University of Leeds
SCHOOL OF COMPUTER STUDIES
RESEARCH REPORT SERIES
Report 96.01

Constructive Solid Geometry Using Algorithmic Skeletons[1]

by

J R Davy, H. Deldari & P M Dew
Division of Computer Science

Jan 1996

[1] To appear in *Proc. 5th Eurographics Workshop on Programming Paradigms for Graphics*, Springer, 1995

Abstract This paper presents a study in the use of parallel algorithmic skeletons to program applications of constructive solid geometry (CSG). The approach is motivated by the frequent use of divide-and-conquer (D&C) methods in this domain, which are amenable to highly parallel implementation. A prototype *Geometric Evaluation Library* (GEL) is presented, with a small set of polymorphic higher-order functions, which capture the fundamental algorithmic structures independently of the underlying geometric domain. An efficient parallel implementation of one of these functions is described. The paper concludes with a discussion of the potential of the algorithmic skeleton paradigm.

1 Introduction

Solid modelling is a computationally-intensive technique which can benefit from the exploitation of parallelism. Parallel processing brings substantial performance benefits but creates new programming challenges. Besides writing the code which defines the fundamental computational activity of the applications, the programmer must also consider issues such as decomposition into parallel processes, balancing the load between these processes, communicating between them, and synchronising their operations. These latter activities, some of which more properly belong to the system level, may occupy a large part of both development time and final code. Moreover, many parallel solutions are heavily dependent on the target machine, so that it has often been difficult to exploit the benefits of :x

parallelisation when moving to a new platform.

This paper discusses the potential for developing parallel programs for *constructive solid geometry* (CSG) in a way which avoids these problems. Specifically we carry out a case study in the use of *algorithmic skeletons*, enabling applications to be defined in terms of a small number of well-defined constructs which can be mapped onto known parallel implementations. Program development then emphasizes appropriate selection and customisation of existing algorithmic components described at a high level of abstraction, leading naturally to faster development times and machine-independent solutions. A similar approach has already been used for special-purpose languages in the domain of image processing [20, 21] and numerical solution of differential equations by multigrid methods [18].

For this paradigm to be successful, two conditions must be satisfied:

- there must be suitable set of skeletons which are adequate to define applications in the domain in question;
- these skeletons must be capable of efficient parallel implementation.

The initial stage of the study addressed the first issue. A prototype *Geometric Evaluation Library* (GEL) was developed in a functional language, to investigate the potential for developing CSG applications by means of a small number of *higher order functions* specific to this application domain. The second stage, still in progress, has begun the process of developing parallel algorithmic skeletons for the key operations in GEL.

A previous paper [7] reported in detail on the development of GEL as an exercise in functional programming. This paper gives a broader view of the algorithmic skeleton paradigm; we briefly summarise GEL, report on progress towards parallelising CSG skeletons, and evaluate the potential of the paradigm in this field.

2 Algorithmic Skeletons

There has recently been considerable interest in the parallel programming world in the use of *algorithmic skeletons* [4, 5, 8]. These are generic patterns of parallel computation which can be parameterised by a small number of sequential functions or procedures (called here *customising functions*). A complete parallel program can be generated by inserting the customising functions into a template with an efficient parallel implementation for the generic solution. Thus machine-independent parallel programs can be developed in which all the implementation issues of parallelism are hidden from the programmer.

There is an obvious analogy between skeletons and higher-order functions; indeed much of the existing research in algorithmic skeletons has taken place in the context of higher-order functional languages. This has the benefit that program transformations may be used to derive more efficient implementations. Skeletons can, however, be used within more conventional language paradigms.

Most existing work on skeletons has studied general algorithmic forms, such as pipelines and divide-and-conquer. By contrast, *application-specific languages* exploit skeleton-based computation in

a particular application domain. For instance, Apply [20] is a procedurally-based language which capture various aspects of image processing. Here knowledge of the application domain, particularly the underlying data types, enables efficient parallel implementations to be generated from partial sequential descriptions.

In the following section we present initial evidence that CSG computations are also amenable to a skeleton-based application-specific approach. The rest of this paper supports this view, firstly by showing constructs to describe CSG applications in terms of a small number of higher order operations, and secondly through an efficient parallel implementation of one of these key constructs.

3 Constructive Solid Geometry (CSG)

Solid modelling systems manipulate descriptions of three-dimensional objects. The distinguishing feature of solid models is *completeness*: a description of a solid contains sufficient information to compute any geometric property of that solid.

