
Proceedings Paper:
Duke, DJ and Hosseini, F (2015) Skeletons for Distributed Topological Computation. In: Rompf, T and Mainland, G, (eds.) FHPC 2015 Proceedings of the 4th ACM SIGPLAN Workshop on Functional High-Performance Computing. Functional High Performance Computing, 03 Sep 2015, Vancouver, Canada. ACM Press , pp. 35-44. ISBN 978-1-4503-3807-3

https://doi.org/10.1145/2808091.2808095

Skeletons for Distributed Topological Computation

David J. Duke Fouzhan Hosseini
School of Computing, University of Leeds, Leeds, UK
{D.J.Duke,F.Hosseini}@leeds.ac.uk

Abstract

Parallel implementation of topological algorithms is highly desirable, but the challenges, from reconstructing algorithms around independent threads through to runtime load balancing, have proven to be formidable. This problem, made all the more acute by the diversity of hardware platforms, has led to new kinds of implementation platform for computational science, with sophisticated runtime systems managing and coordinating large threadcounts to keep processing elements heavily utilized. While simpler and more portable than direct management of threads, these approaches still entangle program logic with resource management. Similar kinds of highly parallel runtime system have also been developed for *functional* languages. Here, however, language support for higher-order functions allows a cleaner separation between the algorithm and 'skeletons' that express generic patterns of parallel computation. We report results on using this technique to develop a distributed version of the Joint Contour Net, a generalization of the Contour Tree to multifields. We present performance comparisons against a recent Haskell implementation using shared-memory parallelism, and initial work on a skeleton for distributed memory implementation that utilizes an innovative strategy to reduce inter-process communication overheads.

Categories and Subject Descriptors D. Software [*D.1 Programming Techniques*]: D.1.1 Applicative (Functional) Programming

Keywords Computational Topology, Performance, Eden, Haskell

1. Introduction

Computational science has a long history of driving advances in computing, from the development FORTRAN by Backus in the 1950s, through development of parallelism frameworks such as OpenMP and MPI, to the emerging challenge of exa-scale computing. Our work is situated at the intersection of three recent advances that have emerged in response to the continuing challenge of scale:

- *Topological analysis* to provide compact representations of large datasets, and providing scaffolding for exploration and analysis. This paper is concerned with one such technique, the Joint Contour Net (JCN), a recent addition to the topological toolbox that approximates the Reeb space for *multifield* data.
- *Parallel runtime systems* (RTS) that provide a level of abstraction above disparate parallel hardware platforms, facilitating resource management and inter-thread coordination. Well-known examples in computational science include Charm++ [19] and UINTAH [25] in scientific computing, and VTK [33] in pipelined visualization.
- *Higher-level languages* that reduce the programming effort needed to create and manage parallelism, with or without RTS support. Domain-Specific Languages (DSLs) are a recent trend in this area, with examples including ViSlang [32] for parallel volume rendering, and Liszt [8] for portable PDE solvers.

Rather than start from a *special-purpose* runtime system and language, our work seeks to exploit advances elsewhere in the programming system design space: parallel *functional* programming, and specifically Haskell [28] and its distributed variant Eden [22]. Current Haskell compiler and RTS implementations have many of the characteristics described above. For example, GHC[1] has an advanced multi-core runtime system with very lightweight threads, supporting work-stealing and coordination, while the Eden compiler implements language extensions for distributed processes via MPI-based coordination similar to that underlying e.g. Charm++.

Previous work [12] reported on the utility of Haskell in implementing the JCN algorithm, and results on shared-memory parallelization using hand-crafted strategies for work allocation. Starting from that baseline, this paper makes three contributions:

1. Describes and evaluates a *distributed* memory implementation of the JCN algorithm in Eden, an essential step towards analysis of larger-scale datasets such as hurricane simulation [16, 18].
2. Investigates the use of generic skeletons to simplify the task of parallelization, and identifies limits to this approach linked to properties of the application domain.
3. Implements a novel approach to computing the JCN as a distributed data structure, setting the stage for a new class of skeletons applicable to a range of computational science tasks.

Our application domain, multifield topological analysis for computational science, is introduced in Section 2; we also summarise progress to date on parallelization using conventional languages. We assume that the reader is familiar with functional programming and simple features of Haskell (data type definitions, higher order functions, and type classes), but in Section 3 we provide an introduction to Eden's approach to distributed parallel programming. Section 4 describes the distributed-memory implementation, and examines the use and performance of different high-level skeletons to implement parallelization. A key finding is the high cost imposed by inter-node communication. In section 6 we address this by developing a new skeleton that reduces communication costs by using a technique of boundary updates. We

[1] Glasgow Haskell Compiler

conclude the paper by reflecting on the tension between generic 'reusable' skeletons and the need in high-performance computing to exploit domain-specific knowledge.

2. Background

Computational science involves the synthesis, collection and analysis of data sampled over some domain, where both the sampled phenomena and underlying domain are continuous spaces. Applications are diverse and numerous, and the insight from these can have profound implications for social policy, technological development, or fundamental scientific insight. Examples include

- meteorological and climate models, at both global level and of isolated phenomena such as storms;
- geophysical models of petroleum reservoirs, phenomena such as the earth's magnetic field, or events such as earthquakes;
- models of molecular structure, ranging in detail from quantum-level studies of simple interactions through to macro-level models of biomolecular structure and function, for example of the HIV virus [39];
- studies of physical materials and interaction, from the low-level physical structure of materials for carbon capture through to complex interactions in combustion and explosions.

These data are usually defined over a spatial domain that is homeomorphic to a subset of R^2 or R^3. In many cases the data is part of a time series in which case time can be treated as a further dimension, allowing for example, the interpolation of models between discrete simulation steps [2]. In general we consider the phenomena of interest to be finite samples taken from the continuous function $f : R^{m(+1)} \rightarrow R^n$, where m is the spatial domain dimension. The range (R^n) of the function captures the properties being modeled or observed, including scalars (e.g. temperature, pressure), vectors (flow), and/or tensors (e.g. diffusion). In practice it is usual to separate the individual fields of interest. For example, a hurricane simulation studying three scalar fields (temperature, pressure, and precipitation) plus a single vector field (wind velocity) over a 3D time series corresponds to a function $f : R^4 \rightarrow R^7$, but would also be considered as four functions defined over a common domain, *temp*, *pres*, *precip* : $R^4 \rightarrow R$ and *vel* : $R^4 \rightarrow R^3$.

In the 1980s the need to analyse (relatively) large volumes of data from computer simulation and imaging devices led to the emergence of scientific visualization [24]. This combines high-performance computing and graphics to generate imagery that affords insight into the data, and has led to a suite of widely used visualization techniques such as isosurfacing and volume rendering for scalar fields, and streamlines, streamsurfaces, and LIC for vector fields [33, 38]. However the challenge facing visualizaion (and computational science generally) is that of scale: increasing levels of ambition from domain experts, and the need for finer levels of resolution to develop further insight into physical phenomena. Dataset scale challenges visualization (i) through the computational cost of generating visual representations, and (ii) through the mis-match between the volume of data to be represented, and the capabilities of the human perceptual system. Topological abstractions appear the most promising solution to date, but still present significant mathematical and computational challenges.

