

# Assemblies of Objects\*

Roberto Lublinerman

Pennsylvania State University  
rluble@psu.edu

Swarat Chaudhuri

Pennsylvania State University  
swarat@cse.psu.edu

Pavol Černý

University of Pennsylvania  
cernyp@cis.upenn.edu

## Abstract

We present a data-centric programming model and a language for irregular, heap-manipulating parallel applications, such as Delaunay mesh refinement or epidemiological simulations. Our aim is to syntactically capture *locality of access* in such applications — the property that accesses to large, global data structures are often restricted to small, contiguous, dynamically determined “neighborhoods.”

## 1. Introduction

Calls for new programming models for parallelism have been heard often of late [13, 15]. It is increasingly clear that the currently popular models of concurrent 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 concurrent programming, of course, poses difficult challenges: for regular array-based codes, static parallelization is known to work. Far more challenging is the problem of efficiently coding applications that combine parallelism with accesses to heap-allocated data structures like lists, trees, and graphs. These are the applications that interest us in this paper. In such *irregularly parallel* applications [9], the typical instance exhibits data-parallelism, but the worst-case instance does not. This makes compile-time parallelization impossible; in fact, it has been noted [11] that most current implementations of *optimistic* parallelism (using software transactional memory) suffer in this setting as well. Vexingly, numerous important applications where parallelism is needed—epidemiological simulations [5], Delaunay mesh refinement [4], *n*-body simulation [3], social network analysis [7], agglomerative clustering [8]—fall in this category.

Consequently, many of these applications are perfect challenge problems for designers of new high-level models of shared-memory parallel programming. This understanding is reflected in the recently released Lonestar benchmarks [2], which offer code and datasets for a number of such problems. In this paper, we present our response to this challenge: a data-centric programming model called *Concurrent Assemblies*, and a concrete language, called *J-Sirius* (for Java Sirius), implementing it.

**Delaunay Mesh Refinement** We motivate Concurrent Assemblies and J-Sirius via the example of Delaunay mesh refinement, a problem of great importance in graphics and scientific computing that is included in the Lonestar benchmarks. Here we are given an initial triangulation of a plane; however, some “bad” triangles may not meet the desired quality constraints. The problem is to retriangulate the mesh so that there is no bad triangle.