Several schemes have been devised for complete representations of solids [17]. One of the most used is CSG, in which solids are represented by a set of *primitive solids* combined using *regularised boolean operations* such as union, intersection and difference. It is common for a small set of *bounded primitives*, such as spheres and cones, to be available to the users of CSG systems. Internally they are often represented as a combination of simpler unbounded primitives called *halfspaces*. These are defined by functions of the form $f(x, y, z) \leq 0$ which partition space into two halves, inside and outside the primitive.

CSG represents a solid as a tree structure with primitives at the leaves and boolean operators at interior nodes. This leads naturally to recursive *divide-and-conquer* (D&C) algorithms, which compute results for primitives, then combine the results using the boolean operators when returning up the tree. Tilove identified this as a generic paradigm for CSG, applying it to *set membership classification* problems [19]. An example used later in this paper is point membership classification (PMC), which determines whether a point is inside, outside or on the boundary of a solid. Classifications are carried out on primitives and results from subtrees are determined using simple rewrite rules: for instance if a point is 'in' two solids it

is 'in' their intersection.

3.1 Spatial Subdivision

It is common to convert CSG trees into secondary data structures based on *octrees* [15], using a process of *spatial subdivision*. Here the (usually cubical) space containing the solid object is recursively partitioned into eight quasi-disjoint subcells. Associated with each terminal subcell is a localised CSG tree containing only the primitives which intersect the subcell. The depth of subdivision is controlled by the local complexity of the model. For instance, subdivision may stop when the number of primitives in a localised tree falls below some threshold.

The benefits of spatial subdivision are exemplified in ray-tracing [10]. Crude CSG ray-tracing is very expensive, intersecting every ray with every primitive. When ray-tracing a spatially divided model, each ray is tracked through the octree and intersects only the primitives of the localised trees encountered, giving substantial performance improvements.

There are also many algorithms which use spatial subdivision but without explicitly creating a tree: the family of algorithms for computing integral properties of solids [14] is a good example.

Many spatial subdivision algorithms follow a D&C approach. Thus CSG computations lead to D&C algorithms both because of the primary data structure and because of frequently used spatial subdivision techniques. It is these algorithmic structures which we seek to capture and exploit in a generic fashion as skeletons. They are naturally highly parallel, since subtrees can be processed independently, but managing the tree structures causes substantial practical problems for parallelisation.

3.2 Potential Parallelism

The high computational requirements of solid modelling indicate the desirability of parallel processing. Earlier work explored this through the Mistral series of experimental parallel CSG systems [9, 10]. The most recent of these, Mistral-3, achieved over 80% efficiency for ray-tracing a spatially divided CSG model on 128 T800 transputers, including creating the spatially divided model in parallel [10]. Parallelism was obtained almost entirely by exploiting the D&C paradigm.

Since the tree structures were typically highly unbalanced and developed dynamically, distributed dynamic load balancing techniques were required to gain this scalable performance.

These results demonstrate that excellent parallel performance is possible for CSG by exploiting the underlying data structures through D&C processing. The price paid, however, is the complexity of the software; most of the code (and effort) of Mistral-3 provided system-level infrastructure for parallelism, particularly dynamic load balancing and the movement of complex data structures. Arguably these should not be the responsibility of an application programmer; this motivates the desirability of a higher-level approach to free the programmer from these concerns. The recurring use of D&C algorithms provides a promising approach for investigation.

4 GEL: a prototype library

GEL (Geometric Evaluation Library) is a prototype library of higher order functions designed to study the systematic use of D&C operations for CSG and spatial subdivision. It has been used to explore the hypothesis that a realistic range of algorithms could be developed using such a restricted set of operations. A pure functional language, Hope+, was chosen to prototype GEL, since functional languages provide a natural means to define higher order functions.

Here we briefly outline the main features of GEL; a more comprehensive discussion can be found in [7]. Readers familiar with other higher-order functional languages should find no difficulty with the notations of Hope+.

4.1 Principles of GEL

The current version of GEL is based on the following principles:

- The main data structures are trees parameterised by an arbitrary geometric type, enabling the same operations to be carried out in different geometric domains.
- The primary skeletons are D&C operations on these data structures, which are all specialised variants of a general D&C operation *divacon* invoked as

```
divacon(data, leaf, divide, solve, combine);
```

with four customising functions: *leaf* determines whether a problem is small enough to solve directly, *divide* splits a problem into a list of subproblems, *solve* computes the direct solution of a 'small' problem, *combine* combines the results of a list of subproblems.

- Variant forms of the primary skeletons are provided as a set of overloaded functions, disambiguated by the number and types of the parameters.

