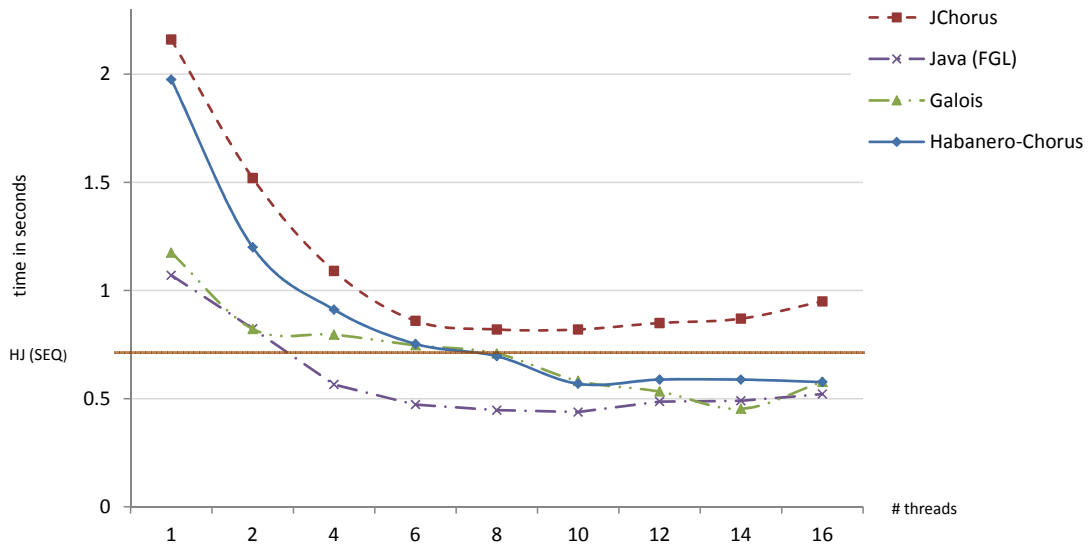


Figure 4.13. Boruvka minimum spanning tree on a 16-core Xeon (Note: For reasons of scale HJ (ISO) and DSTM2 are left out of the plot HJ (ISO)- 1 thread: 5.5s, 16 threads: 10.3s, DSTM2- 1 thread: 4.8 , 16-threads:2.7.s)

is at 1.7× at 64 threads and both are bettered by Java FGL at 3.3× at 16 threads. The self-relative speedups are 8.8× for HABANERO-CHORUS, 7.1× for Galois and 4.23× for Java FGL.

The speedups and performance numbers indicate that HABANERO-CHORUS is a very competitive approach for parallelizing irregular applications, especially when compared with other JVM based systems. The reduced speedup of HABANERO-CHORUS and Galois over Java fine-grained locking on Boruvka tree is due to the overhead involved in keeping track of ownerships, conflicts and the maintenance of rollback logs, compared to a much lighter-weight implementation in Java using fine-grained locks. In Boruvka, the transactions perform very little work, so the relative impact of the bookkeeping overhead is very high, as evidenced by the difference between self-relative and sequential-relative speedups.