2.1 Computational Topology

The value of an abstraction lies in its ability to reduce the volume of data while preserving important details in some context. One of the most important details when understanding the behaviour of continuous functions are the set of *critical points*, and relationships between these points. Topological abstractions have been successful precisely because they capture these points and their relationships within a rigorous mathematical foundation that permits further analysis. For scalar fields, the critical points consist of minima and maxima (local and global), and saddle points. Two scalar topological abstractions that partition the domain into regions defined in terms of critical points are the Reeb graph, and the Morse-Smale complex.

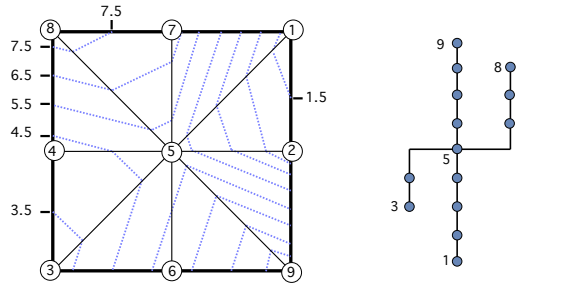


Figure 1. A scalar field defined over a 2D domain, showing (left) a set of contour lines, and (right) the contour tree.

1. The Reeb graph captures the nesting relationship between contours of the function (connected components of the level sets). Where the function is defined over a simple manifold, which is often the case in computational science, the Reeb graph is an unrooted tree and is called the Contour Tree.
2. The Morse-Smale complex partitions the domain into regions based on gradient. The gradient of all points within a given Morse-Smale region directs flow to a common sink, and dually receives flow from a common source.

We will not expand further on the Morse-Smale complex, nor do we discuss the considerable body of work on vector field topology, or emerging results on tensor topology. Instead, as the results presented in this paper concern a generalisation of the Contour Tree we illustrate the latter via a small example in Figure 1. Each circle in the figure represents a point, consisting of a location in 2D space and a single scalar value, written inside. For the sake of illustration, these could be considered as height above sealevel arising in a climate change simulation.

As we have only discrete samples, if we want to find the height between samples we need a means of *interpolation*. This requires partitioning the spatial domain into local regions called *cells*. There are different schemes for partitioning and interpolation with known trade-offs [6]; in this paper we use decomposition into simplicial complexes (triangle meshes in 2D, tetrahedral meshes in 3D) and then barycentric interpolation within each simplex.

2.2 The Joint Contour Net

Scientists and engineers can rarely work with a single phenomenon in isolation. Most computational studies involve multiple scalar fields (*multifields*), and progress often comes from qualitative and quantitative insight into the complex interactions between the fields. Examples of recent multifield studies include proton & neutron density in nuclear physics [11, 34, 35], gas temperature, particle density and turbulence in astrophysics [10], and temperature, pressure and precipitation in meteorology [16]. Our discussion of multifield techniques is limited to multiple scalar fields; analysis of interactions between scalar and vector/tensor fields is an open research problem.

Unfortunately, finding topological abstractions for even the 'simple' case of scalar multifields has proved challenging. Work to date has included the concept of the Jacobi Set [13], and of the Reeb space [14] which generalises the Reeb graph for single scalar

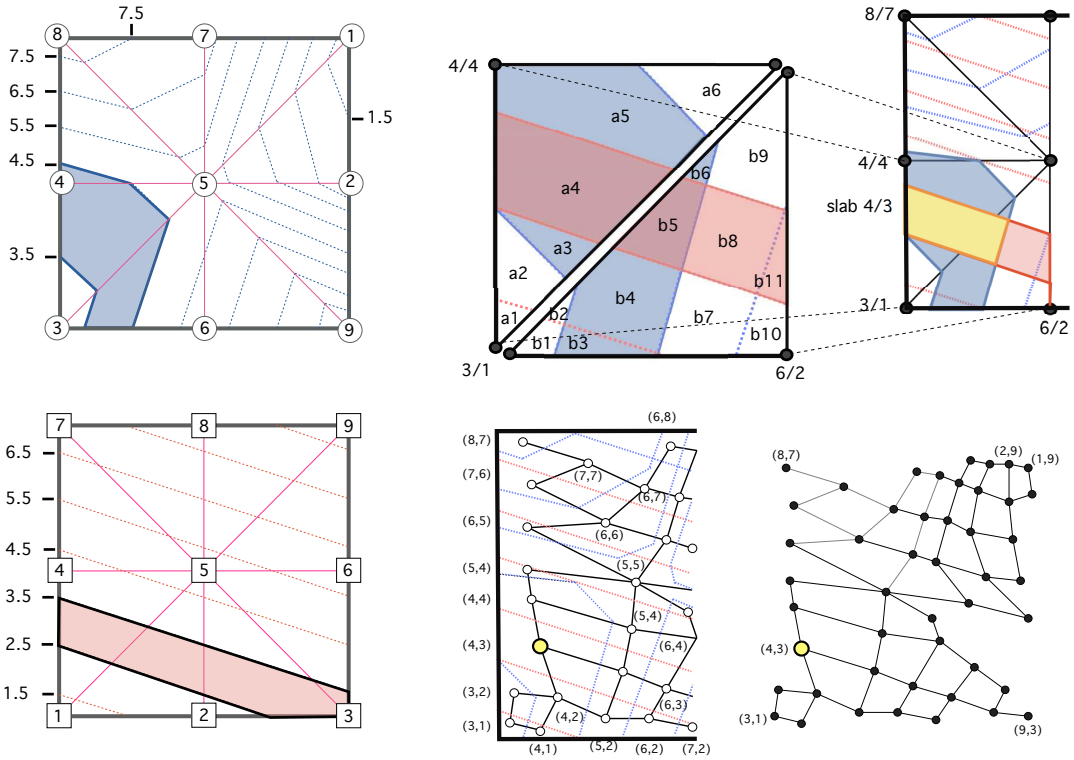


Figure 2. Construction of the Joint Contour Net for a 2-field dataset defined on a 2-dimensional domain, i.e. $R^2 \rightarrow R^2$, shown on the left.

fields to a set of fields. However, there are few computationally effective tools for analysis of the exact Reeb space generated from a set of samples defined over a mesh.

The Joint Contour Net [5] is an *approximation* to the Reeb space, constructed by first quantising the range of the function in each of the n dimensions, and then constructing a graph representing adjacency in the spatial domain between the equivalence classes ('slabs') induced by this quantisation. For each range dimension $1 \leqslant i \leqslant n$, the desired level of quantisation in dimension i is expressed as a *slab width* w_i. Given a simplicial mesh with samples at the vertices, the algorithm can be expressed in terms of two phases:

fragmentation: The simplicial cells are subdivided into polytope fragments by families of parallel cutting planes, one family per range dimension, where the distance between planes is defined by the slabwidth of the respective dimension. At the end of the phase, each cell's fragments form a partition of that cell, and collectively the fragments partition the original domain. Each fragment is assigned a quantised coordinate in range space, defined by the minimum point on the polytope.

merger: No two fragments derived from any one simplicial cell will have the same quantised coordinate, but adjacent fragments from neighbouring cells may do. Such equivalent fragment polytopes are merged to produce slabs, which extend across multiple cells and partition the domain into regions that are equivalence classes of the quantised range values.

The Joint Contour Net J is the dual graph of the slab regions. It has one node per slab, with an edge (u, v) in J just when the slabs corresponding to u and v are adjacent in the domain.