4.2 CSG Trees

CSG trees are binary trees with instances of primitive solids at the leaves and instances of boolean operators at interior nodes. Since CSG is a set-theoretic representation, there should be a means of denoting the empty set ($\emptyset$) and the universal set (Ω) in a CSG system. These considerations lead to the following algebraic type definition, parameterised by *rho* and *chi*, the types for primitives and operators respectively:

```
data CSG(rho, chi) ==
      Emptysolid  ++  Fullsolid  ++  Primitive (rho)  ++
      Compose (CSG(rho, chi) # chi # CSG(rho, chi));
```

Note that *chi* is an enumeration of the specific set of operators used, such as {*Intersection, Union, Difference*} or {*Union, Difference*}. Its inclusion as a parameter, rather than as a fixed enumeration, is motivated by the observation that CSG systems vary in both the number and specific kinds of boolean operations allowed. The actual implementation of these operations is carried out by combining functions.

The primary higher-order operation on CSG trees is an *evaluation* of the tree. This is a D&C traversal, suitable for the family of set membership classification problems noted in section 3. A typical invocation, to classify a point against a solid, is

```
CSGtraverse(point, tree, solve, combine);
```

where *solve* classifies a point against a primitive, and *combine* implements the boolean operations to combine classifications.

Note that a binary CSG tree may include Ω nodes (to model complementation as $\overline{P} = \Omega - P$), and trees with $\emptyset$ primitives may

exist as intermediate steps. Also, there are situations in which there is no external 'query' data like the *point* above – a trivial example is an algorithm to count the number of primitives in a tree. These variants are implemented by overloading.

GEL also provides higher-order operations to input or output CSG trees. The customising functions for *CSGget* require the programmer to specify how primitives and operators should be read from an input stream (lazy list of characters); *CSGput* uses corresponding customising functions. For instance, to input a CSG tree and classify it against a point

```
let (tree, _) == CSGget(instream, get_hs, get_op) in
    CSGtraverse(pt, tree, class, combine);
```

where *get_hs*, *get_op* are the customising functions to input primitives and operators respectively. Only two higher-order functions and four (relatively straightforward) customising functions have been needed, emphasising the simplicity of the approach.

4.3 Geometric Decomposition Trees

Secondary solid representations based on octree-based spatial subdivision are generalised to a *Geometric Decomposition Tree* (GDT). Noting that the interior nodes of the octree hold no geometric information, we parameterise GDTs by the type of geometric objects stored at the leaves, τ.

```
data GDT(tau) == Terminal (tau) ++ Interior (list (GDT(tau)));
```

Two fundamental operations are needed for GDTs. *GDTcreate* builds a GDT from the primary solid representation. *GDTtraverse* is a D&C traversal of an existing GDT, analogous to *CSGtraverse*. For instance a GDT can be derived from a CSG tree by invoking

```
GDTcreate((cll,tr), leaf, divide, solve, 7);
```

where *leaf(cll, tr)* returns true if the number of primitives in *tr* is less than some threshold, and *divide(cll, tr)* creates eight subcells of *cll*, and prunes *tr* to each subcell. Pruning involves a further D&C operation, easily implemented using *CSGtraverse*. *Solve* is the identity if pruned trees are stored unchanged at terminal nodes, but might otherwise involve some transformation of the terminal geometry. The

optional final parameter, again implemented by overloading, allows, a maximum subdivision depth to be specified.

An example of the use of *GDTtraverse* is the family of recursive integral property algorithms described in [14]. For instance, computing the mass of a solid represented in octree form invokes

```
GDTtraverse(gdt, solve, combine);
```

where *solve* computes the mass of a solid at a leaf of the octree and *combine* simply adds the result of subtrees. As with *CSGtraverse*, there is an optional parameter for an external object; for instance the moment of inertia of a solid about a point *pt* is computed by

```
GDTtraverse(gdt, pt, solve, combine);
```

where *solve* now computes the moment of inertia of the terminal tree about *pt*.

Like CSG trees, GDTs can be input or output by a single higher-order functions.

4.4 Using *divacon* directly

The original description of integral property algorithms in [14] does not explicitly create an octree but evaluates the result 'on-the-fly' as each branch of the subdivision terminates. This cannot directly be captured by the GDT operations, but can easily be described by a general *divacon* operation, which effectively combines the customising functions from *GDTcreate* and *GDTtraverse*. Thus *divacon* acts as a 'safety net', catching D&C operations which do fall within the more specialised forms. In fact, all the CSG and GDT operations described above can be derived formally from *divacon* using 'fold-unfold' transformations.