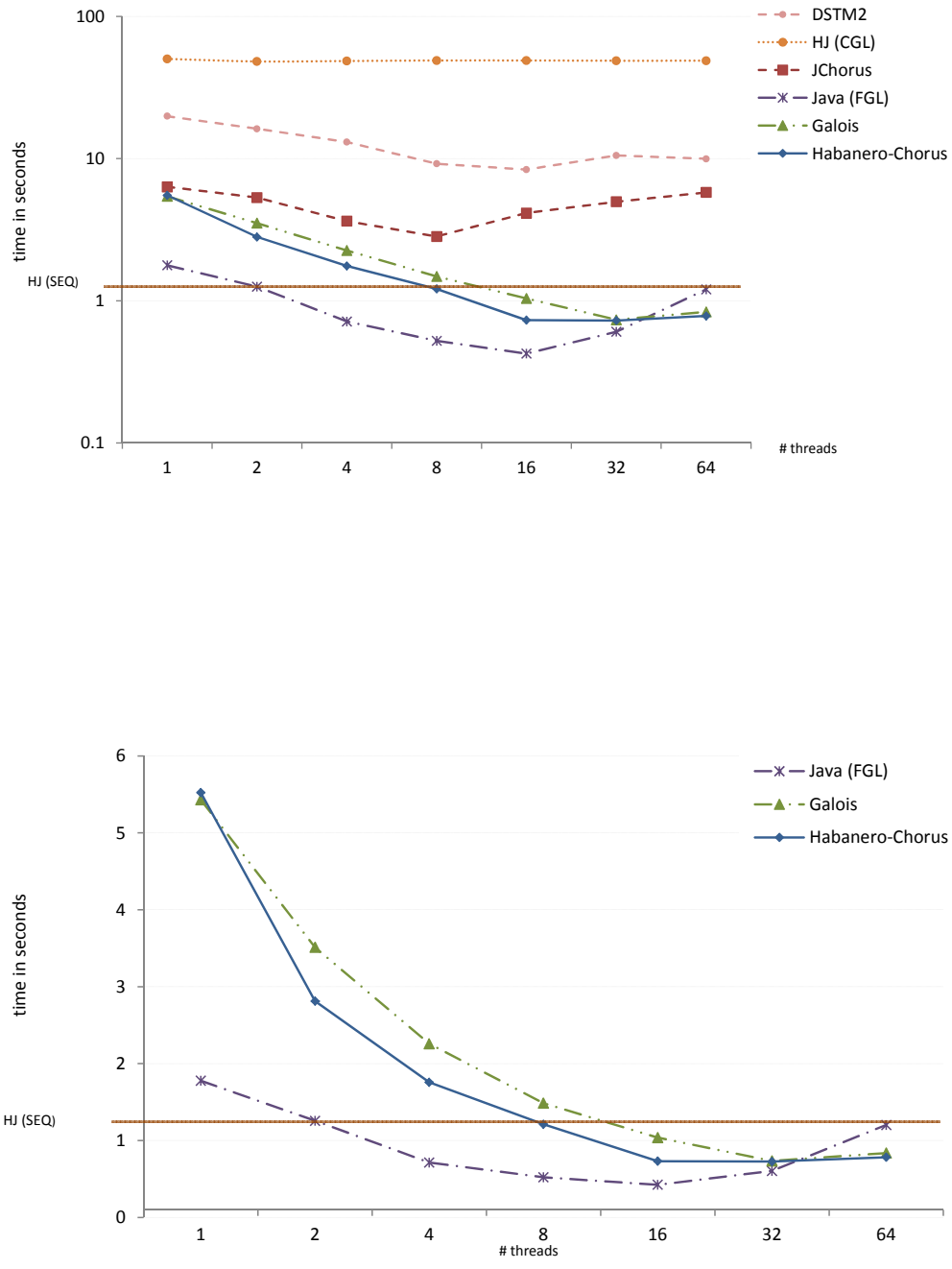


Figure 4.14. Boruvka minimum spanning tree on a 64-way UltraSPARC T2 (left: logarithmic scale; right: inset of the bottom 6 seconds)

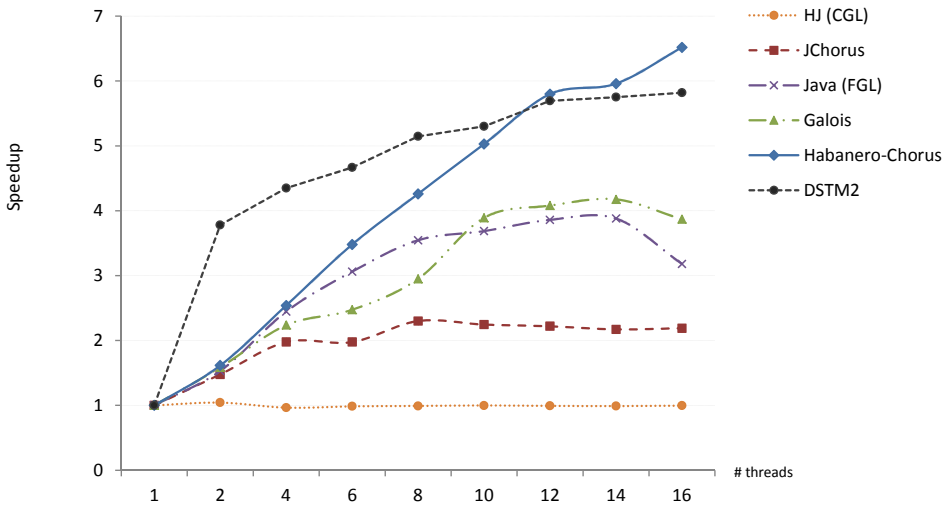


Figure 4.15. Delaunay self relative speedup

HABANERO-CHORUS is one to two orders of magnitude faster than both Galois and DSTM2 on HashTable and RBTree benchmarks. Since the Galois implementation of these two benchmarks was straightforward and was done using minimal programming effort (about the same effort that is required to write these benchmarks in DSTM2 or HABANERO-CHORUS), we suspect that a more careful implementation using more annotations and done by a Galois expert might yield better results than what we have obtained in Table 4.2. Either way, we feel that these two benchmarks are much better suited to illustrate the effects of speculation discussed below, than for raw speed comparisons.

4.4.3.3 Aborted speculative work

DSTM2, Galois and HABANERO-CHORUS, like many approaches to parallelization and concurrency, rely on speculative execution to improve performance. However, in scenarios with relatively high contention, speculative execution might hinder performance and resource usage. Figure 4.17 compares the impact of speculative

