Parallel Programming with Object Assemblies

Roberto Lublinerman
Pennsylvania State University
University Park, PA 16802, USA
rluble@psu.edu

Swarat Chaudhuri
Pennsylvania State University
University Park, PA 16802, USA
swarat@cse.psu.edu

Pavol Černý
University of Pennsylvania
Philadelphia, PA 19104, USA
cernyp@cis.upenn.edu

Abstract

We present *Chorus*, a high-level parallel programming model suitable for irregular, heap-manipulating applications like mesh refinement and epidemic simulations, and *JChorus*, an implementation of the model on top of Java. One goal of Chorus 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.

Chorus addresses dynamism and locality through the unifying abstraction of an *object 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 and deadlock-free, and inherently data-centric.

We demonstrate that Chorus and JChorus allow natural programming of several important applications exhibiting irregular data-parallelism. We also present an implementation of JChorus based on a many-to-one mapping of assemblies to lower-level threads, and report on preliminary performance numbers.

Categories and Subject Descriptors D.3.2 [*Programming Techniques*]: Concurrent Programming; D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed, and parallel languages

General Terms Languages, Design

Keywords Parallel programming, Programming abstractions, Irregular parallelism, Data-parallelism, Ownership

OOPSLA'09, October 25–29, 2009, Orlando, Florida, USA.

1. Introduction

Calls for new programming models for parallelism have been heard often of late [29, 33]. On one hand, the demand for parallel programming is now higher than ever: inexpensive multicore machines are now near-ubiquitous, and the bottleneck in this space is now software rather than hardware. On the other hand, it is increasingly clear that the currently popular models of parallel programming—locks and messages—are too low-level, complex, and error-prone, and do not scale well with software complexity. Consequently, numerous teams of programmers and researchers are seeking high-level models of programming that are intuitive as well as scalable.

Not all parallel programming, of course, poses difficult challenges: "embarrassingly parallel" applications are easy to program efficiently, and for regular array-based codes, static parallelization [35] is known to work. Far more challenging is the problem of efficiently coding applications that combine parallelism with accesses to sparse, mutable data structures like trees and graphs. Parallelism in such *irregular* applications [24] is highly input-dependent and prone to changes at runtime, with some pathological instances exhibiting no parallelism at all. This makes compile-time parallelization impossible; in fact, Kulkarni et al. have noted [27] that most current implementations of *optimistic* parallelism (using transactional memory [28]) suffer in this setting as well. Vexingly, numerous important scientific and graphical applications where parallelism is needed fall in this category—examples include physical [17] or epidemiological simulations [16], mesh refinement [11], spanning tree computations [22], n-body simulation [5], social network analysis [21], and sparse matrix computations [14].