Figure 2 illustrates the approach using a dataset consisting of two fields, 9 points, and 8 simplicial cells (triangles). Both fields are quantised to a slab width of 1. The upper right of the figure focuses on just two of the cells. Polygonal regions defined by the cell boundaries and cutting planes define the first-phase fragments: 6 (a1-a6) in one cell, and 11 (b1-b11) in the second. Within a cell each fragment is uniquely defined by its field values, for example fragment 'a4' lies in the inverse image of the half-open interval [3.5-4.5) for the first field, and [2.5,3.5) for the second, so would be assigned the range coordinate (3.5, 2.5). The upper-right of the figure shows the result of the merger phase. Two adjacent fragments defined over the same interval, a4 and b5, have been merged, producing the slab shown in yellow. The lower part of the figure contains the graph of slab adjacency, with the newly-merged slab marked by a yellow node. The JCN for the full dataset is shown in the bottom-right. To date, implementation of the JCN algorithm has used two intermediate structures [5]:

1. a *disjoint set* data structure, initialised to the set of fragments by the first phase, and which then captures the union of equivalent spatially adjacent fragments to eventually form the slabs; and
2. a graph G whose nodes are initially the fragments, with an edge between two nodes iff the corresponding fragments are adjacent but differ in one or more field intervals. Merger of fragments in the spatial domain triggers merger of the corresponding nodes in the graph; at completion of the second phase, G has been reduced to the JCN graph J.

2.3 Parallel Computation of Topology

Given the challenge that scale poses across computational science it is unsurprising that there has already been considerable interest in exploiting parallel computing for constructing topological abstractions. Ironically, the very property that makes topology useful (that it provides a *global* summary of data) has complicated development of parallel algorithms, which want to operate on *local* subsets of the domain.

To date, successful approaches to parallel computational topology fall into two broad groups. Examples of the first group include parallel computation of the Contour tree [29] and more recently the Morse-Smale complex [17, 36, 37]. These implementations use variants of the sequential algorithms that are able to compute local fragments of the topological data structure across subdivisions of the domain, and then employ a reduction strategy to generate the global abstraction. Beyond the complexity of the algorithms, the 'cost' is the additional data needed to perform reduction across neighbouring regions; this can be significant, and has been a major obstacle to greater parallel scalability. However recent work has led to a new family of strategies that avoid generation of a global abstraction, either

1. by using a distributed data structure [21, 26, 27], and/or
2. by terminating reduction once a desired level of refinement has been reached [17, 21].

These strategies work when the topological abstraction is used to answer queries, e.g. on the cells intersected by isosurfaces, or where there is a rigorous strategy for simplification. Compared to the Contour tree and Morse-Smale complex, development of the JCN is still at an early stage, and current applications are limited to using the full structure for feature analysis, see for example [34]. Ongoing work [7] may deliver a formal basis for simplification/refinement of the JCN, but for the present paper our strategy is to implement reduction to a global full-resolution graph that can be visualized to gain insight into interaction between fields. However, in later sections of the paper we will return to the strategy of building distributed data structures.

3. Distributed Functional Programming

While pure functions are side-effect free, high-performance parallel and distributed programming requires some ability to control the allocation of work to processing elements. This management can be handled in two ways: through the underlying runtime system, and/or through abstractions (monads) that 'sandbox' effectful computation. Both routes have been exploited in Haskell implementations. The GHC compiler includes a primitive operation, *par* for sparking speculative threads in the runtime system, and a further primitive, *seq* for controlling when evaluation of an expression actually takes place.

On top of this low-level substrate, a number of libraries provide combinators that capture common patterns of processing. The 'Par' monad [23], for example, implements a deterministic model of shared-memory parallelism on top of the GHC's support for multiple execution contexts, with the standard *bind* and *return* operators implementing suspension and resumption via continuation-passing. With the addition of write-once mutable references (IVars), this minimal set of forms underpins a rich set of derived operators, from simple thread forking through to generic divide-and-conquer and stream processing strategies. This framework was used in [12] to develop a parallel shared-memory implementation of the JCN.

Although GHC/Haskell has monadic libraries that support distributed applications, *efficient* distributed processing currently requires support at the RTS level. Eden [22] is a dialect of Haskell, implemented as a branch of GHC tailored to distributed-memory architectures. In place of the low-level 'par' operator for sparking threads, Eden provides a *process* abstraction and methods for remote process instantiation. The abstraction, *process* in Figure 3, converts a (pure) function of type $a \rightarrow b$ into a process of type $Process\ a\ b$ while the process instantiation operator, #, launches a process abstraction as a child process, potentially on a remote node, creates communication channels between the processes, sends inputs to the child process, and retrieves the output once the remote function has completed evaluation. The *type context* $(Trans\ a, Trans\ b)$ in Figure 3, indicates that types a and b must belong to $Trans$ type class, i.e. they are values that can be transmitted via channels.

```
process :: (Trans a, Trans b) ⇒ (a → b) → Process a b
(#) :: (Trans a, Trans b) ⇒ Process a b → a → b
class NFData a ⇒ Trans a where
  write      :: a → IO ()
  createComm :: IO (ChanName a, a)
parMap :: (Trans a, Trans b) ⇒ (a → b) → [a] → [b]
```

Figure 3. Eden constructs, and a simple parallel skeleton.

Eden extends the functionality of the GHC runtime system with primitive operations that perform basic tasks such as creating a process on another processor, creating communication channels, and sending data over channels. These primitive operations are then used to implement the higher-level Eden constructs such as *process*, #, and the *Trans* type class.

While task placement, synchronisation, and data communication are handled by the parallel RTS, Eden also provides a rich set of parallel *skeleton* libraries. In analogy to the monadic interfaces for shared-memory parallelism in GHC, these capture common patterns of *distributed* computation. For example, the 'map reduce' paradigm can be expressed by the following small block of code, the first two lines expressing map/reduction on a local node, and the remainder specifying distribution across distributed nodes, where *noPe* is a built-in primitive giving the number of available processing elements [22].

```
mapRedr :: (b → c → c) → c → (a → b) → [a] → c
mapRedr g e f = (foldr g e) ∘ (map f)
parMapRedr :: (Trans a, Trans b)
              ⇒ (b → b → b)
              → b → (a → b)
              → [a] → b
parMapRedr g e f
  = if noPe ≡ 1 then mapRedr g e f
    else (foldr g e)
         ∘ (parMap (mapRedr g e f))
         ∘ (splitIntoN noPe)
```

During process creation static channels are created between parent and child processes; hence they can only form a hierarchical process topology. Although this is the default data communication model, direct communication between child processes is possible via dynamic channels; see [4, 22] for details.

4. Distributed Implementation of the JCN

Compared with other topological abstractions, parallelization of the JCN is straightforward, and a parallel implementation of the JCN on shared memory has already been reported [12]. In overview, an implementation consists of the following three phases:

1. Divide the given dataset into a set of mutually disjoint subsets.
2. Compute the JCN for each subset, using a sequential implementation of the algorithm.
3. Merge the JCNs of subsets into a global JCN. As the original algorithm includes a 'merge' phase in which fragments from one cell are added to the evolving JCN, no other information needs to be carried forward for parallel implementation.

```
data JCN d r = JCN { slabs   :: [(Int, r)]    -- Mapping from slab ids to quantized range coordinates
                   , edges   :: [(Int, Int)]  -- Edges between slabs
                   , border  :: [(Int, d)]    -- Domain coordinate of slab facet centers
                   , slabNo  :: Int }         -- Number of nodes in this JCN
```

Figure 4. JCN data representation.