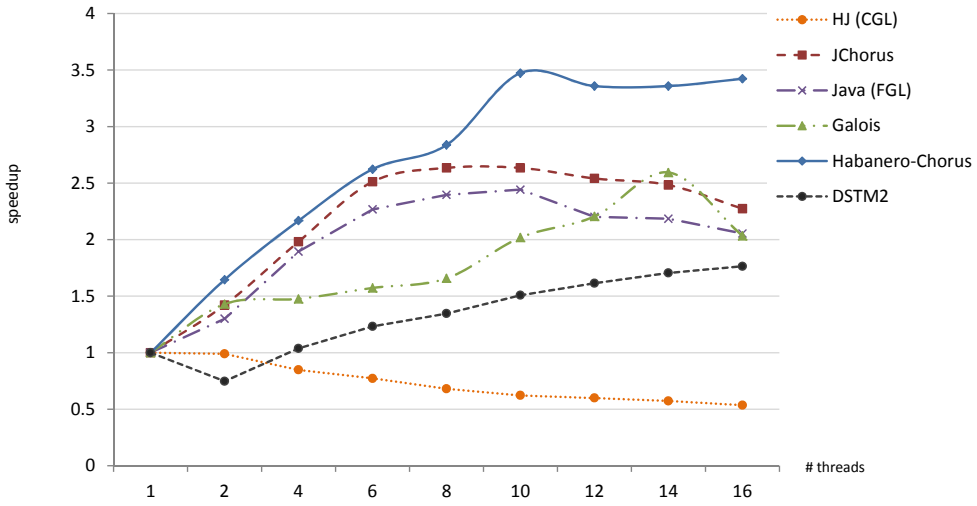
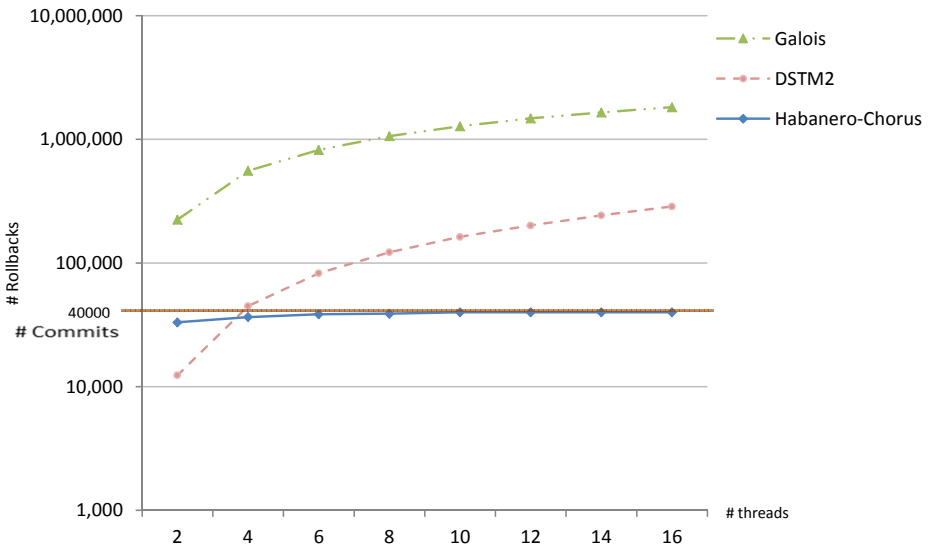


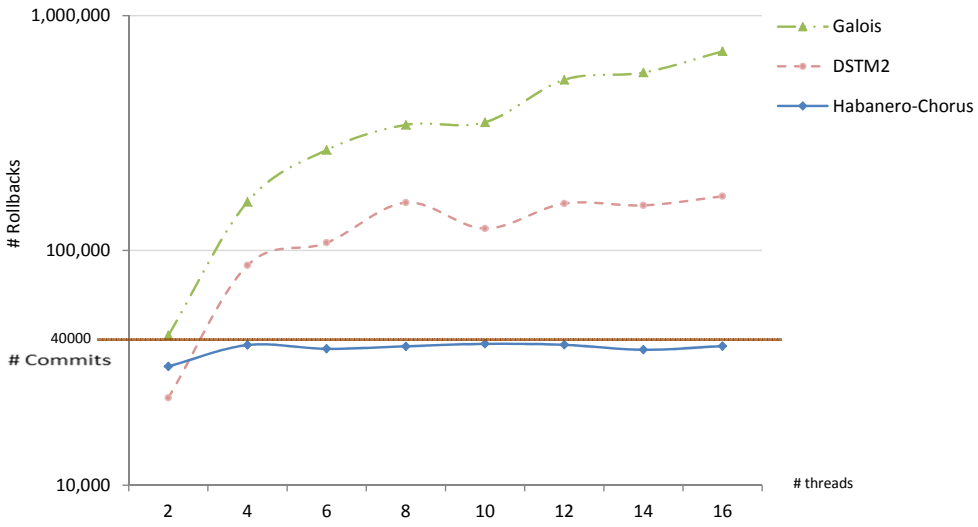
Figure 4.16. Boruvka self relative speedup

execution for HABANERO-CHORUS Galois and DSTM2 in terms of the numbers of commits and aborts for a given workload. HABANERO-CHORUS offers a lower bound on the number of aborted speculative tasks compared to most transactional memory implementations. As discussed in Section 4.2.3, HABANERO-CHORUS guarantees that there will be only as many aborted speculative tasks as there are successful ones, whereas DSTM2 only bounds the number of aborted tasks in relation to the total number of shared objects. Galois provides a probabilistic progress guarantee based on exponential back-off for conflict resolution. Figures 4.17 (a) and (b) show DSTM2 having an order of magnitude and Galois having two orders of magnitude more aborts than HABANERO-CHORUS.

We have shown the HashTable and RBTree benchmark results with a 90% read-only and 10% write operation mix, which is generally considered to be a realistic medium-to-high contention scenario [24, 33]. We also tested even higher contention scenarios, where as expected HABANERO-CHORUS fared even better compared to Galois and DSTM2.



(a) RBTree



(b) HashTable

Figure 4.17. Impact of speculative execution

Benchmark	Model	1	2	4	8	16
RBTree	Habanero-Chorus	0.62	0.55	0.52	0.57	0.63
	Galois	21.07	19.50	19.68	22.24	22.87
	DSTM2	41.89	49.29	50.54	53.84	62.458
HashTable	Habanero-Chorus	5.58	2.19	1.83	1.90	1.98
	Galois	182.92	164.10	196.63	166.84	153.124
	DSTM2	414.54	266.84	290.72	268.23	306.64
Delaunay Mesh Refinement	HJ (SEQ)	5.27				
	Habanero-Chorus	7.65	4.03	2.98	1.70	0.97
	Galois	6.58	4.15	2.94	1.77	1.70
	Java (FGL)	7.55	5.33	3.05	2.13	2.21
	JChorus	13.37	9.02	6.73	5.88	6.11
	HJ (ISO)	10.37	9.02	10.74	10.46	10.41
	DSTM2	962.55	291.84	258.46	225.95	176.92
Boruvka Minimum Spanning Tree	HJ (SEQ)	0.64				
	Habanero-Chorus	2.00	1.20	1.02	0.69	0.55
	Galois	1.09	0.83	0.73	0.69	0.57
	Java (FGL)	1.11	0.86	0.58	0.47	0.55
	JChorus	2.22	1.56	1.12	0.89	0.95
	HJ (ISO)	5.45	5.56	6.49	8.08	10.27
	DSTM2	4.821	6.453	4.64	3.58	19.22

Table 4.1. Benchmark results a 16-core Xeon (time in seconds).