Consequently, many of these applications are perfect challenge problems for designers of new models of shared-memory parallel programming. This understanding is reflected, for example, in the recently released Lonestar benchmarks [2, 25], which offer code and datasets for several such

```
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 1. Delaunay mesh refinement: sequential algorithm

problems. In this paper, we present a response to this challenge: a data-centric programming model called *Chorus*, and a concrete language called *JChorus*, implementing it.

1.1 Locality and dynamism

The key insight behind Chorus is one also identified by Pingali, Kulkarni et al. [24, 30]: while typical irregular applications require global memory access in the worst case, the effects of their imperative updates are usually restricted to small, *local* regions in the heap. This locality forms the essence of parallelism in this setting (Pingali and Kulkarni call it *amorphous parallelism*), and Chorus gives it a first-class treatment by exposing it at the language level. As such locality is highly instance-dependent and intractable through static analysis, Chorus negotiates it *dynamically* rather than through static data partitioning.

For example, consider Delaunay mesh refinement, a classic and thoroughly studied [27] irregular application. The input here is a triangulation of a plane—viewed as a graph where nodes are triangles and edges represent adjacency—and a subset of "bad" triangles that do not meet certain quality constraints. The problem is to retriangulate the mesh so that there are no bad triangles left. It is a property of the application that such retriangulation affects a "cavity": a local region in the mesh. A sequential algorithm (Figure 1) for Delaunay mesh refinement uses the following property: in each iteration of a loop, the algorithm builds a cavity `c` consisting of a bad triangle `t` drawn from a worklist `wl` of bad triangles, *expands* `c` to the needed extent, and locally *updates* it using a retriangulation routine.

In this algorithm, there is no theoretical bound on the sizes of cavities rewritten by `c.retriangulate()`—at worst, they may encompass the entire mesh. Therefore, cavities need to be identified *dynamically* through the `expand` routine. At the same time, in practice, cavities are almost always small (see Figure 2), so that imperative modifications to the mesh are *local*. This combination of locality and dynamism appears in numerous other irregular applications known to be difficult to parallelize—indeed, it shows up in *every irregular application* that we have considered.

Exploitation of parallelism in Delaunay refinement and many other irregular applications is directly tied to the locality. In parallel Delaunay refinement, we need to guarantee that cavities—local regions—are the units of space that need to be accessed and retriangulated atomically; cavities that do not overlap can be retriangulated in parallel. Thus, the programming pattern for a thread is:

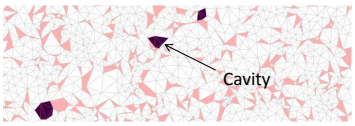

Figure 2. Snapshot of a Delaunay mesh from our experiments

> "Dynamically identify a region to own; establish ownership over it. Atomically update the region. Relinquish ownership over the region."

And yet, no current language for shared-memory parallelism can express this combination of locality and dynamism in a general way. In popular multithreaded languages like Java or C#, the heap is a global pool: unless explicitly locked, a shared object can be accessed by any thread at any time. Locality of data structure accesses is not expressed in the program text, and there are no abstract primitives capturing ownership of, and contention for, regions.

This global nature of shared-memory accesses has a negative effect on programmability as well as performance. In lock-based programming, the programmer manually manages references that can be aliased globally, leading to races and deadlocks. In languages using non-blocking software transactions, the burden of reasoning about global accesses is passed to the transaction manager—consequently, in most implementations of software transactional memory, the transaction manager must track reads and writes to the entire memory to detect conflicts. As Kulkarni et al. [27, 26] point out, this makes them behave inefficiently while handling large irregular applications. Global heap-manipulation also makes precise static analysis extremely difficult.

While Partitioned Global Address Space languages like X10 [10] and Chapel [8] allow language-level *partitioning* of the heap, they do not permit dynamic, lightweight creation and reconfiguration of the partitions. This makes them unable to grapple with the unpredictable, dynamic nature of irregular applications. Partitions that looked useful initially could very well turn useless during the execution, so that any workable data partitioning strategy for this setting must be *adaptive*.

At the other end of the spectrum are programming models such as Actors and Active Objects [20, 3, 32, 13], where data is encapsulated within actors, and memory accesses are modeled by message-passing. In an actor-based encoding of Delaunay mesh refinement, each actor possesses a set of cavities, and cavity expansion is modeled by the passing around of triangles between actors. While such an implementation captures the locality of the problem, coordination in it is low-level, error-prone, and potentially expensive. For one, the simple `c.expand()` method is now replaced by a

protocol between actors that has to be carefully coded to guarantee properties like deadlock-freedom. Second, copying and sending data involves high overheads, and if references rather than data are passed around, there is the potential of data races. Third, the *speculative* parallelism that has been argued [27, 26] to be needed for irregular applications seems hard to encode within the actor framework.

Thus, none of these styles of programming allow for high-level expression of ownership of regions in the heap (cavities in case of mesh refinement), and dynamic reconfiguration of such ownership. Chorus, on the other hand, is designed precisely to capture these programming patterns.

1.2 Our solution

The Chorus programming model 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 an *object assembly*: a dynamically defined local region in the heap equipped with a short-lived, speculative 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, Chorus permits massive, object-level parallelism. Of course, assemblies are just programmer 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 assembly with which it merges is terminated. An assembly is speculative, meaning it can merge with, and terminate, a neighboring assembly without explicit "consent" from the latter.
