Activity Graphs: A Model-Independent Intermediate Layer for Skeletal Coordination

Murray Cole[1] and Andrea Zavanella[2]

[1] Institute for Computing Systems Architecture
University of Edinburgh
mic@dcs.ed.ac.uk
[2] Dipartimento di Informatica
Universitá di Pisa
zavanell@di.unipi.it

Abstract. Activity Graphs are introduced as a simple and practical means of capturing model independent aspects of the operational semantics of structured (and in particular, skeletal) parallel programs. AGs provide a notion of parallel activities, dependencies between activities, and the process groupings within which these take place. They are independent of low level details of parallel implementation and so can act as an intermediate layer for compilation to diverse underlying models. The paper introduces Activity Graphs and gives a compilation scheme from a simple skeletal language into them. The compilation process uses a set of graph generators (templates) to derive the Activity Graph.

1 Introduction

The skeletal approach to parallel programming [4, 8, 5, 2] advocates the use of program constructors, or "skeletons", which abstract useful patterns of parallel computation and interaction. This is held to ease the burden on the programmer, who is freed to think in terms of these higher-level strategies while being absolved of responsibility for their detailed implementation. The methodology also promises to encourage code re-use and to facilitate portability between architectures. While much progress has been made, existing work has a number of practical weaknesses:

1. Operational semantics of skeletal constructs tend to be presented with a high degree of informality. This leads to confusion over the precise meaning of nested programs, and also hinders meaningful comparison of skeletal languages and their relative expressiveness.
2. Compilation schemes tend to to be closely tied to language specific mechanisms and models. This is pragmatic, but is in conflict with the goal of simple portability: as much of the compilation process as possible should be model independent.
3. Fragments of base level code (i.e. the components which are co-ordinated by the skeletal layer) are constrained to be sequential. This is convenient from

the implementation perspective, but threatens to severely constrain applicability. Many (perhaps most) real parallel applications contain components whose structures do not fit the "skeletal straitjacket".

In this paper we present the "Activity Graph" as an intermediate level concept which addresses these issues. Our aim is to capture as much as can be said about the operational behaviour of skeletal programs while remaining generic across underlying models. In particular, we do not embed any assumptions on either the data or interaction model of the underlying parallel layer.

2 Our Skeleton Language

This section introduces the toy skeleton language L adopted within the paper to demonstrate the power and the simplicity of our approach. L is very simple in order to avoid distraction from the Activity Graphs which are the main subject of the paper. For example, we assume that the number of processors is a power of two. This is not a fundamental requirement and could be relaxed in a real language at the expense of a more complex definition. Similarly, we would expect a full language to incorporate other skeletal control constructs. The small set chosen here suffices to illustrate principles and to implement our chosen example.

The syntax of L is given in Fig. 1. It allows sequential composition, calls to base level functions and provides three parallel skeletons. **map** indicates concurrent, independent execution of the body statement: a group of p processors executing a **map** are dispersed into p single processor groups each executing the body independently. **div** constructs a binary tree of executions: a group of p processors executing a **div** first execute the body collectively as a group, then independently as two groups of size $\frac{p}{2}$, then as four groups of size $\frac{p}{4}$, and so on, finally executing the body as $\frac{p}{2}$ groups of size 2 (NB not p groups of size 1). **con** behaves symmetrically, executing the body on groups of size 2, then 4 and so on. Constructs can be nested arbitrarily, though notice that calling a **con** or a **div** within a map will have no effect, since the tree oriented constructors require at least two processors to activate the body). The *lowerlayer* defines

$$
\begin{array}{lll}
\mathit{program} & ::= & \mathit{statement}\ \ \mathit{lowerlayer} \\
\mathit{statement} & ::= & \mathit{statement}\ \ \mathit{statement}\ \ | \\
 & & \mathit{skeleton}\ \{\ \mathit{statement}\ \}\ | \\
 & & \mathit{basefn} \\
\mathit{skeleton} & ::= & \textbf{div}\ |\ \textbf{con}\ |\ \textbf{map}
\end{array}
$$

Fig. 1. The Language L

data structures and functions in the base language and $basefn$ corresponds to calls to these functions from L.