4.4.3.4 Overhead

Our performance numbers indicate that the sequential overhead of our Habanero-Chorus implementation is similar to that of Galois. We note that as the work performed in isolation increases, the relative overhead decreases, evidenced by the different single thread slowdowns with respect to a sequential code.

4.4.3.5 Programmability

In this section we compare Habanero-Chorus, to the most successful approach to our knowledge to parallelize irregular applications. Kulkarni, Pingali et al's Galois project[27, 34] aims to enhance thread-based, optimistic parallelism with data-types for unordered and partially ordered sets, which give the runtime information about data dependencies [27], and heuristics for data partitioning [34]. The **Galois** programming system is a library on top of the Java programming language. Being a library based approach its programmability is hampered by

Benchmark	Model	1	2	4	8	16	32	64
RBTree	HABANERO-CHORUS	3.43	3.21	4.16	4.26	4.36	4.42	4.56
	Galois	106.64	109.53	110.01	123.28	110.13	124.17	123.52
	DSTM2	203.25	197.46	206.10	188.39	201.17	228.33	260.35
HashTable	HABANERO-CHORUS	92.10	65.39	90.33	89.97	89.68	90.44	97.03
	Galois	649.00	539.74	516.02	556.72	600.46	640.21	705.85
	DSTM2	1636.41	1120.30	901.04	886.22	899.95	914.31	903.24
Delaunay Mesh Refinement	HJ (SEQ)	23.87						
	HABANERO-CHORUS	25.43	18.67	9.54	5.47	3.08	2.01	2.32
	Galois	23.39	15.59	8.08	4.67	3.02	2.48	2.07
	Java (FGL)	24.72	17.32	10.15	6.21	4.72	4.33	4.98
	JCHORUS	33.35	32.94	24.25	19.44	17.85	17.33	18.6
	HJ (ISO)	50.30	48.23	48.58	48.99	49.00	48.78	48.83
	DSTM2	795.34	423.83	282.02	187.95	133.12	101.32	81.01
Boruvka Minimum Spanning Tree	HJ (SEQ)	1.26						
	HABANERO-CHORUS	5.52	2.81	1.49	0.88	0.63	0.63	0.73
	Galois	5.33	3.37	2.06	1.31	0.93	0.79	0.75
	Java (FGL)	1.61	1.20	0.69	0.45	0.38	0.55	0.91
	JCHORUS	14.25	11.36	9.86	9.4	10.7	11.75	13.62
	HJ (ISO)	37.74	37.30	37.75	38.14	41.19	46.32	50.25
	DSTM2	19.22	18.66	12.72	8.66	7.76	10.97	10.77

Table 4.2. Benchmark results ona 64-way UltraSPARC (time in seconds).

```
 1: void doCavity(Triangle start) {
 2:    async isolated {
 3:     if (start.isActive()) {
 4:       Cavity c = new Cavity(start);
 5:       c.initialize(start);
 6:       c.retriangulate();

          // launch retriagnulation on new bad triangles.
 7:       Iterator bad = c.getBad().iterator();
 8:       while (bad.hasNext()) {
 9:         final Triangle b = (Triangle)bad.next();
10:         doCavity(b);
          }

          // if original bad triangle was NOT retriangulated,
          // launch its retriangulation again
11:       if (start.isActive())
12:         doCavity(start);
        }
      } // end isolated
    }
```

Figure 4.18. Actual code for Delaunay Refinement in the HABANERO-CHORUS programming model

the heavy syntax of function calls and anonymous classes as we will show in the sequel. Even factoring out the aforementioned aspects, the **Galois** project also requires a heavy use of programmer annotations in order to achieve performing implementations.

```
    // The parallel loop
 1: GaloisRuntime.foreach(badNodes,
 2:   new Lambda2Void<... >() {
 3:   public void call(GNode<Element> item,
 4:     ForeachContext<GNode<Element>> ctx) {

 5:      if (!mesh.contains(item, MethodFlag.CHECK_CONFLICT))
 6:         WorkNotUsefulException.throwException();

 7:      Cavity cavity = new Cavity(mesh);
 8:      cavity.initialize(item);
 9:      cavity.build();
10:      cavity.update();

        //remove the old data
11:      List<...> preNodes = cavity.getPre().getNodes();
12:      for (int i = 0; i < preNodes.size(); i++)
13:        mesh.remove(preNodes.get(i), MethodFlag.NONE);

        //add new data
14:     Subgraph postSubgraph = cavity.getPost();
15:     List<...> postNodes = postSubgraph.getNodes();
16:     for (int i = 0; i < postNodes.size(); i++) {
17:        GNode<Element> node = postNodes.get(i);
18:        mesh.add(node,  MethodFlag.NONE);
19:        Element element = node.getData( MethodFlag.NONE);
20:        if (element.isBad())
21:           ctx.add(node,  MethodFlag.NONE);
        }
24:     List<...> postEdges = postSubgraph.getEdges();
25:     for (int i = 0; i < postEdges.size(); i++) {
26:       ObjectUndirectedEdge<...> edge = postEdges.get(i);
27:       mesh.addEdge(edge.getSrc(), edge.getDst(),
28:                    edge.getData(),  MethodFlag.NONE);
        }
29:     if (mesh.contains(item, MethodFlag.NONE )) {
30:        ctx.add(item,  MethodFlag.NONE);
        }
      }
31: }, Priority. first(ChunkedFIFO.class)
                .thenLocally(LIFO.class)) ;
```