Three issues have driven our distributed implementation:

1. Datasets in computational science often simply exceed the capacity of shared memory machines, and in practice there is a need for implementations that can run on distributed memory machines from modest clusters to supercomputers.
2. Even for small datasets, peak memory and churn were already affecting performance of the shared memory implementation. Distribution would allow further investigation into scalability.
3. Although parallelisation on shared memory was built on the *Par* monad, extensive use was made of custom strategies. One of the claimed benefits of higher-order programming is the ability to capture general patterns of computation, and for distributed Haskell, *skeleton* libraries appear to offer a useful starting point for parallelisation.

Given point (3), we began our implementation by exploring two of the generic strategies from the Eden skeleton library, *divide and conquer* and *workpool*. However, before reporting this and subsequent work, we first address the Haskell representation of the JCN, and the general issue of IO in a distributed setting.

4.1 Data Representation

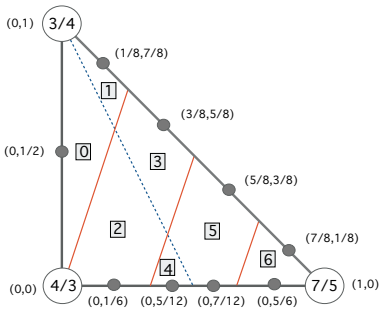
Figures 4 and 5 show, respectively, the data type used to represent the JCN within nodes, and an instance of this data type for the fragments generated from one of the initial cells in Figure 2. The 'border' field plays an important role in later developments. When merging the JCNs of two regions, we need to determine which slabs in the two regions are adjacent. By construction, polytope facets on the boundaries of adjacent slabs will be identical: in the two-dimensional case, the polytopes are the polygons, and their facets are polygon edges. Determining adjacency between polytopes is tricky; we can't for example use vertex ids because the polytopes have been generated by functions operating independently on the different subdomains that happen to share a face. Our solution exploits a geometric invariant: in a spatial partition, two polytopes are adjacent iff each has a facet with same center. During the fragmentation phase we compute the center of each facet, and then use this during reduction to identify adjacent facets. Adjacent facets are either merged (if they have the same range coordinate), or contribute an edge to the JCN (if they don't), so at each stage the only facet centers that are carried forward as part of the JCN are those that lie on the boundary of the (sub)domain.

4.2 Input/Output

There are two extreme choices in providing data to parallel tasks:

1. the parent reads data and transfers it to child processes, or
2. the child process is provided with a handle to the file(s) and an index to determine the subset of the data that it requires.

Even on modest datasets, preliminary experiments showed that the former approach generates significant communications overhead, and would become untenable on larger data. Consequently, we have taken option (2) in our implementations. Input files are stored in a Lustre parallel filesystem, with one file per field in the dataset. In following sections, references to 'dividing the dataset', should be understood as computing the bounds and indices of subdomains, rather than subdivision of the data per se. This is an obvious strategy, and one widely adopted in high performance computational science, so as we will discuss in the conclusion it was surprising that it was not well supported by distributed skeletons.




```
slabs  = [(0, (3, 3)), (1, (3, 4)), (2, (4, 3)), (3, (4, 4)), (4, (5, 3))
         , (5, (5, 4)), (6, (6, 4))]
edges  = [(0, 1), (0, 2), (2, 3), (1, 3), (2, 4), (3, 5), (4, 5), (5, 6)]
border = [(0, (0, 1 / 2)),      (1, (1 / 8, 7 / 8)), (2, (0, 1 / 6))
         , (3, (3 / 8, 5 / 8)), (4, (0, 5 / 12)),    (5, (0, 7 / 12))
         , (5, (5 / 8, 3 / 8)), (6, (7 / 8, 1 / 8)), (6, (0, 5 / 6))]
slabNo = 6
```

Figure 5. Data representation of a JCN.

4.3 Divide and Conquer (DC)

Our first implementation used the divide-and-conquer *disDC* skeleton from the Eden skeleton library[15]. The idea is to split the input into a set of sub-problems, recursively compute the results of each in parallel, and then merge these into a single solution. Recursion terminates when a problem can be solved directly, or when the *ticket list* used to manage allocation of work to processors is empty. The skeleton assumes that problems are always split into the same number of sub-problems, called the branching degree. If processor i divides a problem into n subproblems $[p_{i1}, p_{i2}..p_{in}]$, processor i continues with computation of p_{i1} and the remaining problems are distributed across the processors available in the ticket list. The skeleton has the following signature:

```
disDC :: (Trans a, Trans b)
  ⇒ Int                  -- the branching degree
  → Places               -- ticket list
  → (a → Bool)           -- is problem directly solvable?
  → (a → b)              -- solve directly
  → (a → [a])            -- split
  → (a → [b] → b)        -- merge
  → a                    -- input
  → b
```

We used a branching degree of two. Dataset size determined whether the JCN could be computed directly; if not, the split function divided the dataset along the largest spatial dimension.

4.4 Basic Workpool

A well-understood problem with parallelization by domain decomposition is that computational work does not depend on the size of the domain, but on properties of the data. Vector field visualization, where streamlines are calculated by integrating along paths from a set of seed points, are one obvious example [31], but similar problems arise even with scalar field analysis. In the case of the JCN, the number of subdivisions in a cell depends on the local gradient, and datasets from many phenomena exhibit substantial variation across the domain. Preprocessing data to find a more judicious partition is one option, though potentially expensive. The second, and the current method of choice in high-performance computational science, is overdecomposition [19], breaking the input into many more subdomains than processing elements, and then maintaining a pool of work that can be allocated to available processors to even out the overall workload.

In the case of Eden, the library provides a family of 'workpool' skeletons with dynamic load balancing capability. Workpool evaluation is a form of parallel map, with the 'list' replaced by a pool of tasks; the skeleton applies a given function to each element of the list. For the JCN, we created a pool of processes to perform the fragmentation phase. Reduction of the intermediate JCNs into the final result was performed by the master process, and unsurprisingly limited scalability as the number of processors increased.

Such bottlenecks are not peculiar to the JCN algorithm. In anticipation, Eden's skeleton library includes extensions to the simple workpool model that integrate map and reduction operations [1, 9]. Unfortunately, these all allow the merger of intermediate results *in arbitrary order*. Merger of non-adjacent JCNs results in a disjoint graph and no opportunity for simplification.

4.5 A New Skeleton: Multilevel Map-Reduce Workpool

Our initial work using Eden's generic skeleton library identified limitations: either too little flexibility, in the case of the $disDC$ decomposition strategy, or too much flexibility in the form of unconstrained reduction in $workpool$. Motivated by the success of overdecomposition in other computational science applications, we decided to develop a more specialised $workpool$ model that would:

1. permit dynamic load balancing across both fragmentation (map) and merger (reduce) stages of the JCN algorithm; and
2. allow multi-level reductions, but with constraint on reduction order capturing, in our case, spatial adjacency.