- It can *split* into a collection of smaller (disjoint) assemblies, each possessing a new thread of control.

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*. There is no global ordering between merges and splits—e.g., merges between distinct pairs of assemblies can always happen in parallel.

For an application, 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 merge calls to 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."

Our concrete contributions are the following:

- We introduce the Chorus programming model, present a core language for Chorus and its operational semantics, and show it to be free of data races and deadlocks.
- We present JChorus, a programming language that embeds our model of concurrency into the sequential subset of Java.
- We demonstrate the utility of JChorus in programming real-life applications via several case studies. In addition to mesh refinement, we consider the problems of Barnes-Hut n-body simulation [5], "Focused community" discovery in a social network [21, 4], an epidemiological simulation problem [16], and an algorithm for computing minimum spanning trees [22].
- We present a prototype compiler and runtime system for JChorus[1] that use 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 report performance numbers for two irregular applications: Delaunay mesh refinement and Boruvka's algorithm for minimum-spanning-tree computation.

The paper is organized as follows. In Section 2, we introduce Chorus and study its properties. Section 3 outlines the JChorus language, and Section 4 demonstrates our case studies. Section 5 describes our implementation of JChorus; Section 6 reports on performance numbers. Related work is discussed in Section 7; we conclude with some discussion in Section 8.

2. Chorus

Now we present the main features of the Chorus programming model. We start with an informal description of the

[1] The prototype, as well as our encodings of these benchmark examples, are available at `http://www.cse.psu.edu/~swarat/chorus`.

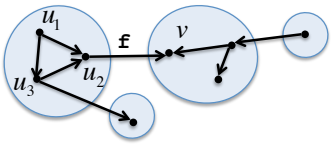

Figure 3. A heap

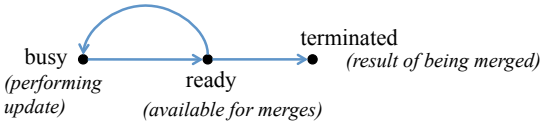

Figure 4. Control flow in assemblies

available programming constructs, then offer a more formal presentation using a core language.

2.1 Essential Chorus

2.1.1 Heaps

The central structure in Chorus 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 3, 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.

2.1.2 Object assemblies

An *object assembly* in G is a region of G equipped with a set of local variables and a sequential thread of control. The typical execution scenario of Chorus programs 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.

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

$$:: \ Guard : Update$$

where $Guard$ is a condition that is evaluated atomically, and $Update$ 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 4) 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 $Update$. 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.

Thus, a merge is a synchronization operation—in fact, it is our only synchronization operation. Figures 5-(a) and 5-(c) show the states of a parallel program before and after the assembly i_1 merges with i_2. Note that the operation *locally coarsens* the granularity of parallelism in the heap.

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 5-(a) and 5-(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.

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

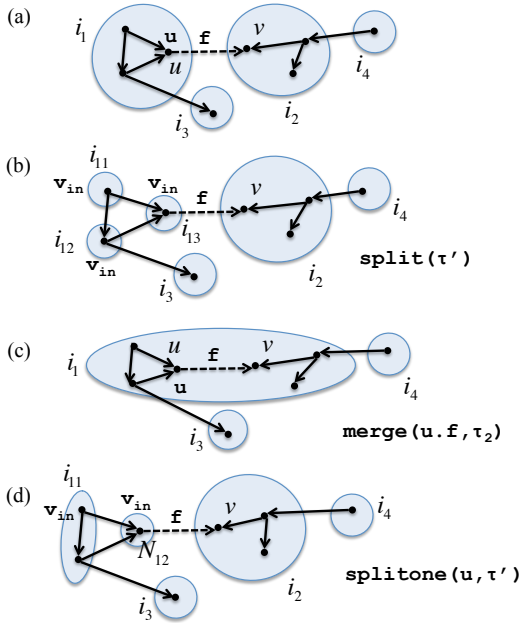

Figure 5. Merging and splitting

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. 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.

2.1.3 Merging, splitting, and heap modification

Now we examine the programming constructs that Chorus 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 $\mathtt{u}$ refers to the object u and u has an $\mathtt{f}$-labeled edge to v, then the name $\mathtt{u.f}$ refers to v.

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

$$:: \ \mathtt{merge}(\mathtt{u.f}, \tau_2) : \mathit{Update}.$$

Here, τ_2 is an assembly class, $\mathtt{u}$ is a variable in the assembly i_1 (of class τ_1) executing the guarded update, and $\mathtt{u.f}$ refers to an object in a different assembly i_2. For the guard here to be *enabled*, i_2 must be of class τ_2 and 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 Update is now executed. For example, if the assembly i_1 in Figure 5-(a) is able to execute this command, a heap as in Figure 5-(c) results.

A second merge construct allows i_1 (the assembly performing the merge) to *copy* the local-variable state of i_2 into a local variable. To see the usefulness of this construct, note that the merge here terminates i_2. While this is permissible as i_2 is in its ready state, the local state of i_2 may contain information that i_1 may benefit from.

This construct has the form

$$:: \ \mathtt{merge}(\mathtt{u.f}, \tau_2, \mathtt{v}_1 := \mathtt{v}_2) : \mathit{Update}$$

where $\mathtt{v}_1$ is a local variable in assemblies of class τ_1 and $\mathtt{v}_2$ is a local variable in assemblies of class τ_2. A merge here is as before, except in addition, the variable $\mathtt{v}_1$ is set during the merge to the value of $\mathtt{v}_2$ in i_2.

We also allow a construct

$$:: \ \mathtt{merge}(\mathtt{u.f}, \tau_1', \tau_2\ [\mathtt{v}_1' := \mathtt{v}_1, \mathtt{v}_2' := \mathtt{v}_2]) : \mathit{Update}$$

for "reflective" merge (the syntax within the square brackets is optional). Here τ_1' and τ_2 are assembly classes, $\mathtt{u}$ and $\mathtt{v}_1$ are variables in τ_1, $\mathtt{v}_1'$ and $\mathtt{v}_2'$ are variables in τ_1', and $\mathtt{v}_2$ is a variable in τ_2.

This merge terminates both i_1 and i_2, creating a fresh assembly i_1' of class τ_1'. The local variables of i_1' are initialized in accordance with the declaration of τ_1'. In addition, we have some optional "parameter-passing": the variables $\mathtt{v}_1'$ and $\mathtt{v}_2'$ of i_1' are respectively set to the values of $\mathtt{v}_1$ in i_1, and $\mathtt{v}_2$ in i_2. The initial control state of i_1' is *ready*. As i_1 is terminated, the command Update is not executed.

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

$$:: \ \mathtt{merge}(\mathtt{u.f}, \tau_2)\ \mathtt{when}\ g : \mathit{Update}$$

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. We allow two syntactic forms for splitting i_1. Of these,

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

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.