Figure 4.19. Delaunay Refinement in the Galois programming model

Figures 4.18 and 4.19 contrast the code for Delaunay Mesh Refinement in HABANERO-CHORUS and Galois respectively to show the differences in programmability between these two approaches. Galois uses three different types of annotations: annotations to denote the scope of the parallel loop, performance oriented hints and annotations regarding task scheduling policies. Lines 1-4 in In the aforementioned example define the scope of the parallel loop.

Galois provides three performance oriented hints to annotate each access to a shared data structure: `NONE`, `CHECK_CONFLICT` and `SAVE_UNDO`. By default in each access to the shared data structure the runtime checks whether the nodes are being in use by a concurrent tasks, and saves state information in a log to provide for an eventual undo. The `NONE` hint instructs the runtime not to perform any check nor save undo data. This flag reduces most of the overhead and is used in lines 13, 18, 19, 28 and 30. The `CHECK_CONFLICT` hint instructs the runtime not to save undo data and is used on line 5. Finally the annotation at line 31-32 instructs the runtime to group activities into chunks and use a global FIFO queue for the chunks, and a local LIFO stack for tasks in a chunk. HABANERO-CHORUS in contrast, requires only an extra annotation with respect to sequential Java code which is the fragment "`async isolated`" in line 2 to denote the scope of the isolated asynchronous task.

Chapter 5

Related Work

We itemize the work related to ours as follows.

The Galois project. Of related work, the closest to us is Kulkarni, Pingali et al's Galois project, which also aims to parallelize irregular, data-intensive applications (indeed, the Lonestar benchmarks that we use for assessing the performance of our approach came out of this project). The solutions there [27, 34] are to enhance thread-based, optimistic parallelism with data-types for unordered and partially ordered sets, which give the runtime information about data dependencies [27], and heuristics for data partitioning [34]. While these papers were inspirations for our work, they do not have a language design where structures like assemblies actively drive a parallel computation.

A more recent paper from the project [35] proposes *amorphous parallelism*—the combination of locality and dynamism that we also exploit—as the essence of parallelism in irregular applications, and establishes this with a profiling tool. A subsequent technical report [18] proposes an approach to amorphous parallelism using operators that morph and update local regions in a graph, substantiating this with a masterly review of irregular applications. However, these papers do not offer a concrete, high-level language, and do not have analogs of assemblies, merges and splits. In fact, our paper may be seen as a response to the hope expressed in the conclusion of [18]: "...that insights into amorphous data-parallelism in irregular algorithms will spur research into the design of abstract data types, concrete representations, and synchronization mechanisms."

PGAS languages. Also related are Partitioned Global Address Space languages like X10 [36, 37] and Chapel [38], which allow each thread to only access certain *partitions* in the heap (for example, in X10, accesses outside a task's "place" lead to exceptions, just like our out-of-region accesses). However, these languages do not offer high-level abstractions for dynamic creation and/or reconfiguration of places, and are thus unable to capture the dynamic growth and shrinkage of assemblies that is essential to our model.

Actors. The Actor model [39, 40] is one of the earliest data-centric models of concurrency, and is the backbone of concurrency in Erlang, Scala, and Singularity. Actors are active, concurrent entities that encapsulate some private data and communicate via asynchronous message-passing. One point of difference between Chorus and Actors is that the latter, in general, permits the transmission of references via messages, and is therefore not race-free. On the other hand, in a version of the Actor model where passing of references is disallowed, shared-memory applications such as ours would require copying of data. Finally, Chorus operates at a high-level, and does not need hand-coded message-passing.

More recent work on *isolation-typed*, ultralightweight actors [41] presents an actor framework where mutable messages do not have internal aliases, and can only be owned by a single actor at a time. This system—called Kilim—bears some similarities with ours. One difference is that it does not offer guarantees of deadlock-freedom. More importantly, isolation here is statically enforced, obtained by keeping the set of mutable objects reachable from one actor's local variables disjoint from another's. It seems that this would make Kilim unsuitable for irregular applications, where parallelism is highly instance-dependent, and involves complex data-structures like meshes and graphs that are almost impossible to analyze statically (the same critique applies to other static approaches to isolation, including ownership types [42, 43] and linear types [44]). In contrast, we forgo the static route altogether, instead building an execution model that maintains and reconfigures isolation dynamically via merges and splits.

Language-level transactions. The language-level transaction [45, 46, 23] is a composable, declarative abstraction for atomicity that is typically implemented using transactional memory. This abstraction, however, was not designed with locality of heap access in mind—as Kulkarni et al. [27, 34] point out, most im-

plementations of transactional memory track reads and writes to the entire memory to detect conflicts, and consequently perform poorly in irregular applications. Proposals to overcome these shortcomings include those for *privatizing* transactions [47] and combining them with data-centric synchronization [48]. While these systems share some design intentions with us, they do not have anything akin to our assembly-level parallelism or merge and split abstractions.

In this work a flavor of software transactions, HABANERO-CHORUS, is defined via a mapping into the Chorus abstract model. Hence the *Concurrent Assemblies* abstract model can be seen as a lower-level formalism than that of language-level transactions.