It is a property of the problem that such retriangulation affects a “cavity”: a contiguous region in the mesh. Thus, a way to retriangulate would be to maintain a worklist *wl* of bad triangles and, in each

```
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

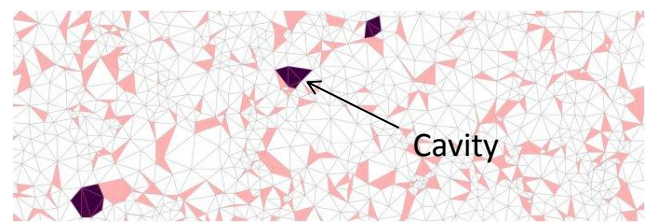


Figure 2. Snapshot of a small part of a Delaunay mesh

iteration, build a cavity *c* consisting of a bad triangle *t* in *wl*, *expand* *c* to the needed extent, and *retriangulate* it. Indeed, this is the basis of a classic sequential algorithm for the problem (pseudocode in Figure 1). While in the worst case, a cavity may encompass the entire mesh, in practice, it almost always is a small neighborhood (see a snapshot of a typical mesh in Figure 2).

**Locality of access** We say that the typical iteration of Delaunay mesh refinement satisfies *locality of access*: the property that an update to a large, global data structure only affects small “neighborhoods” in it.

Locality of access is a feature of numerous other irregular parallel applications that have been previously considered and found to be difficult to parallelize. In fact, it is true of *every application* in the Lonestar suite and in other applications (including an epidemiological simulation and a minimum spanning tree algorithm) that we have considered.

Exploitation of parallelism in applications like the above is directly tied to locality of access. While parallelizing Delaunay refinement, we need to guarantee that cavities—neighborhoods—are retriangulated atomically; cavities that do not overlap can be retriangulated in parallel. Thus, in parallel solutions to the problem, a cavity is the unit of space that a thread needs to access atomically.

### 1.1 Our solution

Concurrent Assemblies frames parallel computations as interactions between assemblies of objects in shared-memory heaps. An *assembly* is a neighborhood equipped with a thread of control. A *neighborhood* is a contiguous region in the shared data structure.

\* The full version of the paper is available [14].

(Note that contiguous here does not refer to memory layout, but rather to the pointer structure).

The only shared objects that assembly can access are the ones belonging to the neighborhood; an assembly can only interact with assemblies that are *adjacent* to it; programming abstractions include a *merge* operation, which merges adjacent assemblies, and a *split* operation, which splits an assembly into smaller ones. All concurrency is captured by assemblies, and there is no separate notion of processes or threads. Our abstractions are race and deadlock-free, and inherently data-centric.

The language J-Sirius embeds a declarative specification of concurrent assemblies into the sequential subset of Java. We demonstrate that it allows natural programming of several important applications exhibiting irregular data-parallelism. A prototype implementation, based on a mapping of assemblies to low-level threads, has competitive performance as well.

The key new technical contribution is that the unit of concurrency is not an individual object, but an *association of objects*—called an assembly—forming a contiguous neighborhood in a shared data structure. Each assembly is equipped with a thread of control, and objects coordinate their updates by organizing themselves into assemblies, which then update themselves.

An assembly can perform three kinds of actions:

- It can read and write objects within it—notably, it cannot access objects within any other assembly, which means objects in assemblies are *isolated*.
- It can *merge* with an *adjacent* assembly to form a new assembly. This is our synchronization primitive.
- It can *split* into a collection of assemblies.

All concurrency in our model is captured with these primitives. The size of the assemblies is a proxy for the granularity of concurrency that the data structure and algorithm permit—the smaller the assemblies, the greater the exploitable parallelism. Of course, an assembly is not required to be bounded; its size depends on the input and the execution history, and it can, in the worst case, encompass the whole heap.

In this paper, we use the Delaunay mesh refinement as a running example. For the other case studies mentioned in the introduction, we refer the reader to [14].

## 1.2 Related Work

No language for shared-memory parallelism that we are aware of is designed with locality of access as a first principle—or, for that matter, can express locality of access in a general way. In the 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, assemblies. This global nature of shared-memory accesses has a negative effect on programmability as well as performance.

**Software transactions.** In languages using non-blocking software transactions [12], the burden of reasoning about global accesses is passed to the transaction manager—as a result, 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. [11, 10] point out, this makes them behave inefficiently while handling large, irregularly parallel heap-manipulating programs.

**Galois project.** The Galois project [11, 10], which also aims to parallelize irregular, data-intensive applications (indeed, the Lonestar benchmarks came out of this project). While these papers were