Using *divacon* directly might have a performance benefit when only a single traversal of the GDT is required. Experiments with GEL, however, suggest that if the traversal is to be repeated only once more it is advantageous to create the GDT explicitly [7]. Where the GDT is re-used many times, as in ray-tracing, the performance benefits are very large.

4.5 Geometric extensibility

The key CSG and GDT operations intentionally do not depend on a specific geometric domain; in this sense GEL is 'geometry-

independent'. Parameterisation by an arbitrary geometric type allows the same algorithmic structures to be used in different domains. Indeed the main elements of GEL contain no geometric routines, other than support for basic entities such as points and vectors. This high level of polymorphism appears to be a significant strength of GEL. Domain-specific parts of geometric computations are isolated in type definitions and customising functions. Since these aspects of computational geometry are well-known for difficulties with floating point accuracy, this isolation helps to localise such problems. In principle, it should be possible to import code from other geometric libraries, taking advantage of the extensive efforts on this area.

The first task for a user of GEL is therefore to customise it for the application's requirements by defining a suitable geometric domain. For instance, a collection of three-dimensional primitives can be defined by:

```
type SP_HS  ==  real # POINT3;   ! radius # centre
type PL_HS  ==  real # VECTOR3;  ! dist. from 0 # normal to plane
! similarly for CY_HS, CO_HS

data HS  ==  sphere(SP_HS) ++ plane(PL_HS) ++
             cyl(CY_HS)    ++ cone(CO_HS);
data ROP ==  Union ++ Inter ++ Diff;

type SOLID  ==  CSG(HS, ROP);    ! using generic CSG type
```

Basic operations on these primitives, including I/O and classification, can then be written using a systematic pattern-matching approach.

Adding an additional primitive to an existing domain is also straightforward; an extra constructor is needed in the relevant type definition with a corresponding extra equation to match that constructor in each customising function. Thus adding new geometry can be done in a simple, organised fashion. Effectively, GEL provides syntactic support for extensibility, with a flexible framework for developing geometric applications in different domains.

5 Parallel CSG

A parallel version of GEL would need to provide skeleton implementations of the main higher-order functions which can be parallelised

through their D&C structure. In this section we consider the parallelisation of *CSGtraverse*, using the earlier example of PMC as a case study. The next section shows how this may be extended to a generic skeleton.

An obstacle to parallelising CSG traversals is the unpredictable and (usually) unbalanced structure of the tree. In Mistral-3 [10] CSG trees are stored in pointer-less postfix form, enabling easier migration between nodes than pointer-based versions. Terminal nodes include indices into an array of primitives. Parallel CSG operations are carried out by 'farming'; the array is partitioned into packages, these are sent on request to other processors for classification, and the results are returned. When all the primitives have been classified the results are combined sequentially using a conventional stack-based algorithm.

The main drawback of this method is that it is not possible to parallelise the combining stage. An alternative approach, described in [2], is based on the principle of *tree contraction*. Here the tree is stored in infix form in a global shared array with a separate array of primitives, stored in the natural left-to-right order. The array can be partitioned among the cooperating processes, all processes receiving approximately equal numbers of contiguous primitives. In parallel, all processes classify their segment of primitives. Then the tree is contracted from the bottom upwards, combining the classifications each time an operator is reached.

A problem arises when an operator is reached via one of its descendants (the left say,) while the other subtree is still not fully evaluated. This will lead to a delay while the other subtree is evaluated, and possible load imbalances.

There is an elegant solution to this problem, based on the *shunt* operation [2], provided the operators are limited to union and intersection. The same approach may be used for expression evaluation over any semi-ring which has $*$ and $+$ operators with identities [11].

Each node N of the tree is labelled with a pair (a_n, b_n), initialised to the identities for $*$ (intersection) and $+$ (union). For PMC these are IN and OUT respectively. Consider the situation when a node U is an evaluated left subtree (with value u), of a node F_u (with operator Op), and the sibling S_u of U is the root of an unevaluated right subtree T_2. Let the parent of F_u be G_u, and assume that the other subtree T_1 of G_u is unevaluated. This situation is shown in

figure 1.

