Journal of Systems Integration, 10, 127–143 (2001)
© 2001 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Coordinating Heterogeneous Parallel Systems with Skeletons and Activity Graphs

MURRAY COLE mic@dcs.ed.ac.uk
Institute for Computing Systems Architecture, Division of Informatics, University of Edinburgh, Scotland

ANDREA ZAVANELLA zavanell@di.unipi.it
Dipartimento di Informatica, Universitá di Pisa, Pisa, Italy

Abstract. Large scale parallel programming projects may become heterogeneous in both language and architectural model. We propose that skeletal programming techniques can alleviate some of the costs involved in designing and porting such programs, illustrating our approach with a simple program which combines shared memory and message passing code. We introduce Activity Graphs as a simple and practical means of capturing model independent aspects of the operational semantics of skeletal parallel programs. They are independent of low level details of parallel implementation and so can act as an intermediate layer for compilation to diverse underlying models. Activity graphs provide a notion of parallel activities, dependencies between activities, and the process groupings within which these take place. The compilation process uses a set of graph generators (templates) to derive the activity graph. We describe simple schemes for transforming activity graphs into message passing programs, targeting both MPI and BSP.

Keywords: skeleton, parallelism, activity graph

1. Introduction

The skeletal approach to parallel programming [3, 4, 6, 13] 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. As well as simplifying the initial coding of a parallel application, skeletons may offer significant help when porting to a new architecture is required. The nature of high performance computing makes this a regular and labor intensive exercise. Emerging standards such as MPI (for message passing) and OpenMP (for shared memory) have helped, but porting between such models remains problematic. Current trends towards ''cluster of SMP'' architectures (in which small and medium scale shared memory machines are connected to form larger ensembles) are most efficiently programmed in different language frameworks at different levels [12]. This only serves to further complicate porting. Abstracting the essential structure of a parallel application into a short composition of skeletal primitives, each supported by diverse implementations, offers the prospect of significant compiler support for traditional porting and also the ability to express multi-framework computations in a way which cleanly separates code which is framework independent from that which is framework specific.

We also speculate that the need for this style of programming may become more acute with the impending development of "computational grids" [8, 9] within the scientific programming community. Conceptually, a grid connects a number of diverse computational and storage resources (including diverse parallel computers) into a single resource, for the duration of a computation. Programmers will solve problems by using appropriate combinations of local and remote resources.

Current parallel programs are normally written to be portable between different sizes of the same machine by parameterizing with respect to p, the number of processors. This allows the same code to run on different sized sub-domains of the same machine on different days. Effectiveness relies on the fact that such domains tend to be similarly structured and so what is good on a small domain tends to be acceptable on a large one (though one can construct counter examples). Porting code to a new architecture is a major effort, if not a completely new endeavor. However, this happens infrequently enough (every few years) to be seen as cost-effective.

With grids, it will not just be p that varies, but also the component architectural structures and inter-site communications performance, and this will happen not year by year, but day by day, and in extreme circumstances, perhaps even during execution. For a given application program, radically different implementation strategies may be appropriate on different runs and perhaps even between phases of the same run. Programming such flexibility directly into (for example) an MPI framework will be very challenging given dramatically shorter time-scales. Automated support through clever compilation will become central. Program structuring mechanisms such as skeletons will allow the programmer to tell the compiler enough about the high-level opportunities for parallelism to make the task tractable while avoiding over specification of detailed implementation strategy. The programming model becomes one in which programs are structured at the top level with skeletal annotations coordinating conventional parallel or sequential code and in which there are a range of possible compile–execute paradigms:

1. recompile the same source for the "grid of the day";
2. compile once, but embed in the executable code which explores the available resources and selects from a number of possible schedules according to these (an extension of the current "parameterize by p" model);
3. as above, but compile in the ability to monitor and adapt the strategy in the face of changing resource availability or performance.

These are exciting opportunities which present technical challenges. In particular, while much progress has been made in skeletal programming, existing work has a number of practical weaknesses:

1. Operational semantics of skeletal constructs are often 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 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 coordinated 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 behavior 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. In Section 2 we introduce a simple skeleton language and show that it can describe programs which combine arbitrary forms of underlying parallelism. This section also introduces our running example. Section 3 introduces activity graphs while Section 4 describes the graph generators for the simple language and the front-end compilation process which calls them. Sections 6 and 7 outline back-end compilation processes from activity graphs to MPI and BSP respectively. Finally, Section 8 draws conclusions and proposes further work.