The skeleton is parameterised over four functions and the initial input. Figure 6 shows a schematic of the skeleton including internal components; its type signature and (outline) definition are given below. Note that construction of the task list (ts) from merge tasks (mts) and reduction tasks (rts) exploits lazy evaluation.

$$
\begin{aligned}
&newWorkpool\ ::\ (Trans\ t, Trans\ r)\\
&\qquad \Rightarrow (t \to ([t], RedOrder)) && \text{-- split function}\\
&\qquad \to (t \to r) && \text{-- map function}\\
&\qquad \to (r \to r \to r) && \text{-- reduce function}\\
&\qquad \to (r \to Int) && \text{-- inquiry function}\\
&\qquad \to t\\
&\qquad \to r
\end{aligned}
$$

$$
\begin{aligned}
&newWorkpool\ split\ mf\ rf\ qf\ p\\
&\quad = fetch \circ snd \circ last\ \$\ ir\\
&\quad \textbf{where}\\
&\qquad (mts, redOrd) = split\ p\\
&\qquad ir\ \ = workpoolSorted\ noPe\ 3\ (mapRed\ mf\ rf\ qf)\ ts\\
&\qquad ts\ \ = mts +\!\!+ rts\\
&\qquad rts = mergerTask\ (map\ snd\ ir)\ redOrd
\end{aligned}
$$

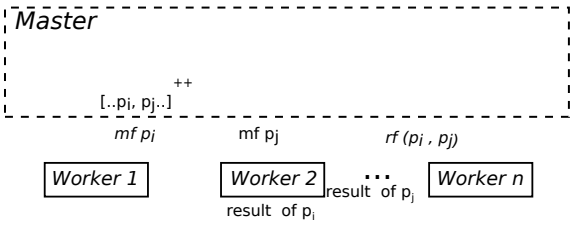


Figure 6. Structure of the newWorkpool skeleton. This figure also shows that $Worker\ 1$ and 2 are assigned to apply the 'map' function on sub-problems p_i and p_j, respectively, and $Worker\ n$ is assigned to merge their result. Dynamic links are created between these workers.

To expand on the implementation,

1. $split$ subdivides a given problem, returning a list of subproblems along with a *reduction order*, a list of integers which represent the number of subsets in each dimension. Reduction order is used in the merger phase to identify spatially-adjacent subdomains.
2. map is applied to each sub-problem returned by split. To obtain dynamic load balancing, Eden's basic workpool skeleton is used to apply map function to the sub-problems, creating a set of worker processes and distributing the map tasks among them.
3. $mergerTasks$ uses the reduction order to match intermediate results in a multi-level scheme. Its implementation again uses the workpool skeleton for dynamic load balancing.
4. $reduce$ effects the merger of two JCNs.

As shown in Fig. 6, a worker is usually assigned to perform a merger task on two JCNs that are computed by other workers. In our implementation, to reduce the overhead of communication to the master process, dynamic communication links are created between workers to transfer the intermediate JCNs[2].

5. Performance and Evaluation

This section reports performance results obtained from a cluster running CentOS 6. Each node has a dual socket with 2.6GHz 8-core Intel E5-2670 processors (thus 16 cores per node), 32GB of RAM, and 500Gb local hard drive. Storage is provide via a Lustre filesystem delivering 4GB/s via an InfiniBand network. We have used two datasets:

1. Scission: a simulation of nuclear fragmentation, previously reported in [11, 12]. It consists of two scalar fields over a $40 \times 40 \times 66$ mesh, with one byte per sample. Slab width was set to 2. Input dataset size is 211KB; the final JCN has in the order of 17K vertices and 34K edges, derived from 2.9M fragments produced in the first phase.
2. Isabel: The hurricane Isabel simulation developed by the National Center for Atmospheric Research in the United States [18]. While this consists of multiple time-varying scalar and vector fields, for present purposes we work with two fields,

[2] The *inquiry* function passed as a parameter to the skeleton is an unfortunate implementation detail. We use Eden's 'remote data' capability to avoid the overhead of Eden's default hierarchical data transfers. However, passing the remote data back to the parent process does not force the computation on a child process. Instead, the parent process can ask the child process, via the *inquiry* function, for a datum, evaluation of which will force the child computation. The overhead of sending this datum back to the parent is much less than sending the whole results.

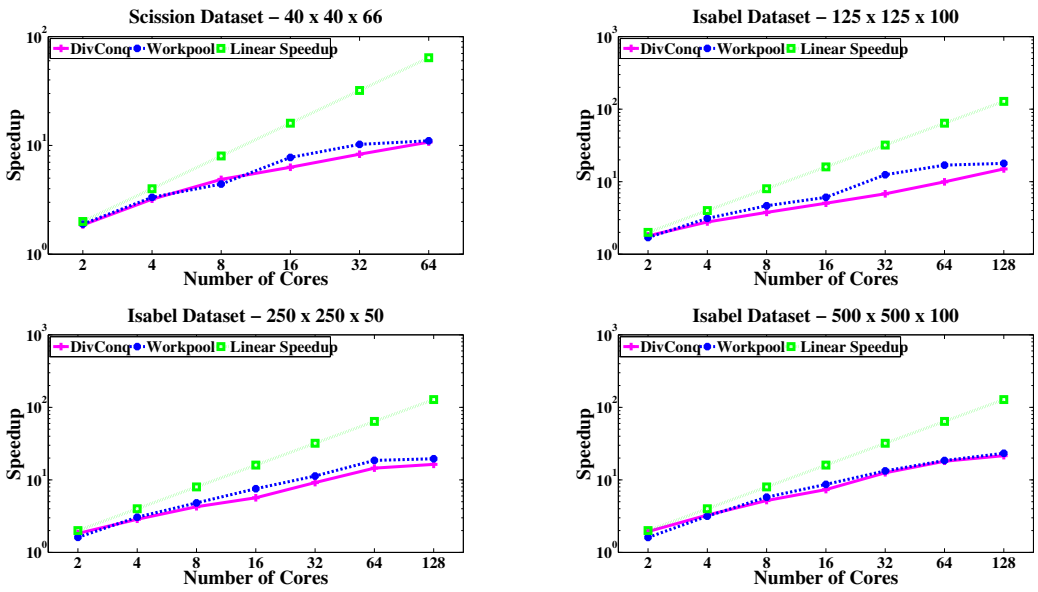


Figure 7. Speedup of the JCN computation relative to best sequential runtime (log-log plot).

Table 1. Isabel Runtime (s) for Divide and Conquer (DC) and newWorkpool (WP), and with non-merger (WoM).

Resolution	$125 \times 125 \times 100$		$250 \times 250 \times 50$			$500 \times 500 \times 100$		
Nr Cores	DC	WP	DC	WP	WoM	DC	WP	WoM
1	224.76	273.94	452.54	597	448.2	3351.06	5097.65	3354.78
2	125.55	132.33	247.81	280.69	226.18	1727.59	2083.81	1700.77
4	80.8	72.02	157.99	147.73	120.63	1024.73	1061.33	876.33
8	59.6	43.21	106.16	94.01	68.65	645.81	580.33	457.56
16	44.53	36.92	79.89	59.88	47.09	455.86	386.22	275.88
32	33.02	18.03	49.3	40.02	23.88	267.35	251.9	160.99
64	22.58	13.29	31.15	24.38	12.83	185.42	181.16	76.33
96	15.51	13.09	28.68	21.96	9.2	175.81	158.72	52.44
128	14.99	12.56	27.64	23.12	8.17	155.25	143.96	41.83

'precip' and 'cloud', from timestep 10. Fields are stored as 4-byte floating point values. The native resolution of the mesh is $500 \times 500 \times 100$, but we also used down-sampled versions, at $250 \times 250 \times 50$ and $125 \times 125 \times 100$ resolution. The slab widths for the precip and cloud fields were set to 0.0005 and 0.00007, respectively. At full resolution the input dataset size is 200MB; the JCN has in the order of 17K vertices and 30K edges constructed from 6M fragments.