The definition of a *basefn* is provided by the programmer in the base language, augmented by two values obtainable at run time through reserved identifiers `range` (the size of the subgroup of processors executing the *basefn*) and `id` (the identifier of a processor within a subgroup). These can be used to specialise each processor's behaviour, in conventional SPMD style. Base functions will be written to be sequential or parallel to fit the context in which they are called. In our simple language, functions called within a **map** will be sequential (because **map** indicates a group size of one) while those called within **div** or **con** (but outside **map**) will be parallel (exploiting the unstructured parallel mechanisms of the base level).

As an example, we have selected a relatively complex parallel algorithm: the block-based bitonic mergesort. The algorithm was originally presented in [1], while the more realistic block-based (many items per processor) variant is discussed in [6] for example, to which the reader is referred for a discussion and justification of the algorithm itself. Very briefly, the algorithm is essentially a mergesort (hence the outer **con**) in which the mergestep involves some data re-organisation followed by a division process in which smaller and smaller subsequences are separated out (hence the inner **div**). The program can be expressed as a composition of our skeletons, essentially following the analysis made in [3]. The program is presented in Fig. 2. Notice that the semi-colons and parentheses belong to the base language syntax. Base function `quick_sort` is the usual sequential operation, `merge_split` is an internally parallel merging step and `reverse_half` is a parallel data redistribution step required to form bitonic sequences.

```
map {quick_sort(a);}
con {
  reverse_half(a);
  div {merge_split (a);}
}
```

Fig. 2. Bitonic Mergesort Program

We emphasise that there is no reference in L to data or its distribution. These matters are handled by the model specific lower layer, in conjunction with the functions which operate there. Thus, in the example, the meaning of `a` and the functions called to act upon it are a property of the base language. The purpose of the upper layer is to capture parallel algorithmic structure. For example, with C+MPI as the base language, the statement `map {quick_sort(a);}` simply means "run `quick_sort(a);` on each processor in the group". As is normal with MPI's SPMD style, the conception of the p local arrays `a` as a single nameless global array exists only in the programmer's head. The semantic nature of the constructs is analogous to that of control constructs in any imperative language, rather than that of pure higher-order functions.

3 Activity Graphs

Activity Graphs (AGs) express the coordination structure of groups of processors operating concurrently. They are designed to provide an intermediate layer for the process of skeletal program compilation, serving as a common, language (and model) independent target notation for the translation from purely skeletal code, and as the source notation for the language specific phase of base language code generation. In the former role they also provide a precise operational semantics for the skeletal layer, thereby enhancing the programmer's understanding of the language, and serving as a useful (and previously lacking) common ground for the comparison of diverse skeletal languages. The right hand half of figure 8 depicts the AG for our bitonic mergesort example on four processors indicating abbreviated names for the base language function calls and active processor ranges in parentheses.

The type of activity graphs is defined in Fig. 3, where *program* corresponds to a source program statement.

$$
\begin{array}{ll}
AG & :: \{V\} \times \{E\} \\
V & :: activity \times range \\
E & :: V \times V \\
range & :: nat \times nat \\
activity & :: AG + program
\end{array}
$$

Fig. 3. Activity Graph definition

Vertices correspond to activities which take place on contiguously indexed groups of processors. In a flattened AG, all activities will be function calls to the base level language but during the compilation process they may correspond to unexpanded skeletal constructs or to other self-contained activity graphs. An edge between two vertices indicates that there may be a dependency between the corresponding activities. The resolution of such dependencies falls to the base language dependent phase of compilation. Notice that no assumption is made on the underlying implementation model. Different implementation strategies (e.g. message passing or shared memory) can be instantiated as appropriate.

It is important to stress that while the definition allows arbitrary graph topology, the graphs which actually emerge from our intended usage will exhibit various regularities following the operational structure of the programming constructs they represent. Such properties will be exploited in the compilation phase from activity graph to base level parallelism.

4 Compiling Skeletons to AGs

In this section we explain the methodology for transforming a program written using our skeleton language into an AG once a range of processors is chosen. The AGs we obtain when compiling skeletons program are *flat*.