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 our main subject. 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 semantic definition. Similarly, we would expect a full language to incorporate other skeletal control constructs. Finally, in order to illustrate the possibility of dynamic choice between heterogeneous base levels (and architectures) we include a very simple mechanism which permits the programmer to specify control decisions based on the number active processors. A real language could be more flexible in this respect. The language defined here suffices only to illustrate principles and to implement our chosen example.

The syntax of L is given in Figure 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 $\frac{1}{2}p$, then as four groups of size $\frac{1}{4}p$, and so on, finally executing the body as $\frac{1}{2}p$ groups of size 2 (NB not p groups of size 1). **con** behaves symmetrically, executing

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

Figure 1. The language L

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 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 `mygrpsize` (the size of the subgroup of processors executing the *basefn*) and `myrank` (the identifier of a processor within a subgroup). These can be used to specialize each processor's behavior, 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). The `?` construct denotes choice between two base functions. Purely for the purposes of our example, its semantics are that `bigfn ? smallfn` will execute `bigfn` if the number of processors in the executing group is larger than the number of processors in a cluster of the executing architecture (assumed to be provided as a run-time constant by the system), and `smallfn` otherwise.

2.1. Homogeneous Programs

As an example, we have selected a relatively complex parallel algorithm: the block-based bitonic mergesort. In this section we express the algorithm in L with two alternative different base levels, expressed in MPI and the shared memory language Fork95 [10] respectively. In the next section we will show how the program can be adjusted for a "cluster of SMPs" architecture to use MPI between clusters and Fork95 within clusters.

The algorithm was originally presented in [1], while the more realistic block-based (many items per processor) variant is discussed in [11]. Very briefly, the algorithm is essentially a mergesort (hence the outer **con**) in which the merge step involves some data re-organization 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, following the analysis made in [5]. The top level of

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

Figure 2. Bitonic mergesort program.

```
void rev_half (void)
{
  if (myrank >= mygrpsize/2) {
    MPI_Sendrecv_replace(&a, n/p, MPI_INT,
                         mygrpsize-myrank+(mygrpsize/2)-1, 0,
                         mygrpsize-myrank+(mygrpsize/2)-1, 0,
                         mycomm, &mystat);
  }
}
```

Figure 3. Excerpt from base level in C/MPI.

the program is presented in Figure 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.

We emphasize 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);`} means "run `quick_sort(a);` on each processor in the group." As is normal with MPI's SPMD style, the conceptualization 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.

In MPI, the function `rev_half`, is implemented with a single collective communication as presented in Figure 3. Treating the data as a sequence of per-processor chunks, its effect is to reverse the sequence of chunks in the lower half of the data set (without reversing the chunks internally). In Fork95, the data is stored in an array shared memory and to achieve the same effect we arrange for the group of processors to re-arrange the lower half of the array directly, as depicted in Figure 4). In contrast to the MPI case, we can use all the processors in the group to achieve this, since all data is equally accessible to all processors (hence the absence of the

```
void rev_half(void)
{
    int i, myread, mywrite;

    mywrite = myrank*(n/(2*p));
    myread  = (mygrpsize-myrank-(myrank%2?0:2))*(n/(2*p));

    for (i=0; i<n/p; i++) {
      a[mywrite+i] = a[myread+i];
    }
}
```

Figure 4. Excerpt from base level in Fork95.

```
map {quick_sort(a);}
con {
     MPI_reverse_half(a); ? F95_reverse_half(a);
     div {
          MPI_merge_split (a); ? F95_merge_split (a);
     }
}
```

Figure 5. Heterogeneous bitonic mergesort program.

condition on `myrank`, the halved chunk size during copying and the index calculation for the start address of chunks). Fork95's synchronization semantics ensure that there are no race conditions during copying (pairs of processors are simultaneously exchanging the *i*th items in their chunks). The other base level computations can be expressed similarly in various parallel base languages.

2.2. Heterogeneous Programs

As noted in the introduction, architectural trends are towards hierarchical and heterogeneous systems, in which it may be appropriate to program on different paradigms at different levels. As a simple example, consider a two level "cluster of SMPs" machine, in which we would like to program in MPI between clusters and in Fork95 within clusters. Our model (with its admittedly tailor-made ? construct) allows such possibilities to be expressed concisely, and most importantly, *in a way which does not hide the overall algorithmic structure*, even across base layers. For bitonic mergesort, the required behavior is as expressed in Figure 5, where versions of base level specific functions have had their names prefixed to reflect the base language used.