Figure 7 gives the speedup obtained for different datasets using two parallel skeletons: Divide and Conquer (DC) and newWorkpool (WP). Table 1 gives the corresponding runtimes. Runtime results, backed by inspection of profiling data via Eden's trace viewer show some benefit from the use of dynamic load balancing in the WP skeleton, while performance of the DC skeleton suffers from under-utilization of processing elements during the merger phase. However, the DC skeleton gains from lower communication overheads than WP: each processing element involved in a merger already has computed one of the two JCNs, and therefore has that data locally. WP, in contrast, has no such guarantee, and does not consider the overhead of data communication when assigning merger tasks to processing elements. These costs are significant: Figure 7 shows that particularly for the full-resolution Isabel dataset, as communication overhead for WP wipes out any benefit from greater PE utilization, and performance of the two skeletons converges. This result is confirmed by a separate scalability study, where speedup of the full JCN implementation was compared with speedup over just the low-communication fragmentation phase, i.e. with no merger. Raw runtimes for the Isabel dataset are reported in the 'WoM' columns of Table 1 and are plotted in Fig. 8. With small numbers of cores, merger makes little difference to overall speedup, but the cost increases with core count; for the full-resolution Isabel dataset, the speedup obtained without merger is nearly four times that of the whole computation.

These results underline the work reported in the next section, on alternative approaches to distributed processing to reduce communication costs by limiting the amount of data transferred. But to conclude this section we compare performance of the distributed implementation with the shared-memory implementation [12]. This is not a straightforward comparison, particularly due to differences in IO: in the distributed implementation each process accesses the input files directly, while in the shared memory implementation the main process reads the input files and then copies the relevant subsets to child processes. However, Table 2 sets out the performance of both implementations. The distributed implementations, using either DC or WP skeletons, have shorter runtime and scale better, and significantly the shared memory implementation does not scale beyond 8 cores, due to the overhead of garbage collection and the need to suspend threads on *all* cores while collection takes place.

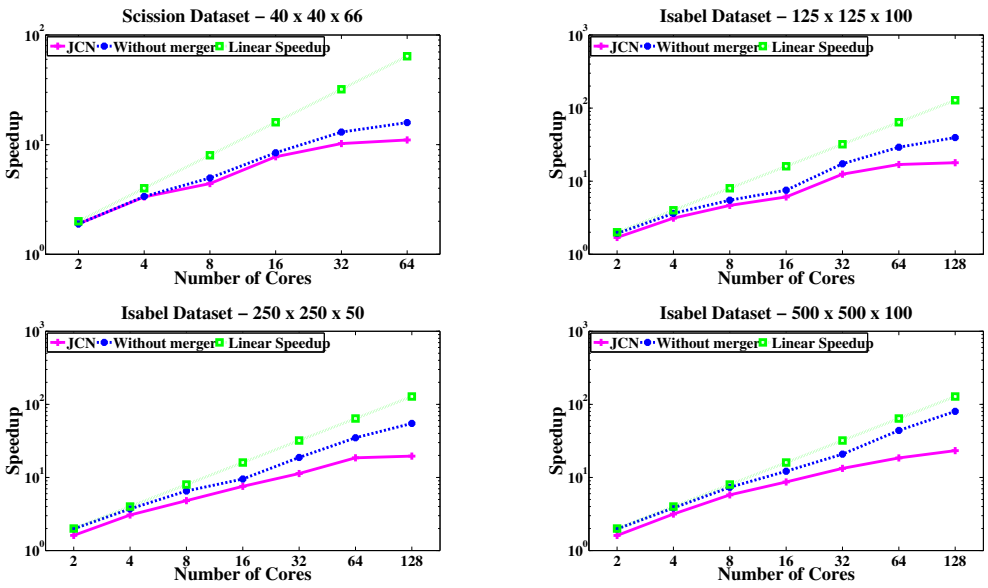


Figure 8. Speedup of the JCN without merger relative to the sequential baseline

Table 2. Performance of Shared Memory implementation compared with Distributed Memory skeletons (nuclear dataset)

	Run-time (s)			Speedup		
Nr Cores	SM	DC	WP	SM	DC	WP
1	119.2	73.1	72.8	1	1	1
2	77.6	39.2	38.5	1.5	1.9	1.9
4	50.2	22.7	21.8	2.4	3.2	3.3
8	36.6	14.9	16.5	3.3	4.9	4.4
16	36.6	11.6	9.4	3.3	6.3	7.8

We do not have performance data for an imperative implementation on the cluster hardware. However, as a crude benchmark, on a MacBook Air with a 2GHZ Intel i7 processor, 8GB RAM, running OSX 10.10.4, the sequential C++/VTK implementation described in [5] takes 106s for the 'scission' dataset, and 123s for the full-resolution 'Isabel'.

6. Distributing the Data Structure

Our strategy of distributing the computation of local JCNs and then reducing these to a single global result does not scale well due to the communication costs in the merger phase. This problem is not unique to the JCN algorithm, and as reported in Section 2.3 has recently motivated innovative work on distributed data structures and incremental computation for other kinds of topological abstraction.

We have therefore begun to investigate an alternative strategy for computing the JCN, consisting of a distributed representation and an incremental update strategy. The approach is based on the following observation: when merging JCNs, only the nodes lying on the boundary will be affected (merged with or linked to other nodes). To merge two JCNs a and b, we first extract and merge their boundary nets, $\overline{a}$ and $\overline{b}$ to give $\overline{c}$. Given this, we can *locally* update a and b to a' and b', respectively, such that the simple set-union of the nodes and edges in a' and b' will be equal to the full merger of a and b. The cost saving comes from (a) for non-trivial data, the size of the boundary JCN will be significantly smaller than the full JCN, and (b) that the final 'union' operator can be effected cheaply by e.g. writing the distributed sub-graphs into a single file.

Figure 9 illustrates the approach. Part (a) shows the input dataset, a bi-variate field divided into two sub-domains, each with two simplicial cells. In (b) the sub-domains have been independently fragmented into sets of slabs on the two fields (subdivisions over the first field shown by solid red lines between circled points, over the right field shown by dashed blue lines between points marked by bars). Part (c) of the figure shows the locally-computed JCNs, with non-border (interior) edges indicated by blue-dashed lines, Finally, part (d) shows the distributed merger of the two JCNs. The two subgraphs are stored on their local nodes, with edges at the boundary allocated to the node with lower PE number depending on a classification of nodes and edges as boundary/internal.