```
assembly Triangle:: ...
    :: merge (v.f, Cavity, Triangle) when isBad: skip

assembly Cavity:: ...
    :: merge (v.f, Cavity) when (not isComplete): skip
    :: isComplete:
          retriangulate(); split(Triangle)
```

Figure 6. The essence of Delaunay Mesh Refinement in Chorus

As with merges, the heap itself is not modified during a split. Figure 5-(b) shows the assemblies i_{11}, i_{12}, and i_{13} resulting from splitting i_1 in Figure 5-(a).

The other split-construct has the form

$$\texttt{splitone}(\texttt{u}, \tau').$$

Suppose u is the object named u in the assembly i. When i executes this update, a new assembly i' that is of class τ' and contains only u is created (each local variable of i' points to u). The original assembly i continues to execute; however, as u is outside the region of the assembly after the split, all local variables of i that pointed to u are de-initialized.

Figure 5-(d) shows the result of splitting i_1, as in Figure 5-(a), in this way.

Finally, updates in i_1 can contain arbitrary sequential code referring to the local variables and region of i_1. However, the evaluation of any field expressions referring to objects outside i_1 raises an exception.

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

2.1.4 Example: Delaunay mesh refinement

Let us now go back to the Delaunay mesh refinement problem briefly mentioned in Section 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). In many applications [11], 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 1. Initially, the worklist is populated by bad triangles from the original mesh. For each bad triangle t, the algorithm proceeds as follows[2]:

- 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 2). 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.
- 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 4). For now, we show how to capture its essence in Chorus. Here, the triangulation at any point is modeled by a heap whose objects are triangles, and whose pointers connect adjacent triangles. Assemblies belong to two classes: `Triangle` and `Cavity`. Initially, every triangle in the heap is in its own assembly (of class `Triangle`).

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 variable `isComplete` that, at any point, is *true* iff the cavity 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 6. 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 1 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 its component triangles. The program terminates when the only assemblies in the mesh are "good" `Triangle`-s.

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 times 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

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

$$
\begin{array}{lcl}
\mathit{Prog} & ::= & [\mathit{ADec}]^* \\
\mathit{ADec} & ::= & \texttt{assembly}\ \tau :: \texttt{local}\ [\texttt{v}]^* \\
 & & \quad [::\ \mathit{Guard} : \mathit{Update}]^* \\
\mathit{Guard} & ::= & \texttt{merge}(\texttt{v.f}, \tau_2\ [, \texttt{v}_1 := \texttt{v}_2]) \mid \\
 & & \texttt{merge}(\texttt{v.f}, \tau_1', \tau_2\ [, \texttt{v}_1' := \texttt{v}_1]\ [, \texttt{v}_2' := \texttt{v}_1]) \mid \\
 & & \texttt{merge}(\texttt{v.f}, \tau_2\ [, \texttt{v}_1 := \texttt{v}_2])\ \texttt{when}\ \mathit{Bexp} \mid \\
 & & \texttt{merge}(\texttt{v.f}, \tau_1', \tau_2\ [, \texttt{v}_1' := \texttt{v}_1]\ [, \texttt{v}_2' := \texttt{v}_2]) \\
 & & \quad \texttt{when}\ \mathit{Bexp} \mid \mathit{Bexp} \\
\mathit{Update} & ::= & \texttt{v} := \mathit{Exp} \mid \texttt{v.f} := \mathit{Exp} \mid \mathit{Update}; \mathit{Update} \mid \\
 & & \texttt{skip} \mid \texttt{split}(\tau) \mid \texttt{splitone}(\texttt{v}, \tau) \\
\mathit{Exp} & ::= & \texttt{v} \mid \mathit{Exp}.\texttt{f} \mid \texttt{error} \\
\mathit{Bexp} & ::= & \mathit{Exp} = \mathit{Exp} \mid \texttt{not}\ \mathit{Bexp} \mid \mathit{Bexp}\ \texttt{and}\ \mathit{Bexp}
\end{array}
$$

where
$\texttt{v}, \texttt{v}_1, \texttt{v}_2, \texttt{v}_1', \texttt{v}_2' \in \mathit{Var}$, $\tau, \tau_1, \tau_2, \tau_1' \in \mathcal{T}$, and $\texttt{f} \in F$. Syntax within square brackets is optional, and $[t]^*$ denotes zero or more occurrences of t.

Figure 7. Syntax of Core Chorus

`retriangulate()`, which is executed atomically in one update.

2.2 A core language for Chorus

Now we present Chorus more formally using a core programming language called *Core Chorus*. As our goal is to capture the essence of *concurrency* in Chorus, 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.

2.2.1 Syntax

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 Core Chorus is given in Figure 7. Here:

- Prog represents programs.
- ADec represents declarations of assembly classes. The declaration consists of a sequence of *local variable declarations* and definitions of *guarded updates*.
- Guard represents *guards*; Update represents *updates*.
- Exp and Bexp respectively represent pointer expressions and boolean expressions.

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 $\mathit{Var}(\tau)$
- the set of guarded updates in τ by $\mathit{Act}(\tau)$.

2.2.2 Semantics

Now we give a formal semantics of Core Chorus. In this presentation, we assume a fixed but arbitrary program P and identify assemblies with unique IDs. However, these IDs appear only in the lower-level representation and are not part of the programming abstraction.

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 $G = (O \subseteq \mathit{Loc}, E \subseteq O \times F \times O)$, 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.

Letting $\mathcal{I}$ be a universe of assembly IDs, we have the following definitions for assemblies and their states:

Definition 2 (Assembly)**.** An *assembly* is an element $i \in \mathcal{I}$. For a heap G, an *assembly state* of i in G is a tuple $N = \langle i, \tau, H, \mu, S\rangle$, where τ is an assembly class, H is a region in G, $\mu : \mathit{Var}(\tau) \to (H \cup \texttt{error})$, and S is either an update or the special symbol ϵ.

Two assembly states $N_1 = \langle i_1, \tau_1, H_1, \mu_1, S_1\rangle$ and $N_2 = \langle i_2, \tau_2, H_2, \mu_2, S_2\rangle$, where $i_1 \neq i_2$, are *disjoint* if H_1 and H_2 do not have nodes in common.