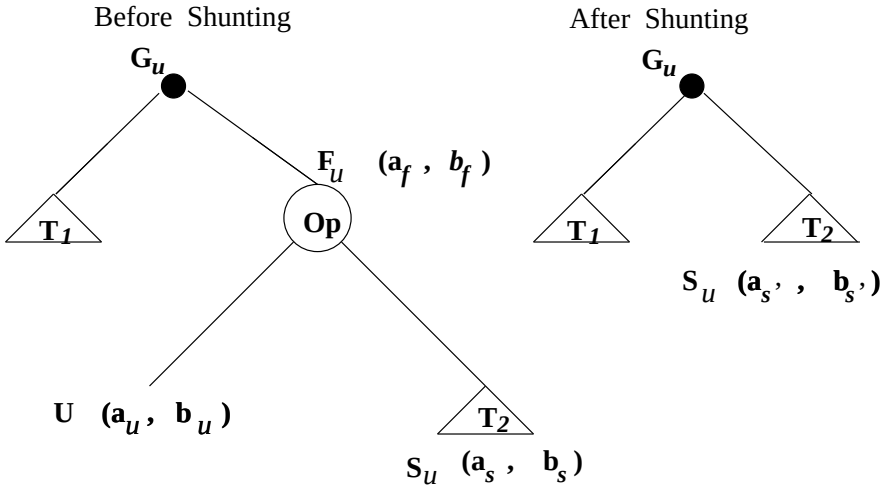

Fig. 1: The *Shunt* operation

The shunt operation allows F_u to be replaced by S_u with a new label $(a_{s'},\ b_{s'})$ given by

$$O_P = * \quad \left[\begin{array}{l} a_{s'} = a_f * (a_u * u + b_u) * a_s \\ b_{s'} = a_f * ((a_u * u + b_u) * b_s) + b_f \end{array} \right.$$

$$O_P = + \quad \left[\begin{array}{l} a_{s'} = a_f * a_s \\ b_{s'} = a_f * ((a_u * u + b_u) + b_s) + b_f \end{array} \right.$$

Thus each operator can be evaluated as it is reached, even if one subtree is not yet evaluated. Hence, effects of tree imbalance are eliminated and a naturally load-balanced algorithm is achieved. Fuller details can be found in [2] and the second author's forthcoming PhD thesis.

A parallel PMC program using this tree contraction technique was implemented in C. The target system was the WPRAM [16], a general-purpose scalable shared memory computational model developed at the University of Leeds. Currently the WPRAM exists only as a simulator, though an implementation using Inmos T9000 transputers is under way. The simulator includes a detailed performance model, parameterised by the characteristics of potential hardware components. Simulated execution times were obtained for up to 64 processors, based on parameters for Inmos T9000 and C104 components. Figure 2 shows the speedups obtained for trees with 1000

and 5000 primitives. These results are very positive in showing the potential performance gain from parallelising CSG, with over 50% efficiency on 64 processors for both trees. They also compare very favourably with the results reported in [2] for a parallel implementation of line membership classification on a BBN Butterfly, where the best reported speedup was 6.9 on 16 processors for a tree with 1001 primitives.

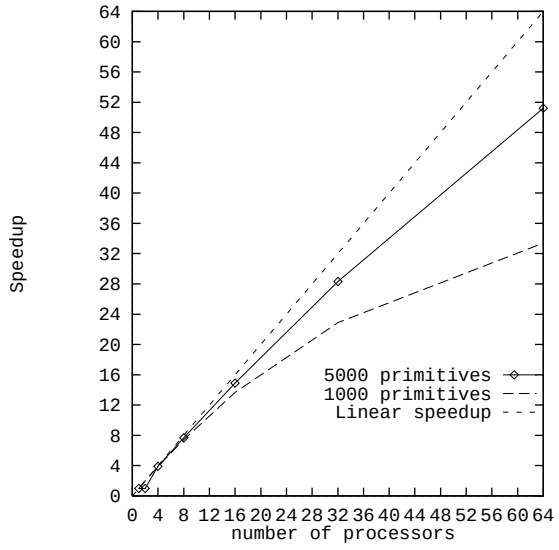

Fig. 2: Speedup of parallel PMC

6 A Parallel Skeleton

Transition to a general skeleton for D&C traversal of CSG trees is straightforward. The parallel algorithm for PMC is the same as was used in [2] for line membership classification. Thus the existing code provides a template into which customising functions for *solve* and *combine* could be inserted. A higher order function could then be mapped onto this code template.

Moving to a parallel skeleton, however, would bring some differences from GEL.

- Since tree contraction with *shunt* depends on the semi-ring structure of the underlying domain, boolean operators would

be confined to union and intersection. Thus it would be less general than the corresponding GEL operation, being restricted to positive forms of CSG tree.