On each node the JCN is implemented as a map structure from node ids to its range value and neighboring node ids. A more compact *unboxed* (UJCN) representation, built on unboxed vectors, is used for communication between processors.

$$\begin{aligned}\textbf{data}\ JCN\ d\ r = JCN\ \{&net\ \ \ \ :: IntMap\ (r, [Int])\\ ,&kdtree :: KDtree\ d\ Int\\ ,&size\ \ \ \ :: Int\}\end{aligned}$$

$$\begin{aligned}\textbf{data}\ UJCN\ d\ r = UJCN\ \{&slabs\ \ :: Vector\ (Int, r)\\ ,&edges\ \ :: Vector\ (Int, Int)\\ ,&border :: Vector\ (d, Int)\\ ,&slabNo :: Int\}\end{aligned}$$

A skeleton is parameterised over (a) the number of divisions in each dimension, the number of communication rounds required, and a list of the spatial subdomains ordered based on spatial adjacency. It implements a series of boundary exchanges across an expanding frontier. Each node first computes its local JCN and boundary JCN, then goes through a number of exchange rounds. In each round it (a) transmits its boundary JCN to the nodes of JCNs on the frontier, and (b) on receiving the boundary JCN from a neighbour, merges that with its local boundary. After each round the frontier for each subdomain expands outwards, so that the number of rounds can be bounded above by $\lceil \log_3 n \rceil$ of the maximum number n of domain divisions across all spatial dimensions.

Preliminary runtime results show a reduction in communication overhead, but further work is required before we can confidently

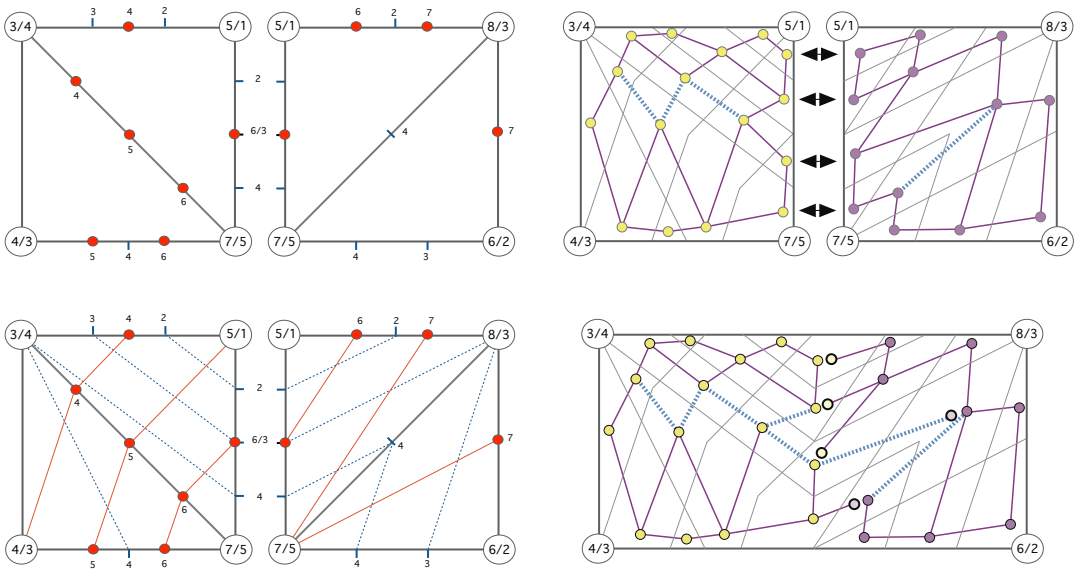


Figure 9. Distributed construction of the JCN.

report the level of performance improvement; outstanding concerns include the cost of converting between different representations of the JCN, and whether these can be avoided.

7. Conclusions

Programmer productivity has had a long influence on language design, with Backus, remarking on the development of FORTRAN, reported as stating "*Much of my work has come from being lazy. I didn't like writing programs ... I started work on a programming system to make it easier to write programs.*" [3]. As computer science has matured, and sub-specialisms have emerged, there is a danger that this tight and productive coupling between programming language and applications research has become obscured. Skeletons for distributed functional programming are a case in point; extending the benefits of higher-order functions to provide generic patterns for distributed processing has resulted in conceptually elegant ideas that work well on a number of problems, but exposure to wider computational challenges, such as computational topology, will inevitably improve the robustness and utility of these abstractions. Dually, much of the progress in computational science, even within highly innovative and effective systems [19], draws heavily on systems and language technologies already established in those communities. One of the longer term goals of this research is to explore whether functional technology can disrupt this equilibrium, delivering a more profound change in approach in the face of the emerging challenges of extreme-scale computing.

Implementation of the JCN algorithm using Haskell/Eden's high level support for distributed memory parallelism has allowed us to explore rapidly different distributed processing strategies and examine the performance trade-offs. Our implementation benefits from a clear separation between the fundamental algorithm and the skeletons that implement distribution; indeed significant parts of the implementation have been carried over from the original sequential version reported in [12]. We have improved scalability from the shared-memory implementation, but have also run into a significant barrier in the form of communication costs. Work on overcoming that barrier by adopting a *distributed* representation of the JCN is ongoing.

The introduction of the paper noted the growing adoptation of run-time systems and DSLs for high-performance computing. Production systems such as Charm++ [19], and experimental platforms like Galois [30] and LVish [20] include sophisticated support for workpools, including flexible schedulers and work allocation strategies. Further work is needed to understand the relationship between these platforms, and to identify where opportunities exist to transfer insights into, for example, workpool models specialised for problems in computational science.

Beyond the general lesson of communication costs, our work to date has flagged up three points related to Eden:

1. Even well-developed functional programs are not immune from correctness issues, particularly when applied to extreme cases: in the course of developing the new workpool skeleton we uncovered a subtle issue where the existing skeleton was more eager than it need be; we are grateful to Eden developers for assistance in finding and correcting this.
2. Although Eden's trace viewer was useful in understanding aspects of distributed performance, significantly more work is required on making these interfaces scalable, for example to processes running on 1000s of cores (our next target).
3. Further work is needed on the foundations for distributed skeletons and process abstractions, in particular on dealing with the interface to distributed filesystems. The current process abstraction sits uneasily with the need for distributed processes to do their own IO (i.e. to run inside the IO monad), and our current solution relies rather awkwardly on Haskell's trap-door, $unsafePerformIO$.

To conclude, we return to the question of skeletons as a generic building blocks for (high performance) distributed processing. Higher-order functions succeed for (at least) two reasons: they capture widely occurring patterns of computation, and advances in compiler technology have made their use relatively cost-free. Given the communication overhead, distribution of processing across a cluster implies greater understanding of the domain, and assumptions about the interaction between the sub-processes. The scope and effectiveness of truly 'generic' skeletons is therefore less clear. One could however envisage a skeleton library specialised to an application domain, with its behaviour constrained by type classes

associated with its inputs. One test for the maturity of distributed Haskell may well be whether such libraries begin to emerge.

Acknowledgments

The work reported in this paper was funded through EPSRC Grant EP/J013072/1 (Multifield Extensions of Topological Analysis). We particularly thank Jost Berthold, Thomas Horstmeyer, and Hans-Wolfgang Loidl for assistance with distributed Haskell and Eden. Our thanks also to the anonymous reviewers for their constructive comments on how to improve the paper.