```

Prog      ::= [Assembly]+
Assembly ::= assem  $\tau$  :: Ldec Act
Ldec     ::= init  $v_{in}$  var [ $v$ ]*
Act      ::= action :: [Guard : Update]*
Guard    ::= merge( $v.f$ ) | merge( $v.f$ ) when Bexp |
             merge( $v.f, \tau, u$ ) |
             merge( $v.f, \tau, u$ ) when Bexp | Bexp
Update   ::=  $v :=$  Exp |  $v.f :=$  Exp | Update; Update |
             skip | split( $\tau$ ) | splitone( $v, \tau$ )
Exp      ::=  $v$  | Exp.f | error
Bexp     ::= Exp op Exp | not Bexp | Bexp and Bexp

```

where

$v, v_{in} \in Var$ ,  $\tau \in \mathcal{T}$ , and  $f \in F$ , and  $[t]^*$  denotes zero or more occurrences of  $t$

Figure 3. Syntax of Core Sirius

inspirations for our work, they do not have a language design where graph neighborhoods actively drive a parallel computation.

**Actors.** The Actor model [6] was one of the earliest data-centric models of concurrency. Actors are reactive rather than active entities, so that our assemblies are better seen as threads rather than actors. While it is possible to define actor-based analogs of assemblies, so far as we are aware, no existing work using actors does so, or has primitives like our merge and split operations.

## 2. Concurrent Assemblies

Now we present the main features of the Concurrent Assemblies programming model. This is done using an abstract definition of shared-memory heaps, and a core programming language called *Core Sirius* that elides all but the most essential aspects of J-Sirius (The details of the full language can be found in [14]).

The central entity in Concurrent Assemblies is the shared-memory *heap*, which maintains the state of all shared data accessed by a parallel program. The heap is modeled by an edge-labeled, weakly connected, directed graph  $G$ . 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 weakly connected subgraph of  $G$ .

For example, the datasets in the Lonestar benchmarks for Delaunay mesh refinement model the mesh as a graph where nodes are triangles, each triangle having edges to its neighboring triangles. In our formulation, each triangle is an object, and every cavity is a region in the complete heap.

### 2.1 Syntax

Our most significant departure from traditional shared-memory models is that parallelism in our model is driven by assemblies: regions in a heap equipped with sequential threads of control. All concurrency is captured by assemblies, and there is no separate notion of threads or processes.

In Core Sirius, assemblies do not call methods on objects or create new objects, and do not have branches and loops. This is because it only addresses the *concurrent* aspects of J-Sirius, whereas branches, loops, etc. belong to the sequential code executed by assemblies. For the same reason, 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 contain 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.

Let us assume a universe  $Var$  of *assembly variables*, and a universe  $\mathcal{T}$  of *assembly classes* that contains a designated *initial class*  $\iota$ . The syntax of programs in Core Sirius is given in Figure 3. Here:

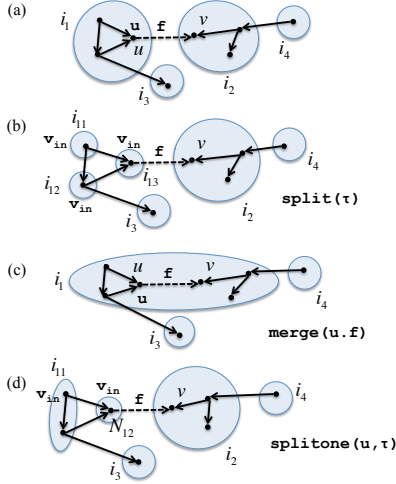


Figure 4. Merging and splitting

(i) *Prog* represents programs, *Assembly* represents declarations of assembly classes; (ii) *Act* represents actions of assemblies, each action being a sequence of guarded updates; (iii) *Ldec* represents variable declarations. The variable declared by the keyword *init* is the initial variable; (iv) *Guard* represents guards, and *Update* represents updates; (v) *Exp* and *Bexp* respectively represent pointer expressions and boolean expressions.

## 2.2 Informal semantics

The active behavior of assemblies of class  $\tau$  is defined by the action  $Act(\tau)$ , which is a set of guarded updates. Each assembly  $i$  repeatedly performs the following three steps: (1) nondeterministically choose a guarded update, (2) atomically evaluate its guard, (3) if the guard is enabled, execute the update (as  $i$  has exclusive access to its region, no extra precaution to ensure the atomicity of the update is needed).

Unlike in other guarded-command languages, a guard here can, in addition to checking local boolean conditions, *merge* the assembly  $i_1$  in question with an adjacent assembly  $i_2$ , changing the state of  $i_1$  and causing  $i_2$  to terminate. In its new state,  $i_1$  operates on the union of the regions previously accessible to  $i_1$  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 not busy—i.e.,  $i_2$  cannot be midway through executing an update. Thus, a merge is a synchronization operation. Figures 4-(a) and 4-(c) show the states of a parallel program before and after the assembly  $i_1$  merges with  $i_2$ .

The two non-standard types of updates are for *splitting* a assembly  $i$  of class  $\tau$ . Of these, the update  $split(\tau')$  splits  $i$  at the finest possible granularity—for each object  $u$  in the region of  $i$ , a new neighborhood, with a globally unique, fresh ID  $i_u$  and a region comprising the single node  $u$ , is created and activated. Each  $i_u$  is of class  $\tau'$ . The neighborhood  $i$  ceases to exist. As with merges, the heap itself is not modified. Figure 4-(b) shows the assemblies  $i_{11}$ ,  $i_{12}$ , and  $i_{13}$  resulting from splitting  $i_1$  in Figure 4-(a).

The  $splitone(u, \tau')$  update, intuitively, splits off one node, which will be a new assembly of class  $\tau$ , and performs the necessary maintenance of the remainder of the original assembly. Figure 4-(d) shows the assemblies  $i_{11}$  and  $i_{12}$  resulting from splitting  $i_1$ , in Figure 4-(a), in this way.

Other kinds of updates let an assembly  $i$  perform imperative modification of its own region. As the computation is located within the region, any expression whose evaluation requires accesses outside returns an error value `error`. Note that updates to local pointers can cause the region to become non-contiguous. Consequently,