Intuitively, in the definition of the assembly state N of the assembly i, H is the region on which i operates, μ provides mappings from the local variables of τ to objects in H (or the "uninitialized" value `error`), and S gives the *control state* of i (at a lower level of abstraction than Figure 4). If $S = \epsilon$, then i is currently at the *ready* control state. Otherwise, it is the update that i must execute before it can return to ready-state behavior of τ. For an assembly state $N = \langle i, \tau, H, \mu, S\rangle$, we write $\mathit{Id}(N) = i$.

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

Definition 3 (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 assembly states 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: (1) $\mathit{Id}(N_1) \neq \mathit{Id}(N_2)$, and (2) N_1 and N_2 are disjoint.

Now we present an interleaving operational semantics of Core Chorus. 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 G' is obtained by repeatedly rewriting G in *isolation*—i.e., without requiring

(ASSEMBLY-STEP) $$\frac{N = \langle i, \tau, H, \mu, S\rangle \in \Gamma \quad \langle H, \{N\}\rangle \longrightarrow \langle H', \Gamma'\rangle}{\langle G, \Gamma\rangle \longrightarrow \langle G[H \rightsquigarrow H'], \Gamma \setminus \{N\} \cup \Gamma'\rangle}$$

(MERGE-1)
$$\frac{\begin{array}{c} N_1 = \langle i_1, \tau_1, H_1, \mu_1, \epsilon\rangle \in \Gamma \quad N_2 = \langle i_2, \tau_2, H_2, \mu_2, \epsilon\rangle \in \Gamma \quad i_1 \neq i_2 \quad \mathtt{v}, \mathtt{v}_1 \in \mathit{Var}(\tau_1) \quad \mathtt{v}_2 \in \mathit{Var}(\tau_2) \\ (\mathtt{merge}(\mathtt{v.f}, \tau_2, \mathtt{v}_1 := \mathtt{v}_2) : S) \in \mathit{Act}(\tau_1) \qquad \mu(\mathtt{v}) = u \qquad u \xrightarrow{G,\mathtt{f}} v \qquad v \text{ is in } H_2 \end{array}}{\langle G, \Gamma\rangle \longrightarrow \langle G, \Gamma \setminus \{N_1, N_2\} \cup \{\langle i_1, \tau, H_1 \sqcup_G H_2, \mu_1[\mathtt{v}_1 \mapsto \mu_2(\mathtt{v}_2)], S\rangle\}\rangle}$$

(MERGE-2)
$$\frac{\begin{array}{c} N_1 = \langle i_1, \tau_1, H_1, \mu, \epsilon\rangle \in \Gamma \quad N_2 = \langle i_2, \tau_2, H_2, \mu_2, \epsilon\rangle \in \Gamma \quad i_1 \neq i_2 \quad \mathtt{v}'_1, \mathtt{v}'_2 \in \mathit{Var}(\tau'_1) \quad \mathtt{v}_1 \in \tau_1, \mathtt{v}_2 \in \tau_2 \\ (\mathtt{merge}(\mathtt{v.f}, \tau'_1, \mathtt{v}'_1 := \mathtt{v}_1, \mathtt{v}'_2 := \mathtt{v}_2) : S) \in \mathit{Act}(\tau_1) \qquad \mu_1(\mathtt{v}) = v \qquad v \xrightarrow{G,\mathtt{f}} w \qquad w \text{ is in } H' \end{array}}{\langle G, \Gamma\rangle \longrightarrow \langle G, \Gamma \setminus \{N, N'\} \cup \{\langle i, \tau', H \sqcup_G H', \mu_{\mathit{Init}}(\tau', w)[\mathtt{v}'_1 \mapsto \mu_1(\mathtt{v}_1), \mathtt{v}'_2 \mapsto \mu_2(\mathtt{v}_2)], \epsilon\rangle\}\rangle}$$

(EXP-1) $$\frac{N = \langle i, \tau, H, \mu, S\rangle \quad \mu(\mathtt{v}) = u}{\langle N, \mathtt{v}\rangle \dashrightarrow u}$$

(EXP-2) $$\frac{N = \langle i, \tau, H, \mu, S\rangle \quad \langle N, e\rangle \dashrightarrow u \quad u \xrightarrow{H,\mathtt{f}} v \quad v \text{ is in } H}{\langle N, e.\mathtt{f}\rangle \dashrightarrow v}$$

(EXP-3) $$\frac{\langle N, e\rangle \dashrightarrow \mathtt{error}}{\langle N, e.\mathtt{f}\rangle \dashrightarrow \mathtt{error}}$$

(EXP-4) $$\frac{N = \langle i, \tau, H, \mu, S\rangle \quad \langle N, e\rangle \dashrightarrow u \quad u \xrightarrow{H,\mathtt{f}} v \quad v \text{ is not in } H}{\langle N, e.\mathtt{f}\rangle \dashrightarrow \mathtt{error}}$$

(GUARDS-WITHOUT-MERGES) $$\frac{N = \langle i, \tau, H, \mu, \epsilon\rangle \quad (g : S) \in \mathit{Act}(\tau) \quad \langle N, g\rangle \dashrightarrow \mathit{true}}{\langle H, \{N\}\rangle \longrightarrow \langle H, \{\langle i, \tau, H, \mu, S\rangle\}\rangle}$$

(ASSIGN-1) $$\frac{N = \langle i, \tau, H, \mu, \mathtt{v} := e\rangle \quad \langle N, e\rangle \dashrightarrow u}{\langle H, \{N\}\rangle \longrightarrow \langle H, \{\langle i, \tau, H, \mu[\mathtt{v} \mapsto u], \epsilon\rangle\}\rangle}$$

(ASSIGN-2) $$\frac{N = \langle i, \tau, H, \mu, \mathtt{v.f} := e\rangle \quad \mu(\mathtt{v}) = u \quad \langle N, e\rangle \dashrightarrow v \quad H' = H[(u, \mathtt{f}, \_) \rightsquigarrow (u, \mathtt{f}, v)]}{\langle H, \{N\}\rangle \longrightarrow \langle H', \{\langle i, \tau, H', \mu, \epsilon\rangle\}\rangle}$$

(SEQUENCE) $$\frac{N = \langle i, \tau, H, \mu, S\rangle \quad \langle H, \{N\}\rangle \longrightarrow \langle H', \{\langle i, \tau, H', \mu', \epsilon\rangle\}\rangle}{\langle H, \{\langle i, \tau, H, \mu, (S; S')\rangle\}\rangle \longrightarrow \langle H', \{\langle i, \tau, H', \mu', S'\rangle\}\rangle}$$

(SPLIT) $$\frac{N = \langle i, \tau, H, \mu, \mathtt{split}(\tau')\rangle \quad H \text{ has objects } u_1, \ldots, u_n \quad i_{u_1}, \ldots, i_{u_n} \text{ are globally unique fresh IDs}}{\langle H, \{N\}\rangle \longrightarrow \langle H, \{\langle i_{u_j}, \tau', \{u_j\}, \mu_{\mathit{Init}}(\tau', u_j), \epsilon\rangle : 1 \leq j \leq n\}\rangle}$$

Figure 8. Structural operational semantics of Core Chorus. Rules for some constructs omitted.

information about objects outside of G. Thus, the semantics is in a sense *modular over space*.

Values in this semantics can be objects, *true*, *false*, 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 state and e is a pointer or boolean expression. 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'.

  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 Core Chorus is presented in Figure 8 (we omit rules for several constructs). 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 state $\langle 0, \iota, G_{in}, \mu_{\mathit{Init}}(\iota), \epsilon\rangle$.

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, $\sigma_i \longrightarrow \sigma_{i+1}$.

Termination. A *terminating program state* in our model is one where each assembly is in the ready state, and yet the program cannot carry out a merge. This can happen only when every assembly i is in a ready state and satisfies one of the following properties: (1) i has no outgoing edges; (2) each merge that i can perform is constrained by a boolean condition, and all such conditions evaluate to $false$. (An assembly i as above is said to be in an *inactive state*.)