Definition 1. *An activity graph is said to be* flat *if all the activities at its vertices are base function calls.*

The compilation process exploits a set of rules one for each skeleton, named *graph generators*. The compilation algorithm takes the initial unexpanded activity graph $(\{(program,\ (0, p-1))\}, \{\ \})$ and recursively expands it by applying appropriate graph generators to unflattened vertices. Notice that the extension of the language with a new skeleton only requires the definition of a new graph generator. Thus, graph generators allow us to express the semantics of new skeletons in terms of "coordinations", in contrast to more abstract functional notations which do not fix such meta-implementation details.

In the remainder of the paper we will use the abstract syntactic object *seq* to indicate statement sequencing.

4.1 Graph Generators

A graph generator is a graph template which can be specialised with a given program and range in order to produce an AG:

$$G_{skel} :: program \times range \longrightarrow AG \tag{1}$$

The Graph Generator for sequential composition is described by Eq. 2.

$$G_{skel}(seq(p_1, p_2), range) = (\{v_1, v_2\}, \{(v_1, v_2)\}) \tag{2}$$

where:
$v_1 = (G_{skel}(p_1, range), range)$
$v_2 = (G_{skel}(p_2, range), range)$

Note that we do not attempt to introduce pipeline parallelism. Such a facility would require examination of cost information which is orthogonal to our purpose here. Such issues are considered in [7].

The conquer [3] paradigm is introduced in our language using the **con** skeleton. The generator of **con** is given by Eq. 3. Notice that the we define the leaves of the tree to be two-processor groups (rather than single processors). This follows naturally from the observation that in real situations, when the quantity of data far out strips the number of processors, it is common to use different

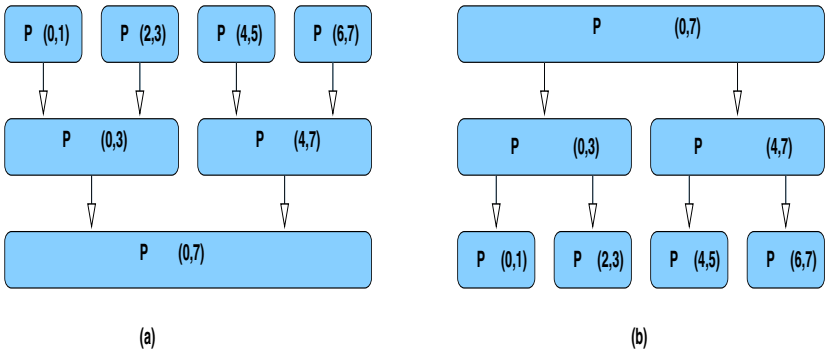

Fig. 4. A graphical representation of con (a) and div (b) graph generators

algorithms for the sequential "reduce within a processor" phase and the parallel tree reduction phase. Our **div** construct behaves analogously. Given that $|range| = 2^t$:

$$G_{skel}(con(p), range) = (V, E) \tag{3}$$

where:
$V = \{v_{ij}, r_{ij}\}$
$1 \leq i \leq t,\ 0 \leq j \leq 2^{t-i} - 1$
$v_{ij} = p,\ \ r_{ij} = (l, u)$
$l = j2^i,\ \ u = l + (2^i - 1)$
$E = \{(v_{ij}, v_{(i+1)(j/2)}) \mid 1 \leq i \leq t - 1\}$

The **con** skeleton is defined as construct to merge subgroups of activities using a standard binary tree. The **div** skeleton automatically generates the tree of a binary divide. The generator for **div** is the Eq. 4.

$$G_{skel}(div(p), range) = (V, E) \tag{4}$$

where:
$V = \{v_{ij}, r_{ij}\}$
$1 \leq i \leq t,\ 0 \leq j \leq 2^{i-1} - 1$
$v_{ij} = p,\ \ r_{ij} = (l, u)$
$l = j2^{t-i+1},\ \ u = l + (2^{t-i+1} - 1)$

$E = \{(v_{(i-1)(j/2)}, v_{ij}) \mid 2 \leq i \leq t\}$

The AGs for some graph generators are shown in Fig. 4.