- Since there are only two standard boolean operators it is simpler to have customising functions for each of these, replacing the single *combine* function by *union* and *intersection* operators. Generic CSG trees need not be parameterised by an operator type.
- The tree contraction algorithm requires the identities for the values to be combined under the intersection and union operations. These depend on the particular task being performed, hence must be specified by the programmer when calling the higher-order function. Thus the interface might be modified to

```
CSGtraverse(query, tree, solve,
            intersect, union, I_intersect, I_union);
```

7 Discussion

Though not yet complete, the current study provides enough evidence to begin to assess the potential for algorithmic skeletons in this domain. We shall consider the extent to which GEL is applicable within solid modelling and in a wider context, the practicality of programming with skeletons, the merits of a functional base language, and prospects for parallelisation.

7.1 Applicability of GEL within solid modelling

GEL's initial scope was rather limited: D&C operations in CSG. Thus it does not currently provide the full functionality of solid modelling; amongst other omissions, alternative solid representations are not currently covered, including the important *boundary representation (B-rep)*.

The potential scope of GEL is, however, much wider than appears at first sight. The parameterisation of the CSG type can capture the wide range of current or proposed CSG systems, which vary in both the formal properties of primitives, and in the specific sets of available primitives, as well as in the set of boolean operators used. CSG traversals directly model the important family of Set

Membership Classification algorithms identified by Tilove [19]; they provide a utility operation which is of wide applicability in CSG modelling.

GDT operations directly model spatial subdivision. This widely used technique has also been used on other solid representations than CSG, including B-reps, polyhedral models and sweeps. Typically, trees are generated in which terminal nodes contain an exact or approximate representation of the model localised to the relevant subspace. Variants of this type have been called *octrees* (with the equivalent *quadtrees* in two dimensions), *polytrees* and *extended octrees*. A similar *bintree* structure has been proposed with a binary subdivision at each stage, cycling between the dimensions. All these structures can be modelled by the GDT type.

In addition to applications previously noted, spatial subdivision techniques have been used for wire-frame edge evaluation, NC program verification, collision detection, finite element mesh generation and boolean operations in polyhedral modellers. In view of this broad applicability, spatial subdivision can be seen as a fundamental approach to solid modelling applications.

On the other hand, the limitation to D&C means that not all possible algorithms on the types provided are currently included in GEL; for instance, ray-casting involves following a single path through an octree, which is not a D&C operation[2]. Such algorithms can of course be programmed directly using the constructors of the types concerned. Similarly, operations between two tree structures (for instance addition and subtraction of octrees) cannot currently be directly described in GEL. Further work is needed to determine whether these and other computations could usefully be captured as higher-order functions.

D&C methods are also widely applicable in other areas of computational geometry. Striptrees and similar multi-resolution structures are already covered by an extension of the GDT type [7]. Other examples include the construction of Voronoi diagrams and Delaunay triangulation, and the determination of convex hulls. While these cannot be expressed using the specialised CSG and GDT operations, they can, in principle, be solved using the general *divacon* function.

It thus appears that the generic nature of D&C, and of geometric

[2]Recall, however, that the motivation behind GEL was to exploit parallelism through recurring algorithmic patterns, which explains the emphasis on D&C. By contrast, the single-path traversal noted above appears inherently sequential.

decomposition trees in particular, makes GEL applicable far beyond its initial aims.

7.2 A Complete Programming Paradigm?

It is clear that GEL does not cover the complete functionality required by applications making use of solid modelling. For instance, there is no support for the analysis or display of solid models. Hence GEL cannot be viewed as a general-purpose programming paradigm, even within the field of solid modelling.

Again, however, there is much scope for development beyond current limitations. Programming with skeletons is not restricted to a particular application domain. Skeletons remain an active research area, and many researchers are aiming to demonstrate an effective general-purpose paradigm. A number of general-purpose skeletons have been implemented, such as pipelines and farms. The *divacon* operation in GEL shows that such skeletons can be used alongside more specific operations, with the same potential to exploit parallelism. Thus the use of skeletons is not inherently limited to a narrow class of applications.

Arguably, however, application-specific skeletons have proved the most effective to date. They simplify programming through the use of specialised data types and may use detailed knowledge of the application to enable subtle optimisations within a compiler which application programmers may not realistically have the time to implement. This is demonstrated in the image processing language Apply, where programs generated from skeleton-like specifications often outperform hand-coded programs. Interestingly, recent work [13] has considered (parallel) skeletons for image display, suggesting that skeletons may have broader applicability within graphics.