```

neighborhood Triangle:: ...
  action::
    merge (v.f, Cavity, u) when bad?: skip

neighborhood Cavity:: ...
  action::
    merge (v.f) when (not complete?): skip
  complete?:
    retriangulate(); split(Triangle)

```

Figure 5. The essence of Delaunay Mesh Refinement in Core-Sirius-style code

we define the region of  $i$  after a pointer update to be the maximal region containing the object referenced by the initial variable of  $i$ . The rest of the region is split into inactive, singleton assemblies.

## 2.3 Case study: Delaunay mesh refinement

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<sup>1</sup>: First, a point  $p$  at the center of the circumcircle of the triangle is inserted into the mesh. Second, 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. Third, the cavity is then retriangulated by connecting  $p$  with all the points at the boundary of the cavity (this is done by `c.retriangulate()`).

We have parallelized this application using J-Sirius. We use Core-Sirius-style code to capture its essence. 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 “bad?” that, at any point, is *true* iff it is bad, and that a `Cavity` can use a boolean variable “complete?” that, at any point, is *true* iff it needs no further expansion. Finally, we let each `Triangle` and `Cavity`  $i$  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 leave out the code maintaining these variables.

The code for our modeling is given in Figure 5. Here, each triangle (forming an assembly of class `Triangle`) checks if it is “bad.” If it is, then it merges with an arbitrary neighbor to create an assembly of class `Cavity`. Cavity expansion is captured by the merge-operation in the `Cavity` class. 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.

## 2.4 Race-freedom, deadlock-freedom, and liveness

A *data race* happens when two concurrent threads concurrently access a shared object, and at least one of these accesses is a write. Updates in J-Sirius are trivially data-race-free, as they operate on disjoint regions in the heap. As for merges, the semantics guarantees that an assembly  $i$  can merge only with assemblies in states is

<sup>1</sup> For ease of presentation, we suppose here that the bad triangle is not near the boundary of the whole mesh.

not inside an update. Thus, we have that programs in J-Sirius are free of data races.

As for 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. One way to adapt this definition to our setting is the following: “A deadlock arises when an assembly  $i_1$  has a locally enabled merge along an edge into assembly  $i_2$ , and vice-versa. 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, this scenario does not prevent progress in our setting. Our operational semantics nondeterministically executes one of the requested merges (say the one invoked by  $i_1$ ), causing  $i_2$  to terminate, and bringing the requested object in the possession of  $i_1$ . Perhaps a progress requirement more appropriate in our setting is *response*: an assembly with an enabled merge-guard must eventually be able to do so, unless it terminates first. Guaranteeing this requires language specification at a level lower than our operational semantics. However, under a certain natural requirement on the runtime system, it is possible to show that responsiveness in the above sense holds.

### 3. Implementation and evaluation

The language J-Sirius embeds a declarative specification of concurrent assemblies into the sequential subset of Java. We have implemented a compiler and a runtime system for the language. The compiler translates J-Sirius into a Java program running on our system.