HABANERO-CHORUS provides a high-level minimalistic programming model similar to *Transactional Memory* [49], with a single construct (`async isolated`) to define blocks of code to be executed concurrently and in isolation. HABANERO-CHORUS guarantees livelock-freedom, which is heuristic-based and implementation-dependent in software transactional memory systems. One of the main objections to software implementations of transactional memory has been their inability to deliver competitive performance [50], HABANERO-CHORUS on the other hand, performs on par with Galois, perhaps the most successful existing approach to irregular data parallelism. Transactional memory is also not data centric.

Fine-grained Locking. Fine-grained locking has long been the most successful method of achieving high performance for irregular parallel applications. It has a data-centric approach, but is only usable by expert programmers. Programmability is very poor, and there are no safety guarantees. The inadequacy of non-expert programmers to express complex concurrent algorithms using fine-grained locking is the main reason for the proliferation of higher-level parallel programming models.

Programming models such as OpenMP [51] and Cilk [25] provide efficient support for *reductions* in deterministic parallel programs, but those constructs are not applicable to the nondeterministic, irregular parallelism supported by HABANERO-CHORUS.

Solaris uses a synchronization scheme similar to delegation in its FireEngine network stack [52]. The operating system has a fixed number of locks that corresponds to the number of processors, and every network connection is assigned to one of these locks using a hash. When a network connection requires processing,

e.g., a packet arrives, if the current processor cannot acquire the lock, it queues the packet for processing by the current lock holder. Aida implements a much more general programming model that is applicable to a wide range of applications (not just networking), does not require any operating system support, and the number of tasks is not limited to the number of processors in the system.

Chapter 6

Conclusions

Isolation has long been one of the most basic concerns in parallel programming. Despite the recent attention paid to software and hardware approaches to transactional memory, achieving a practical balance between programmability, correctness guarantees, and scalability in isolated execution has remained a thorny challenge for the community. In this dissertation, we have taken a step towards meeting this challenge in the context of a notoriously difficult problem domain: imperative parallel computations over large, shared, irregular data structures.

The *Concurrent Assemblies* execution model provides a sound basis for modeling parallel programs operating on shared irregular data structures. JCHORUS and HABANERO-CHORUS programming languages offer several high-level correctness guarantees. One can argue that JCHORUS is a suitable vehicle to implement cautious algorithms, allowing the programmer to reason about their correctness by only considering the relevant interleavings. On the other hand the HABANERO-CHORUS programming language requires minimal programmer expertise— indeed, the only extension needed to a fork-join parallel framework (such as OpenMP 3.0 or Habanero Java) is an isolated statement construct akin to that used in transactional memory. However, HABANERO-CHORUS execution model is shown to be orders-of magnitude faster on irregular applications than the DTSM2 software-based transactional memory system, and to yield performance comparable to that of fine-grain locking and the Galois system with a simpler and more general programming model.

Regarding future work, one immediate goal is develop a sound basis for nested

software transactions [53] by adding a notion of hierarchy to Concurrent Assemblies. As we discussed earlier, some characteristics of parallel algorithms, i.e. cautiousness, can be exploited to render more efficient programs. To further such an objective we aim to develop a static analysis toolkit for optimizing HABANERO-CHORUS programs. For example, some of our runtime overheads are unnecessary for *cautious applications* [15], and we will benefit from static analyses establishing that an assembly is cautious by using approaches such as [28]. On the language design end, we would like greater integration of HABANERO-CHORUS with existing Habanero Java constructs for locality and directed synchronization, such as hierarchical place trees [54] and phasers [55].

Finally, the *Concurrent Assemblies* model seems to provide notions that are very useful when considering distribution. The fact that connected areas of the heap are represented in the model provides a notion for data migration in distributed environments.

Bibliography

[1] Brooks, F. P., Jr. (1987) "No Silver Bullet Essence and Accidents of Software Engineering," *Computer*, **20**, pp. 10–19.
URL http://dx.doi.org/10.1109/MC.1987.1663532

[2] Allen, R. and K. Kennedy (1987) "Automatic translation of FORTRAN programs to vector form," *ACM Transactions on Programming Languages and Systems*, **9**, pp. 491–542.

[3] Wolfe, M. (1996) *High Performance Compilers for Parallel Computing*, Addison-Wesley.

[4] ——— (1986) "Advanced Loop Interchanging," *Proc. of the 1986 Int'l Conf. on Parallel Processing*, pp. 536–543.

[5] ——— (1992) "Beyond induction variables," in *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, ACM Press, New York, NY, USA, pp. 162–174.

[6] Kennedy, K. and J. R. Allen (2002) *Optimizing compilers for modern architectures: a dependence-based approach*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[7] Ruppert, J. (1995) "A Delaunay refinement algorithm for quality 2-dimensional mesh generation," in *Selected papers from the fourth annual ACM SIAM symposium on Discrete algorithms*, Academic Press, Inc., Orlando, FL, USA, pp. 548–585.
URL http://portal.acm.org/citation.cfm?id=203612.203634

[8] Lublinerman, R., S. Chaudhuri, and P. Cerny (2009) "Parallel Programming with Object Assemblies," in *OOPSLA*.

[9] Lublinerman, R., J. Zhao, Z. Budimlić, S. Chaudhuri, and V. Sarkar (2011) "Delegated isolation," in *Proceedings of the 2011 ACM international*

conference on Object oriented programming systems languages and applications, OOPSLA '11, ACM, New York, NY, USA, pp. 885–902.
URL http://doi.acm.org/10.1145/2048066.2048133

[10] Barnes, J. and P. Hut (1986) "A Hierarchical $O(NlogN)$ Force-Calculation Algorithm," *Nature*, **324**(4), pp. 446–449.