7.3 Programming with skeletons

Programming with skeletons proved to have both benefits and drawbacks. There is a helpful discipline to facilitate program development, code is concise, and the low-level, error-prone geometric computations are isolated in a few customising functions. On the other hand, the limited set of operations may lead to more imaginative and appropriate solutions being missed, or incur inefficiencies such as repeating computations in *leaf* and *solve* functions [7]. Moreover,

'short-cut' solutions, such as 'early-outs' may not fit into the higher order function framework; Tilove's work on generic CSG algorithms also pointed to this conflict between generality and efficiency [19]. Adding extra CSG and GDT operations for special cases may partly resolve this, but would conflict th the simplicity of using a small set of generic operations, without guaranteeing that no more special cases will occur. Of course it is still possible to code such special cases directly in Hope+ or any other base language used.

7.4 Appropriate base language for skeletons

Reasons for using a functional base language for GEL were noted in section 4. Skeletons are not, however, restricted to this paradigm, and the merits of functional programming in this context should therefore be assessed.

The arguments for and against functional programming have frequently been rehearsed, and our experiences largely confirm established wisdom. Polymorphic types match the requirement to provide different geometric domains within the same generic structure. Higher-order functions successfully capture the computational structures initially identified and isolated low-level geometric details in customising functions. In these respects functional languages show an excellent match to the nature of the problem.

On the other hand, multilinked structures are perceived as being much harder to describe in functional languages [3]. This would become significant if GEL were extended to include B-reps, which require complex graph structures. Related difficulties were observed with the current version of GEL; for instance, we were unable to implement a *father-of* function to move back up a tree, precluding some of the efficient octree and quadtree traversal algorithms in the literature.

By reputation, functional languages are less efficient in runtime performance than imperative languages. We were therefore pleasantly surprised to find that GEL programs often outperformed the corresponding recursive C versions, a result of the efficient heap management techniques employed by Hope+, which aid fast tree construction. We have no evidence to suggest that Hope+ will generally outperform C over a wide spectrum of applications.

A second positive performance result was that the overheads of using higher-order functions (compared with a directly recursive

Hope+ version) were mostly low, suggesting that the higher level of abstraction provided by GEL was obtained without undue performance penalty. This is important evidence supporting the use of skeletons.

On the other hand, we found several very unpredictable aspects of Hope+ performance, making optimisation problematic. Details can be found in [7]. We have no specific evidence that such problems would occur with other functional languages, but our experience supports the view that performance prediction for functional programs is less tractable than with imperative languages.

7.5 Parallelism and GEL

While preliminary results for parallelising CSG trees are positive, much work remains to be done. More experiments are needed to understand the implications of tree shape on the performance of parallel CSG. We also intend to port our code to a true parallel multiprocessor, to confirm the performance gains and assess whether speedup is significantly affected by the particular parallel platform.

Further development of this work will require parallel GDT skeletons. Unlike CSG trees, the tree structures will be developed dynamically at runtime, so dynamic load balancing will be needed to ensure efficient implementation.

We surmise that parallelising the GDT operation should bring greater benefits than parallelising CSG, for two reasons. First, it is more general, since it allows other solid representations to be parallelised, notably B-reps [12]. Second it provides a coarser grain of parallelism and therefore greater performance gain; in Mistral-3 most speedup was obtained from the (outer loop) octree computations, with smaller gains from parallelising the (inner loop) CSG computation.

Probably the biggest challenge faced by parallel algorithmic skeletons is the effect of composition. Sequential composition occurs, for instance, when a *GDTcreate* skeleton is followed by *GDTtraverse*. Nesting occurs during octree creation, when a customising function for *GDTcreate* invokes *CSGtraverse*.

In a sequential system such as GEL, such composition causes no practical problems, but composition of independent parallel skeleton implementations faces some difficulties. In particular, 'stand-alone' implementations of skeletons invoked in sequence might not be eas-

ily compatible. For instance, the data distribution at the end of one skeleton may not match that required by a subsequent skeleton. Also, an implementation of a skeleton which presupposes sole use of the machine may not be applicable if the skeleton is nested within another partially executed skeleton, particularly if multiple instances of the nested skeleton are invoked in parallel.

To develop a complete parallel application-specific language in this field will require a suitable framework to combine skeletons. Possible approaches can be found in [1] and [6]. The former allows nested and sequential compositions of a small set of general-purpose skeletons, using C++-based syntax. In the latter a high-level declarative framework is used to write programs composed of skeletons defined in standard languages such as Fortran. This appears to combine the conceptual clarity and desirable formal properties of functional languages with the more pragmatic benefits of procedural languages. For GEL it seems particularly desirable to retain the polymorphism which enables varying geometric domains to be handled within the same algorithmic framework.

8 Conclusions

The case study has shown several clear benefits of programming CSG (and other) applications using algorithmic skeletons. Though there still remain a number of areas to explore, we have not yet found any decisive counter-evidence of the feasibility of the paradigm in the sequential world.