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

2.3 Race- and deadlock-freedom

Race-freedom. A *data race* happens when two threads concurrently access a shared object, and at least one of these accesses is a write. Updates in Chorus 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 Chorus 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 $(u, \mathtt{f}, v)$ 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 Chorus, an assembly in a ready control state cannot prevent itself from being merged even if it has locally enabled merges. 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 5 for more details.

3. The JChorus language

The JChorus language embeds the Chorus model into the sequential subset of Java. Instead of presenting the complete syntax and semantics of the language, we outline its main differences from Core Chorus and sequential Java using examples.

Object classes. Unlike in Core Chorus, 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 method calls.

For example, consider Figure 9, which shows part of a JChorus implementation of Delaunay mesh refinement (the essence of the program was previously shown in Figure 6). 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 9 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.

Unlike in the core language, transfer of local state between assemblies during merges and splits is now captured via structured parameter-passing. Specifically, the declaration of an assembly class τ now contains a *constructor* of the form $\tau(\mathtt{f_1}, \ldots, \mathtt{f_k})\{\ldots\}$. Here, the $\mathtt{f_i}$-s are the formal parameters of the constructor, which are used to initialize the local variables of newly created assemblies of this class. A command for creating a new assembly (e.g., a split or a "reflective" merge) now also carries actual parameters evaluated in the context from which creation happens.

Transfer of local state during a merge is performed using the following mechanism. We declare some local variables in assemblies as *public* in their class definitions. If an assembly i_1 merges with another assembly i_2, the public variables of i_2 are accessible to i_1 after the merge (note that i_2 dies during the merge).

Guarded updates in JChorus have the Java-style syntax $Guard \ : \ \{ \ Update \ \}$. In addition to the forms of merges shown earlier, we also permit the syntax $\mathtt{merge}(\mathtt{L}, \tau, \mathtt{x}) \ : \ \{Update\}$ for merges, 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

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

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

14:   Set members; Set border;
15:   Queue frontier;  // current frontier
16:   List outgoingedges;    // outgoing edges on which
                             // to merge
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 9. Delaunay mesh refinement in JChorus

whose target v is contained in L—the precise value of v used for the merge is bound to the name x within the body of *Update*.

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 9 for an example. As mentioned earlier, split-operations can pass parameters, having forms such as $\texttt{split}(\texttt{p}_1, \ldots, \texttt{p}_k)$. In addition to the standard split, we have an operation $\texttt{splitmany}(\texttt{L}, \texttt{p}_1, \ldots, \texttt{p}_k)$ that splits off *all* elements in the collection L from the current assembly (passing the $\texttt{p}_i$-s as parameters).

Finally, in JChorus, we permit an assembly of class τ to change its class to τ' via a command $\texttt{become}(\tau', \texttt{p}_1, \ldots, \texttt{p}_k)$ ($\texttt{p}_1, \ldots, \texttt{p}_k$ are parameters passed to the constructor of the newly created assembly). For example, in Figure 9, a Triangle becomes a Cavity this way.

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 $\texttt{parallel}(\tau(\texttt{p}_1, \ldots, \texttt{p}_k))$ which constructs a assembly of type τ ($\texttt{p}_1, \ldots, \texttt{p}_k$ 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 13).

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 Core Chorus, 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 full 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 13). This is because in Chorus, any writable object that an assembly i can access is guaranteed to be exclusively owned by i. However, the reverse, if carried out

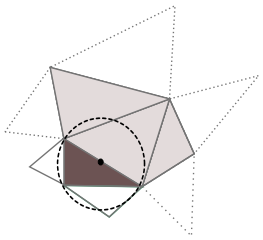

Figure 10. A cavity in construction, its center triangle (in dark), the cavity triangles (light gray) and the border triangles (dotted lines). The white triangles are either part of the frontier or the destination of one of the outgoing edges.

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.

4. 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 [22], Barnes-Hut n-body simulation [5], "focused community" discovery in social networks [21], and a problem of agent-based epidemiological simulation [16].

4.1 Delaunay Mesh Refinement

Now we describe in detail the implementation of Delaunay Mesh Refinement in JChorus. Section 2.1.4 describes the main ideas behind this algorithm using an implementation in Core Chorus. This example illustrates some of the constructs available in JChorus that are not present in the core version of the language. Figure 9 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 `TriangleObjects`, and end by splitting into `Triangles` assemblies, one for each `TriangleObject`. Lines 1–6 implement the assembly `Triangle`; if a triangle 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 caught (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.

4.2 Minimum-spanning tree

Now we use JChorus to code a classic, irregular graph algorithm well-known in the parallel programming literature [12, 30] : Boruvka's algorithm for computing the minimum spanning tree of a graph [22]. 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 11.

4.3 Barnes-Hut n-body Simulation

Now we consider the Barnes-Hut algorithm for n-body simulation [5], an application in the Lonestar benchmarks [2]. The algorithm simulates movements of 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