[11] Hildrum, K. and P. Yu (2005) "Focused Community Discovery," in *ICDM*, pp. 641–644.

[12] Aiello, W., C. Kalmanek, P. McDaniel, S. Sen, O. Spatscheck, and J. van der Merwe (2005) "Analysis of Communities of Interest in Data Networks," in *PAM*, pp. 83–96.

[13] Grune Yanoff, T. *Agent-Based Models as Policy Decision Tools: The Case of Smallpox Vaccination*, *Tech. rep.*, Royal Institute of Technology, Sweden.

[14] Jungnickel, D. and M. Swamy (2004) *Graphs, Networks, and Algorithms*, Springer.

[15] Méndez-Lojo, M., D. Nguyen, D. Prountzos, X. Sui, M. A. Hassaan, M. Kulkarni, M. Burtscher, and K. Pingali (2010) "Structure-driven optimizations for amorphous data-parallel programs," in *PPOPP*, pp. 3–14.

[16] Chew, P. (1993) "Guaranteed-Quality Mesh Generation for Curved Surfaces," in *Symposium on Computational Geometry*, pp. 274–280.

[17] Chung, S. and A. Condon (1996) "Parallel Implementation of Boruvka's Minimum Spanning Tree Algorithm," in *IPPS*, pp. 302–308.

[18] Pingali, K., M. Kulkarni, D. Nguyen, M. Burtscher, M. Mendez-Lojo, D. Prountzos, X. Sui, and Z. Zhong (2009) *Amorphous data-parallelism in Irregular Applications*, *Tech. Rep. TR-09-05*, University of Texas at Austin.

[19] "The Lonestar Benchmark Suite," Available from `http://iss.ices.utexas.edu/lonestar/`.

[20] Burke, D., J. Epstein, D. Cummings, J. Parker, K. Cline, R. Singa, and S. Chakravarty (2006) "Individual-Based Computational Modeling of Smallpox Epidemic Control Strategies," *Academic Emergency Medicine*, **13**(11), pp. 1142–1149.

[21] Galil, Z. and G. Italiano (1991) "Data structures and algorithms for disjoint set union problems," *ACM Comput. Surv.*, **23**(3), pp. 319–344.

[22] "DSTM 2.1 beta," Available from
`http://www.cs.brown.edu/~mph/`.

[23] Larus, J. and C. Kozyrakis (2008) "Transactional memory," *Communications of the ACM*, **51**(7).

[24] W. Scherer III and M. Scott (2005) "Advanced contention management for dynamic software transactional memory," in *PODC*, pp. 240–248.

[25] Blumofe, R. D. et al. (1995) "CILK: An efficient multithreaded runtime system," in *PPOPP*, pp. 207–216.

[26] "Habanero Multicore Software Research Project web page," http://habanero.rice.edu.

[27] Kulkarni, M., K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and P. Chew (2007) "Optimistic parallelism requires abstractions," in *PLDI*, pp. 211–222.

[28] Prountzos, D., R. Manevich, K. Pingali, and K. McKinley (2011) "A Shape Analysis for Optimizing Parallel Graph Programs," in *POPL*.

[29] Anderson, R. and H. Woll (1991) "Wait-free parallel algorithms for the union-find problem," in *STOC*.

[30] "9th DIMACS Implemetation Challenge," Available from
`http://www.dis.uniroma1.it/~challenge9/`.

[31] Budimlic, Z. et al. (2010) "Parallel Object-Oriented Scientific Computing with Habanero-Java," in *9th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing, co-located with OOPSLA'10*.

[32] "Galois," Available from
`http://iss.ices.utexas.edu/galois/`.

[33] Franaszek, P. A., J. T. Robinson, and A. Thomasian (1992) "Concurrency control for high contention environments," *ACM Trans. Database Syst.*, **17**, pp. 304–345.
URL http://doi.acm.org/10.1145/128903.128906

[34] Kulkarni, M., K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. Chew (2008) "Optimistic parallelism benefits from data partitioning," in *ASPLOS*, pp. 233–243.

[35] Kulkarni, M., M. Burtscher, R. Inkulu, K. Pingali, and C. Cascaval (2009) "How much parallelism is there in irregular applications?" in *PPOPP*, pp. 3–14.

[36] Charles, P., C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar (2005) "X10: an object-oriented approach to non-uniform cluster computing," in *OOPSLA*, pp. 519–538.

[37] Chandra, S., V. Saraswat, V. Sarkar, and R. Bodík (2008) "Type inference for locality analysis of distributed data structures," in *PPOPP*, pp. 11–22.

[38] Chamberlain, B., D. Callahan, and H. Zima (2007) "Parallel Programmability and the Chapel Language," *Int. Journal of High Performance Computing Applications*, **21**(3), pp. 291–312.

[39] Hewitt, C., P. Bishop, and R. Steiger (1973) "A Universal Modular ACTOR Formalism for Artificial Intelligence," in *IJCAI*, pp. 235–245.

[40] Agha, G., I. Mason, S. Smith, and C. Talcott (1997) "A Foundation for Actor Computation," *Journal of Functional Programming*, **7**(1), pp. 1–72.

[41] Srinivasan, S. and A. Mycroft (2008) "Kilim: Isolation-Typed Actors for Java," in *ECOOP*, pp. 104–128.

[42] Boyapati, C., R. Lee, and M. Rinard (2002) "Ownership types for safe programming: preventing data races and deadlocks," in *OOPSLA*, pp. 211–230.