Skeleton programming may be used with either sequential or parallel computers. It seems, however, that the advantages will be greater in the latter case. There is far more low-level complexity to hide from the programmer, and the potential for saving development time is much greater. Minor inefficiences which might be incurred (noted in section 7.3) should be insignificant compared with the benefits of efficient pre-packaged parallel implementations.

The implementation of effective systems based on composing parallel skeletons still involves substantial research challenges. Thus, while the paradigm offers some promise, its ultimate potential as a serious development tool remains to be proved.

A functional language provided an elegant and natural base for prototyping GEL, but difficulties with multi-linked structures and

performance analysis have led us to pursue further developments in a more conventional imperative framework.

Acknowledgements

Thanks are due to Professor Alan de Pennington and the Geometric Modelling Project at Leeds for their support, to Professor John Darlington of Imperial College for supplying the Hope+ compiler used, and to Jon Nash for use of the WPRAM simulator. The second author was supported by a studentship from the Iranian government.

References

[1] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: a structured high-level parallel language, and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, May 1995.

[2] R Banerjee, V Goel, and A Mukherjee. Efficient parallel evaluation of CSG tree using fixed number of processors. In *Proc. 2nd ACM Symposium on Solid Modeling*, pages 137–145, May 1993.

[3] F. W. Burton and H-K Yang. Manipulating multilinked data structures in a pure functional language. *Software Practice and Experience*, 20(11):1167– 1185, November 1990.

[4] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman/MIT Press, 1989.

[5] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While. Parallel programming using skeleton functions. In *Proceedings of PARLE 93*, 1993.

[6] J. Darlington, Y. Guo, H. W. To, Q. Wu, J. Yang, and M. Kohler. Fortran-S: a uniform functional interface to parallel imperative languages. Imperial College London, October 1994.

[7] J. R. Davy and P. M. Dew. A polymorphic library for constructive solid geometry. *Journal of Functional Programming*, 1995. forthcoming.

[8] H. Deldari, J. R. Davy, and P. M. Dew. The performance of parallel algorithmic skeletons. In *Proceedings of ZEUS'95*, pages 65–74. IOS Press, May 1995.

[9] N. S. Holliman, D. T. Morris, and P. M. Dew. An evaluation of the processor farm model for visualising constructive solid geometry. In P. M. Dew, R. A. Earnshaw, and T. R. Heywood, editors, *Parallel Processing for Computer Vision and Display*, pages 452–460. Addison Wesley, 1989.

[10] N. S. Holliman, C. M. Wang, and P. M. Dew. Mistral-3: Parallel solid modelling. *The Visual Computer*, 9(7):356–370, July 1993.

[11] R. M. Karp and V. Ramachandran. Parallel Algorithms for Shared Memory Machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science : Volume A, Algorithms and Complexity*, pages 869–941. North Holland, 1990.

[12] A. Kela and M. Wynn. Parallel computation of exact quadtree/octree approximations. In *4th Conference on Hypercube Concurrent Computers and Applications*, Monterey California, March 1989.

[13] M. Kesseler. Constructing skeletons in Clean: the bare bones. In *Proceedings HPFC95*, 1995.

[14] Y. T. Lee and A. A. G. Requicha. Algorithms for computing the volume and other integral properties of solids. ii. a family of algorithms based on representation conversion and cellular approximation. *Communications of the ACM*, 25(9):642–650, September 1982.

[15] D. Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19:129–147, 1982.

[16] J. M. Nash, P. M. Dew, M. E. Dyer, and J. R. Davy. Parallel algorithm design on the WPRAM model. In J. R. Davy and P. M. Dew, editors, *Abstract Machine Models for Highly Parallel Computers*, pages 83–102. Oxford University Press, 1995.

[17] A. A. G. Requicha. Representations for rigid solids: Theory, methods and systems. *ACM Computing Surveys*, 12(4):437–464, December 1980.

[18] T. Ruppelt and G. Wirtz. Automatic transformation of high-level object-oriented specifications into parallel programs. *Parallel Computing*, 10:15–28, 1989.

[19] R. B. Tilove. Set membership classification: A unified approach to geometric intersection problems. *IEEE Transactions on Computers*, C-29(10):874–883, October 1980.

[20] H. Wang, P. M. Dew, and J. Webb. Implementation of Apply. *Concurrency: Practice and Experience*, 3(1):43–54, February 1991.

[21] J. A. Webb. Steps toward architecture independent image processing. *IEEE Computer*, 25(2):21–31, February 1992.