```
 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, ComputeSpanningTree) :
10:          computeNewMinEdge();
11:    }
    // computes the minimal edge of the merged tree
12:    void computeNewMinEdge() { ...}
   }
```

Figure 11. Minimum spanning tree

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 [5] introduced an $O(n \log n)$ time based on the idea that 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 12. For more details on the algorithm and the octree data structure, the reader is referred to [5].

```
 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 12. n-body simulation

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 12 stays conceptually the same – timesteps are executed serially. Figure 13 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 that will be split. Both phases 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). Subsequently, the forces acting on each body are computed. 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, this process is all implemented in the constructor (lines 25-29).

```
 .....
 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, ComputeMass)
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 13. n-body simulation in JChorus

4.4 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 [16] describes several such approaches used for modeling the spread of the smallpox virus.

We consider the model of Burke et al. [7]. 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

```
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 14. Focused Communities

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

4.5 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 [21, 4]). A data set connected to this problem is a part of the Lonestar Benchmark Suite.

We consider an algorithm for focused community discovery from [21]. Figure 14 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 15 has a picture of the algorithm in progress. At each step, the algorithm checks:

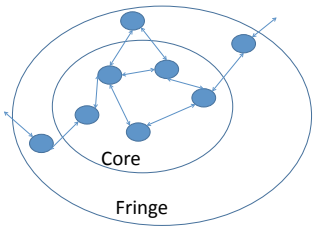

Figure 15. Focused Communities: the core and the fringe.