References

[1] J. Berthold, M. Dieterle, R. Loogen, and S. Priebe. Hierarchical master-worker skeletons. In *Proceedings of the 10th International Conference on Practical Aspects of Declarative Languages (PADL'08)*, pages 248–264, 2008.

[2] J. Biddiscombe, B. Geveci, K. Martin, K. Moreland, and D. Thompson. Time dependent processing in a parallel pipeline architecture. *Trans. on Vis. and Computer Graphics*, 13(6):1376–1383, 2007.

[3] E. Blum and W. Savitch. The software side of computer science - computer programming. In *Computer Science: The Hardware, Software and Heart of It*. Springer, 2011.

[4] S. Breitinger, U. Klusik, and R. Loogen. From (sequential) haskell to (parallel) eden: An implementation point of view. In *In PLILP'98. Springer LNCS 1490*, pages 318–334, 1998.

[5] H. Carr and D. Duke. Joint contour nets. *Trans. on Vis. and Comp. Graphics*, 20(8):1100–1113, 2014.

[6] H. Carr, T. Möller, and J. Snoeyink. Simplicial subdivisions and sampling artifacts. In *Proc. of the Conference on Visualization '01*, pages 99–106. IEEE Computer Society, 2001.

[7] A. Chattopadhyay, H. Carr, D. Duke, and Z. Geng. Simplifying multivariate topology (extended abstract). In *Computer Graphics and Visual Computing*. Eurographics Association, 2014.

[8] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: A domain specific language for building portable mesh-based pde solvers. In *Proc. of 2011 Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 9:1–9:12. ACM, 2011.

[9] M. Dieterle, J. Berthold, and R. Loogen. A skeleton for distributed work pools in eden. In *Proceedings of the 10th International Conference on Functional and Logic Programming (FLOPS'10)*, pages 337–353. Springer-Verlag, 2010.

[10] D. Duke, R. Borgo, C. Runciman, and M. Wallace. Huge data but small programs: Visualization design via multiple embedded DSLs. In *Proc. Practical Applications of Declarative Languages*, volume 5418, pages 31–45. Springer Verlag, 2009.

[11] D. Duke, H. Carr, A. Knoll, N. Schunck, H. Nam, and A. Staszczak. Visualizing nuclear scission through a multifield extension of topological analysis. *Trans. on Vis. and Comp. Graphics*, 18(12): 2033–2040, 2012.

[12] D. Duke, F. Hosseini, and H. Carr. Parallel computation of multifield topology: Experience of Haskell in a computational science application. In *Functional High Performance Computing*. ACM Press, 2014.

[13] H. Edelsbrunner and J. Harer. *Jacobi sets of multiple Morse functions*, pages 37–57. Cambridge University Press, 2004.

[14] H. Edelsbrunner, J. Harer, and A. K. Patel. Reeb spaces of piecewise linear mappings. In *SCG '08: Proceedings of the twenty-fourth annual symposium on Computational geometry*. ACM Press, 2008.

[15] Eden. Eden Skeleton Library. `https://hackage.haskell.org/package/edenskel`.

[16] Z. Geng, D. Duke, H. Carr, and A. Chattopadhyay. Visual analysis of hurricane data using joint contour net. In *Proc. Computer Graphics and Visual Computing*. Eurographics, 2014.

[17] A. Gyulassy, V. Pascucci, T. Peterka, and R. Ross. The parallel computation of morse-smale complexes. In *Parallel Distributed Processing Symposium*, pages 484–495. IEEE Press, 2012.

[18] Isabel. IEEE visualization 2004 contest, 2004. `http://vis.computer.org/vis2004contest/index.html`.

[19] L. Kale and A. Bhatele. *Parallel Science and Engineering Applications: The Charm++ Approach*. CRC Press, 2013.

[20] L. Kuper, A. Todd, S. Tobin-Hochstadt, and R. R. Newton. Taming the parallel effect zoo: Extensible deterministic parallelism with lvish. In *Proc of Programming Language Design and Implementation*, pages 2–14. ACM Press, 2014.

[21] A. G. Landge, V. Pascucci, A. Gyulassy, J. C. Bennett, H. Kolla, J. Chen, and P.-T. Bremer. In-situ feature extraction of large scale combustion simulations using segmented merge trees. In *Proc. of High Performance Computing, Networking, Storage and Analysis*, pages 1020–1031. IEEE Press, 2014.

[22] R. Loogen, Y. Ortega-mallén, and R. Peña marí. Parallel functional programming in eden. *J. Funct. Program.*, 15(3):431–475, 2005.

[23] S. Marlow, R. Newton, and S. Peyton Jones. A monad for deterministic parallelism. In *Proc. of Haskell Symposium*, pages 71–82. ACM, 2011.

[24] B. McCormick, T. DeFanti, and M. Brown. Visualization in scientific computing. *Computer Graphics*, 21(6), 1987.

[25] Q. Meng, A. Humphrey, and M. Berzins. The Uintah framework: A unified heterogeneous task scheduling and runtime system. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 2441–2448. IEEE, 2012.

[26] D. Morozov and G. Weber. Distributed merge trees. *SIGPLAN Not.*, 48(8):93–102, 2013.

[27] D. Morozov and G. H. Weber. Distributed contour trees. In *Topological Methods in Data Analysis and Visualization III*, Mathematics and Visualization, pages 89–102. Springer, 2014.

[28] B. O'Sullivan, J. Goerzen, and D. Stewart. *Real World Haskell*. O'Reilly Media, Inc., 2008.

[29] V. Pascucci and K. Cole-McLaughlin. Parallel computation of the topology of level sets. *Algorithmica*, 38(1):249–268, 2004.

[30] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *Proc. of Programming Language Design and Implementation*, pages 12–25. ACM Press, 2011.

[31] D. Pugmire, T. Peterka, and C. Garth. Parallel integral curves. *in: High Performance Visualization*, 2012.

[32] P. Rautek, S. Bruckner, M. E. Gröller, and M. Hadwiger. ViSlang: A system for interpreted domain-specific languages for scientific visualization. *Trans. on Vis. and Comp. Graphics*, 20(12):2388–2396, 2014.

[33] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*. Kitware, 2006.

[34] N. Schunck, D. Duke, H. Carr, and A. Knoll. Description of induced nuclear fission with skyrme energy functionals: I. Static potential energy surfaces and fission fragment properties. *Physical Review C*, 90(5), 2014.

[35] N. Schunck, D. Duke, and H. Carr. Description of induced nuclear fission with skyrme energy functionals: II. Finite temperature effects. *Physical Review C*, 91(3), 2015.

[36] N. Shivashankar and V. Natarajan. Parallel computation of 3D morse-smale complexes. *Comp. Graph. Forum*, 31(3):965–974, 2012.

[37] N. Shivashankar, S. M, and V. Natarajan. Parallel computation of 2D morse-smale complexes. *Trans. on Visualization and Comp. Graphics*, 18(10):1757–1770, 2012.

[38] A. Telea. *Data Visualization: Principles and Practice*. A.K. Peters, 2008.

[39] G. Zhao, J. Perilla, E. Yufenyuy, X. Meng, B. Chen, J. Ning, J. Ahn, A. Gronenborn, K. Schulten, C. Aiken, and P. Zhang. Mature HIV-1 capsid structure by cryo-electron microscopy and all-atom molecular dynamics. *Nature*, 497(7451), 2013.