[43] Clarke, D., T. Wrigstad, J. Östlund, and E. Johnsen (2008) "Minimal Ownership for Active Objects," in *APLAS*, pp. 139–154.

[44] Kobayashi, N., B. Pierce, and D. Turner (1996) "Linearity and the Pi-Calculus," in *POPL*, pp. 358–371.

[45] Harris, T. and K. Fraser (2003) "Language support for lightweight transactions," in *OOPSLA*, pp. 388–402.

[46] Harris, T., S. Marlow, S. L. P. Jones, and M. Herlihy (2005) "Composable memory transactions," in *PPOPP*, pp. 48–60.

[47] Spear, M. F., V. J. Marathe, L. Dalessandro, and M. L. Scott (2007) "Privatization techniques for software transactional memory," in *PODC*.

[48] Vaziri, M., F. Tip, and J. Dolby (2006) "Associating synchronization constraints with data in an object-oriented language," in *POPL*, pp. 334–345.

[49] Harris, T., J. R. Larus, and R. Rajwar (2010) *Transactional Memory, 2nd Edition*, Morgan & Claypool.

[50] Cascaval, C. et al. (2008) "Software Transactional Memory: Why Is It Only a Research Toy?" *Queue*, **6**, pp. 46–58.

[51] Chandra, R., L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon (2001) *Programming in OpenMP*, Academic Press.

[52] Tripathi, S. (2004) *FireEngine — A New Networking Architecture for the Solaris Operating System*, *Tech. rep.*, Sun Microsystems.

[53] Moss, J. E. B. and A. L. Hosking (2006) "Nested transactional memory: model and architecture sketches," *Sci. Comput. Program.*, **63**(2), pp. 186–201. URL http://dx.doi.org/10.1016/j.scico.2006.05.010

[54] Yan, Y., J. Zhao, Y. Guo, and V. Sarkar (2009) "Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement," in *The 22nd International Workshop on Languages and Compilers for Parallel Computing (LCPC'09)*.

[55] Shirako, J., D. Peixotto, V. Sarkar, and W. S. III (2008) "Phasers: a unified deadlock-free construct for collective and point-to-point synchronization," in *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, ACM, New York, NY, USA, pp. 277–288.

Vita

Roberto Lublinerman

Education

B.S., Computer Science, 2000, Universidad Católica del Uruguay, Montevideo, Uruguay.

Awards

CSE Graduate Student Teaching Award, The Pennsylvania State University, Spring 2008

Internships

Google, USA, June 2012-August 2012.
Cadence Berkeley Labs, USA, May 2008-August 2008.
Cadence Berkeley Labs, USA, May 2007-August 2007.
Verimag, France, June 1997 - August 1997.

Teaching and Research

Research Assistant, The Pennsylvania State University, USA, Fall 2008 - Present.
Teaching Assistant, The Pennsylvania State University, USA, Fall 2006 - Spring 2008.
Research Assistant, The Pennsylvania State University, USA, Fall 2004 - Spring 2006.
Profesor grado 3, Universidad Católica del Uruguay, Uruguay, 2000- 2004.

Professional

R & D Associate, ARTech, Uruguay, 1997 - 2003.
Software Engineer, ARTech, Uruguay 1992 - 1997.

Publications

S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour, **Proving Programs Robust**. (FSE'11) Winner of an ACM SIGSOFT Distinguished Paper Award.

R. Lublinerman, J. Zhao, Z. Budimlic, S. Chaudhuri, and V. Sarkar. **Delegated Isolation.** (OOPSLA'11).

S. Chaudhuri, S. Gulwani and R. Lublinerman. **Continuity Analysis of Programs**. (POPL'10).

R.Lublinerman, S.Chaudhuri and P.Černý. **Parallel Programming with Object Assemblies.** (OOPSLA'09).

R. Lublinerman, C. Szegedy and S. Tripakis. **Modular Code Generation from Synchronous Block Diagrams — Modularity vs. Code Size**. (POPL'09).

P. Caspi, A. Benveniste, R. Lublinerman, and S. Tripakis. **Actors without Directors: a Kahnian View of Heterogeneous Systems**. (HSCC'09).

R. Lublinerman and S. Tripakis. **Translating Data Flow to Synchronous Block Diagrams.** (ESTIMedia'08).

R. Lublinerman and S. Tripakis. **Modular Code Generation from Synchronous Block Diagrams.** (NSV'08).

G. Schindler, P. Krishnamurthy, R. Lublinerman, Y. Liu, and F. Dellaert, **Detecting and Matching Repeated Patterns for Automatic Geo-tagging in Urban Environments.** (CVPR'08)

Y. Liu, J. Hays and R. Lublinerman, **Image De-fencing**. (CVPR'08).

R. Lublinerman and S. Tripakis, **Modular Code Generation from Triggered and Timed Block Diagrams.** (RTAS'08)

R. Lublinerman and S. Tripakis, **Modular Modularity vs. Reusability: Code Generation from Synchronous Block Diagrams**. (DATE'08)

R. Lublinerman, N. Özay, D. Zarpalas and O. Camps, **Activity Recognition from Silhouettes using Linear Systems and Model (In)validation Techniques.** (ICPR'06).

R. Lublinerman, M. Sznaier and O. Camps, **Dynamics Based Robust Motion Segmentation.** (CVPR'06)

R. Lublinerman, **Formal Models based on Petri Nets, Abstract Data Types and Polymorphic Type Classes** (in Spanish). *ndergraduate Thesis. Universidad CatÃşlica del Uruguay. 2000.*