- For each node in the core, whether removing this node would increase the 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 14. 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 [21]. Yet, that paper was concerned with the algorithmic rather than the programming-language aspect of focused community discovery. We view this as evidence that Chorus is a natural programming model for this application.

5. Implementing JChorus

The assembly-level parallel exposed by Chorus is potentially massive. For example, running the refinement algorithm in Figure 1 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.

5.1 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 16.

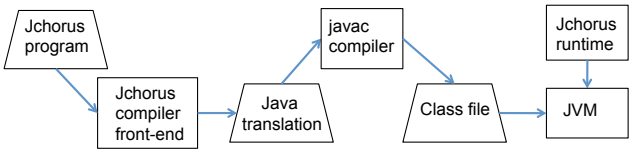

Figure 16. The JChorus compiler and runtime system

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

5.1.1 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 17 shows a heap and some divisions in it. The initial divisions are roughly equal in size, and are obtained heuristically from the input dataset.

5.1.2 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 18) 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

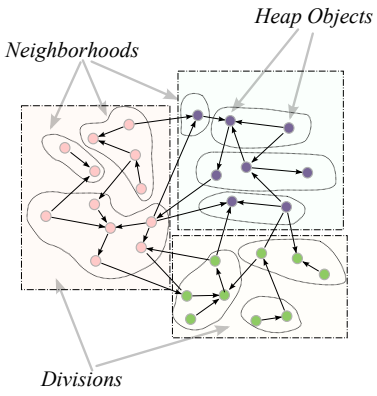

Figure 17. Divisions in a heap

out the merge right away, instead placing the pair $(i, \mathtt{o})$ 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 5.1.3 for explanations on Line 10). In case the target of a remote merge 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

```
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 18. Executor thread loop for division D

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 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 deadlock, 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 deadlocks perhaps extremely unlikely. In particular, we did not face deadlocks in our experiments with this scheduler.

Perhaps this scheduler can also benefit from *randomized* strategies for deadlock 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.

5.1.3 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 18). 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.

5.1.4 Use of Union-Find

The Chorus 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 maintaining 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 [15]. 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 [15].

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.

6. 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 [1, 28].

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.[3]

6.1 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 [2]—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 4.1 and Figure 9. 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 18 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.

As we worked on an 8-core system, we ran these competing approaches varying the number of threads from 1 to 8. Figure 19 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.

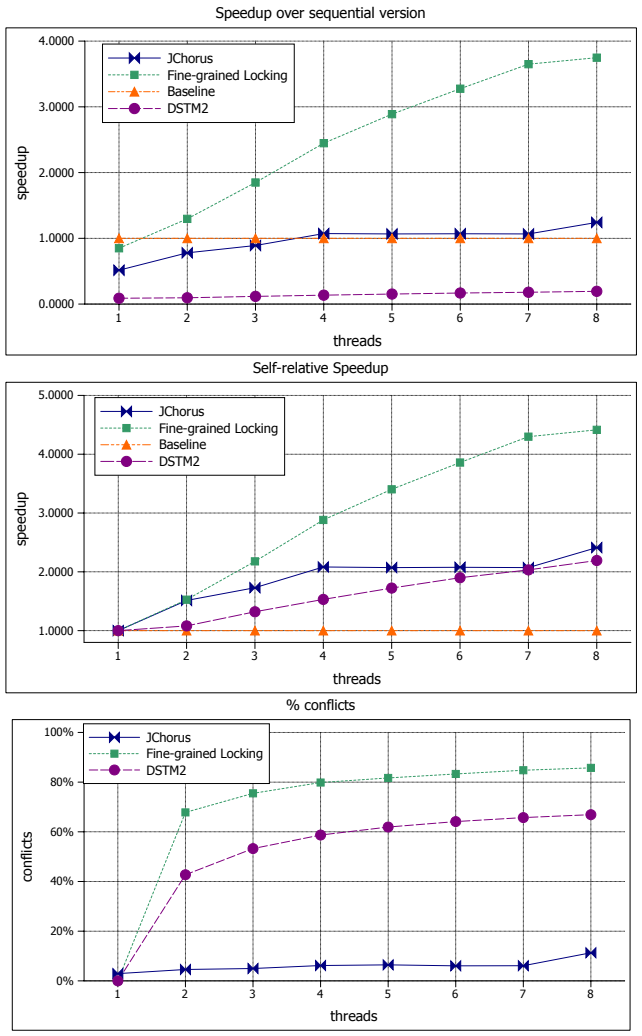

Figure 19. Delaunay Mesh Refinement

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

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

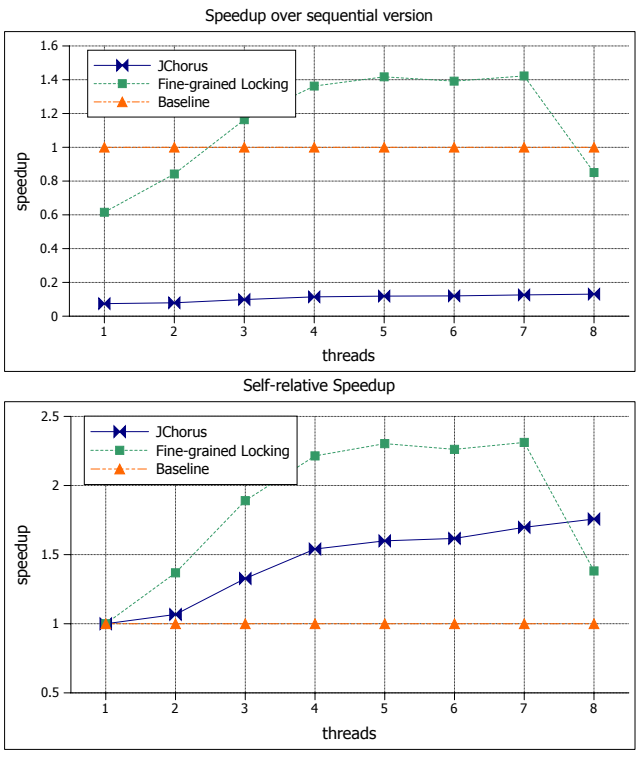

Figure 20. Boruvka's algorithm

the total number of initiated retriangulation tasks (cavities) as the number of threads increased.

6.2 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 US and has 6,262,104 nodes and 15,248,146 edges.

The JChorus implementation used was as in Figure 11. 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 20 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.

7. 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 came out of this project). The solutions there [27, 26] 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 [26]. 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 [24] 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 [30] 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 [30]: "...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 [10, 9] and Chapel [8], 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 [20, 3] 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 [32] 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 [6, 13] and linear types [23]). 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 [18, 19, 28] 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, 26] point out, most implementations 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 [31] and combining them with data-centric synchronization [34]. 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.

8. Conclusion

We have proposed a programming model called Chorus, and a language called JChorus, for expressing and exploiting the locality of effects in irregular parallel applications. The essence of our model is parallelism at the level of assemblies in large, shared data structures; our main concurrency abstraction is an assembly of objects that can split or merge with other assemblies.

As for future work, a more efficient implementation of JChorus is at the top of our agenda. Simultaneously, we want to expand the set of benchmark applications coded using Chorus and release them to the community for feedback on usability. On the language design end, we are currently investigating ways to integrate Chorus with the abstractions of Habanero Java, a cousin of X10 [10].

On the theoretical end, we want to develop a foundational process calculus for Chorus, where processes are assemblies in a graph and able to merge and split. Such a calculus will aim to capture the essence of local concurrent computation in graphs. We would also like to develop a framework for automated reasoning about invariants within and across assemblies. Preliminary investigations on these questions are under way.

Acknowledgements: We thank Vivek Sarkar, Mahmut Kandemir, Zoran Budimlić, Yi Guo, and Milo Martin for valuable discussions and suggestions.

References

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

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

[3] G. Agha, I. Mason, S. Smith, and C. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[4] W. Aiello, Ch. Kalmanek, P. McDaniel, S. Sen, O. Spatscheck, and J. van der Merwe. Analysis of communities of interest in data networks. In *PAM*, pages 83–96, 2005.

[5] J. Barnes and P. Hut. A hierarchical $O(NlogN)$ force-calculation algorithm. *Nature*, 324(4):446–449, December 1986.

[6] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, pages 211–230, 2002.

[7] D. Burke, J. Epstein, D. Cummings, J. Parker, K. Cline, R. Singa, and S. Chakravarty. Individual-based computational modeling of smallpox epidemic control strategies. *Academic Emergency Medicine*, 13(11):1142–1149, 2006.

[8] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *Int. Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[9] S. Chandra, V. Saraswat, V. Sarkar, and R. Bodík. Type inference for locality analysis of distributed data structures. In *PPOPP*, pages 11–22, 2008.

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

[11] P. Chew. Guaranteed-quality mesh generation for curved surfaces. In *Symposium on Computational Geometry*, pages 274–280, 1993.

[12] S. Chung and A. Condon. Parallel implementation of Boruvka's minimum spanning tree algorithm. In *IPPS*, pages 302–308, 1996.

[13] D. Clarke, T. Wrigstad, J. Östlund, and E. Johnsen. Minimal ownership for active objects. In *APLAS*, pages 139–154, 2008.

[14] E. de Sturler and D. Loher. Parallel iterative solvers for irregular sparse matrices in high performance Fortran. *Future Generation Computer Systems*, 13(4-5):315–325, 1998.

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

[16] T. Grune Yanoff. Agent-based models as policy decision tools: The case of smallpox vaccination. Technical report,

Royal Institute of Technology, Sweden.

[17] F. Guidec, P. Calégari, and P. Kuonen. Parallel irregular software for wave propagation simulation. *Future Generation Computer Systems*, 13(4-5):279–289, 1998.

[18] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA*, pages 388–402, 2003.

[19] T. Harris, S. Marlow, S. L. Peyton Jones, and M. Herlihy. Composable memory transactions. In *PPOPP*, pages 48–60, 2005.

[20] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.

[21] K. Hildrum and P. Yu. Focused community discovery. In *ICDM*, pages 641–644, 2005.

[22] D. Jungnickel and M. Swamy. *Graphs, Networks, and Algorithms*. Springer, 2004.

[23] N. Kobayashi, B. Pierce, and D. Turner. Linearity and the pi-calculus. In *POPL*, pages 358–371, 1996.

[24] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Cascaval. How much parallelism is there in irregular applications? In *PPOPP*, pages 3–14, 2009.

[25] M. Kulkarni, M. Burtscher, K. Pingali, and C. Cascaval. Lonestar: A suite of parallel irregular programs. In *ISPASS*, 2009.

[26] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. Chew. Optimistic parallelism benefits from data partitioning. In *ASPLOS*, pages 233–243, 2008.

[27] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, pages 211–222, 2007.

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

[29] E. A. Lee. Are new languages necessary for multicore?, 2007.

[30] K. Pingali, M. Kulkarni, D. Nguyen, M. Burtscher, M. Mendez-Lojo, D. Prountzos, X. Sui, and Z. Zhong. Amorphous data-parallelism in irregular applications. Technical Report TR-09-05, University of Texas at Austin, 2009.

[31] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization techniques for software transactional memory. In *PODC*, 2007.

[32] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for java. In *ECOOP*, pages 104–128, 2008.

[33] H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.

[34] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, pages 334–345, 2006.

[35] M. Wolfe. *High-Performance Compilers for Parallel Computing*. Addison Wesley, 1995.