Finally, the **map** skeleton models independent parallel replication. Its generator is given in Eq. 5.

$$G_{skel}(map(p), range) = (V, \{\}) \tag{5}$$

$$V = \{(p, (i, i)) \mid 0 \leq i \leq |range| - 1)\}$$

4.2 The Algorithm

The algorithm to compile a skeleton program into a flat AG starts from a trivial AG given by: $AG_0 = (\{(program, (0, p-1))\}, \{\})$ and recursively expands the nodes of the graph which contain program subtrees. When all nodes are *basefn* the compilation stops. A high level description of the algorithm is given in Fig. 5. We use `skel` to stand for any skeletal construct.

```
ag=AG0=(V,E)
While notflat(ag) do
      select v in V such that v=(skel(prog),r)
      ag'=Gskel(skel(prog),r)
      replace(ag,v,ag')                    /* node expansion */

      Forall e in E such that e=(v,x)      /* replace outgoing edges */
             delete(ag,e)
             newedges=connected(sinks(ag'),sources(x))
             add(ag,newedges)
      endfor

      Forall e in E such that e=(x,v)      /* replace incoming edges */
             delete(ag,e)
             newedges=connected(sinks(x),sources(ag'))
             add(ag,newedges)
      endfor
endwhile
```

Fig. 5. The compilation algorithm

The functions *notflat*, *sources*, *sinks* and *connected* are defined in Eq. 6- 9.

$$notflat(V, E) \equiv \exists (p, r) \in V : p = skel(prog) \tag{6}$$

$$connected(A, B) = \{(u, v)\} : u \in A, v \in B, \quad range(u) \cap range(v) \neq \emptyset \tag{7}$$

$$sinks(V, E) = \{v \in V : \forall (x, y) \in E \ : x \neq v\} \tag{8}$$

$$sources(V, E) = \{v \in V : \forall (x, y) \in E \ : y \neq v\} \tag{9}$$

The auxiliary functions *delete*, *add* and *replace* implement the intuitive operations on the graph of removing edges, adding edges and replacing a vertex with a subgraph (with the subsequent operations handle the connection of the new sub-graph to the whole).

5 Compiling Bitonic Mergesort

The compilation process may be better understood through our running example. Figure 6 shows the abstract syntax tree for the bitonic mergesort.

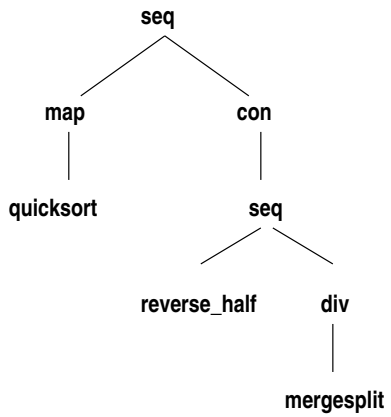

Fig. 6. The syntax tree of Bitonic Mergesort

Let us consider the four steps of the compiling process assuming that $p = 4$ and $r = (0, 3)$. The operations on the edges are shown in Fig. 7 (expansion of **seq** and then **map** and **con**) and Fig. 8 (expansion of **seq** and **div**).

step1 :
$v = seq\{prog1, prog2\}$
$prog1 = map\{quicksort(a)\}$
$prog2 = con\{seq\{reverse\_half(a), div\{merge\_split(a)\}\}\}$
$replace(ag, v, G_{skel}(seq(prog1, prog2), (0, 3)))$

step2 :
$v = map\{quicksort(a)\})$
$prog = quicksort(a)$
$replace(ag, v, G_{skel}(map(prog, r)))$
$v = con\{seq\{reverse\_half(a), div\{merge\_split(a)\}\}\}$
$prog = seq\{reverse\_half(a), div\{merge\_split(a)\}\}$
$replace(ag, v, G_{skel}(con(prog, r)))$

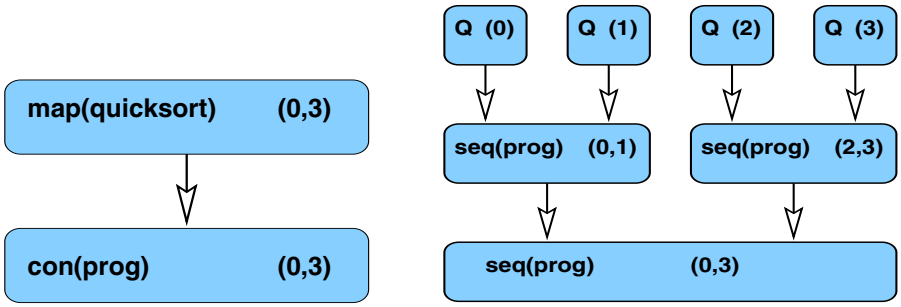

Fig. 7. Compilation of Bitonic: expansion of **seq** then **map** and **con**

step3 :
$v = seq\{prog1, prog2\}$
$prog1 = reverse\_half(a)$
$prog2 = div\{merge\_split(a)\}$
$replace(ag, v, G_{skel}(seq(prog1, prog2), r)))$

step4 :
$v = div\{prog\}$
$prog = merge\_split(a)$
$replace(ag, v, G_{Skel}(div(prog)))$