The abstraction that our model offers to the programmer is that of neighborhood-level parallelism in large data structures. To get a sense of the level of parallelism that may result from this, consider again the sequential Delaunay mesh refinement algorithm. Running this algorithm on a mesh of over 100,000 triangles from the Lonestar benchmarks many times, we found the average cavity size to be only 3.75, and the maximum cavity size to be 12.

Due to this massive parallelism, any runtime system for J-Sirius has to perform a many-to-one mapping of assemblies to the limited number of physical threads possible in a system. We call a set of assemblies mapped to one physical threads a *division*. A remote merge, i.e. a merge between assemblies in different divisions causes migration of assemblies. A (parametrized) heuristic is applied to reduce the number of remote merges and to maintain load-balancing. A simple token-passing strategy is used to prevent deadlocks. We refer the reader to the full version [14] for the details on the compiler and the runtime system.

Using the Delaunay mesh refinement application from the Lonestar benchmarks, we have performed a preliminary evaluation of the performance of our approach. We compared the performance with:

- Baseline: The sequential Java version of the application distributed with the Lonestar benchmarks.
- Fine-grained locking: a version written using fine-grained locks.
- DSTM2: another using the software transaction library DSTM 2.1.

We wanted to compare the performance of our system with that of a well-established system for high-level parallel programming—the constraints were that it had to be Java-based and publicly available. We settled on DSTM2, a state-of-the-art library of software transactions supporting dynamic-sized data structures [1]. We compared performance of the multiple transaction management strategies available in DSTM2 and show the results for the best one.

The experiments were run on an 8-core (dual quad-core) Intel Xeon E5405 system with 2.0 GHz processor speed per core, 8 GB RAM, and a 64-bit Linux operating system. Figure 6 shows the ratio of the performance a system to that of the baseline sequential implementation. For more extensive evaluation, as well as for experimental results on other applications, see [14].

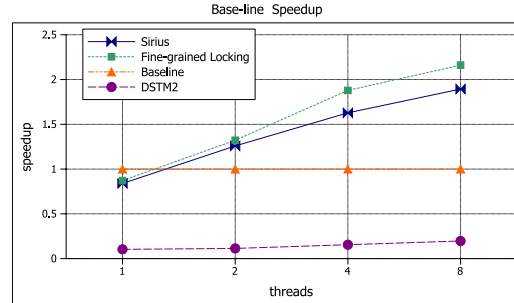


Figure 6. Base-line speedup.

### 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] J. Barnes and P. Hut. A hierarchical  $O(N \log N)$  force-calculation algorithm. *Nature*, 324(4):446–449, December 1986.
- [4] P. Chew. Guaranteed-quality mesh generation for curved surfaces. In *Symposium on Computational Geometry*, pages 274–280, 1993.
- [5] T. Grune Yanoff. Agent-based models as policy decision tools: The case of smallpox vaccination. Technical report, Royal Institute of Technology, Sweden.
- [6] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.
- [7] K. Hildrum and P. Yu. Focused community discovery. In *ICDM*, pages 641–644, 2005.
- [8] A. Jain and R. Dubes. *Algorithms for Clustering Data*. Prentice Hall, 1988.
- [9] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval. How much parallelism is there in irregular applications? In *PPOPP*, pages 3–14, 2009.
- [10] 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.
- [11] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, pages 211–222, 2007.
- [12] J. Larus and C. Kozyrakis. Transactional memory. *Communications of the ACM*, 51(7), 2008.
- [13] E. A. Lee. Are new languages necessary for multicore?, 2007.
- [14] R. Lublinerman, S. Chaudhuri, and P. Černý. Parallel programming with object assemblies. In *OOPSLA*, 2009, to appear.
- [15] H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.