6 Conclusions and Future Work

We have defined the concept of "Activity Graphs" and have demonstrated their utility in the field of parallel program coordination, particularly in the skeletal style. We believe that activity graphs offer a precise formalism for the expression of the operational semantics of parallel program structures in a way which has previously been lacking, operating at a level which captures details of the structure of coordination, but independent of the model specific means by which that coordination may be implemented. We hope that this work will serve as a unifying foundation upon which we and others will build in the future.

There are many possible avenues for future development. Firstly, We have already made a preliminary study of the back-end process of compiling AGs to concrete low-level code, using MPI as a target. Space constraints preclude discussion of this work here but it will proceed. An obvious extension is to target further implementation layers, for example OpenMP, BSP or similar. Secondly, the skeletal language L described served as a demonstrator only. It will be most interesting to extend our approach to fuller, realistic languages. Finally, we have deliberately avoided issues of cost modelling and optimisation in this presentation, since we believe them to be orthogonal to our primary semantic

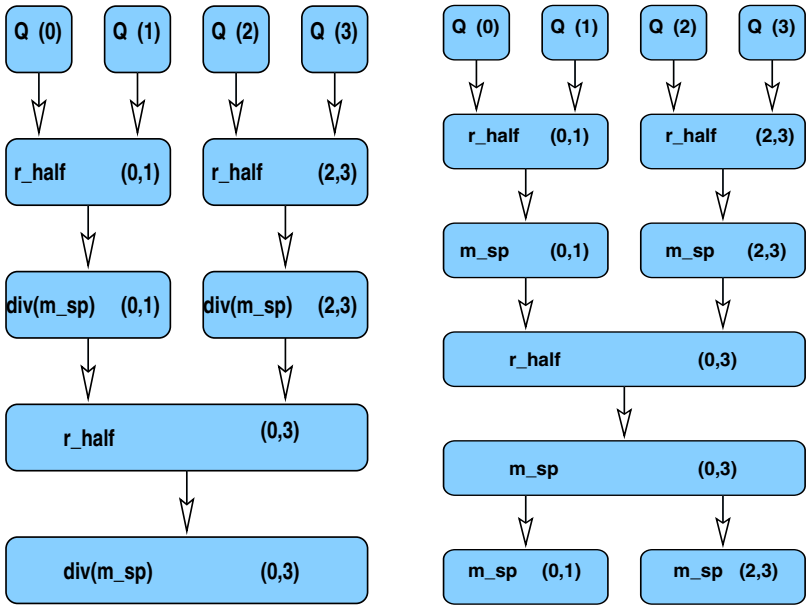

Fig. 8. Compilation of Bitonic: expansion of `seq` then `div`

purpose. However, we are aware that the structural information captured by our activity graphs should be useful in this respect.

References

1. K. Batcher. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Computer Conference*, pages 307–314, 1968.
2. George H. Botorog and Herbert Kuchen. Efficient parallel programming with algorithmic skeletons. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Proceedings of EuroPar '96*, volume 1123 of *LNCS*, pages 718–731. Springer, 1996.
3. M. Cole. On Dividing and Conquering Independently. In *Lecture Notes in Computer Science 1300*, pages 634–637, 1997.
4. M.I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Pitman, 1989.
5. J. Darlington, Y. Guo, H.W. To, and J. Yang. Parallel Skeletons for Structured Composition. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 19–28. ACM Press, 1995.
6. V. Kumar et al. *Introduction to Parallel Computing.* Benjamin Cummings, 1994.
7. H.W.To. Optimising the Parallel Behaviour of Combinations of Program Components. Ph.d. thesis, Imperial College, 1995.
8. S. Pelagatti and M. Danelutto. *Structured Development of Parallel Programs.* Taylor & Francis, 1997.