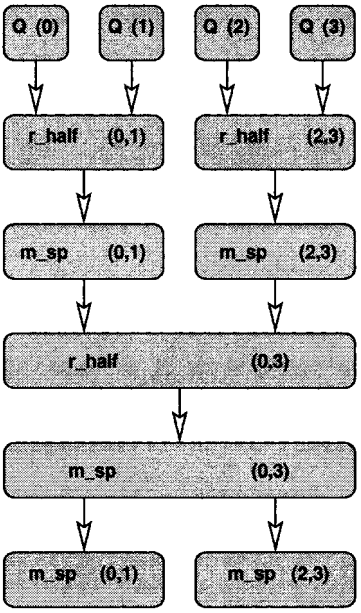

Figure 6. Expanded activity graph for bitonic mergesort.

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

Figure 7. Activity graph definition.

3. Activity Graphs

Activity graphs 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. Figure 6 depicts the activity graph 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 Figure 7, where *program* corresponds to a source program statement. Vertices correspond to activities which take place on

contiguously indexed groups of processors. In a flattened activity graph, all activities will be function calls to the base level language (or to run-time resolved choices between these using the ? operator) 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 Activity Graphs

In this section we explain the methodology for transforming a program written using our skeleton language into an activity graph once a range of processors is chosen. The activity graphs we obtain when compiling skeletons program are *flat* meaning that all the activities at its vertices are base function calls or choices between base function calls involving the ? operator from L.

The compilation process exploits a set of rules one for each skeleton, named *graph generators*. The compilation algorithm takes the initial, completely 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. Since the behavior of the skeleton is introduced using an intermediate layer, a new constructor does not require any change to the back-end compilation. This property can be exploited to refine the behavior of "generic" coordination patterns to fit with several classes of parallel algorithms: multi-grid, divide and conquer etc. 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 specialized with a given program and range in order to produce an activity graph:

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

The Graph Generator for sequential composition is described by Equation (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 [14].

The conquer [5] paradigm is introduced in our language using the **con** skeleton. The generator of **con** is given by Equation (3). Notice that 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 algorithms for the sequential "reduce within a processor" phase and the parallel tree reduction phase. Our **div** construct behaves analogously. The activity graphs generated for **div** and **con** are shown in Figures 8 and 9. Given that $|range| = 2^t$:

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

where:

$$V = \{v_{ij}, r_{ij}\}$$

$$1 \le i \le t, 0 \le j \le 2^{t-i} - 1$$

$$v_{ij} = p, r_{ij} = (l, u)$$

$$l = j2^i, u = l + (2^i - 1)$$

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

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 given in Equation (4).

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

where:

$$V = \{v_{ij}, r_{ij}\}$$

$$1 \le i \le t, 0 \le j \le 2^{i-1} - 1$$

$$v_{ij} = p, r_{ij} = (l, u)$$

$$l = j2^{t-i+1}, u = l + (2^{t-i+1} - 1)$$

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

```
ag=AG0=(V,E)
While notflat(ag) do
      select v in V with v=(skel(prog),r)
      ag'=Gskel(skel(prog),r)

      /* node expansion */
      replace(ag,v,ag')

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

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

Figure 10. The compilation algorithm.

Finally, the **map** skeleton models independent parallel replication. Its generator is given in Equation (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 activity graph starts from a trivial activity graph 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* or a choice between these, the compilation stops. A high level description of the algorithm is given in Figure 10. We use **skel** to stand for any skeletal construct.

The functions *notflat*, *sources*, *sinks* and *connected* are defined in Equations (6)–(9).

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

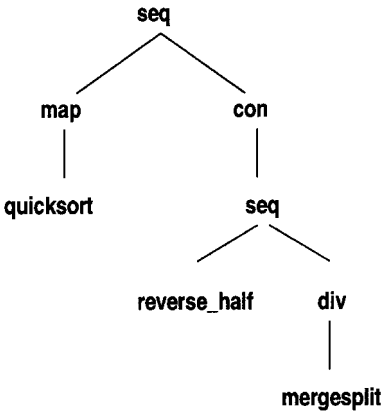

Figure 11. The syntax tree of Bitonic Mergesort.

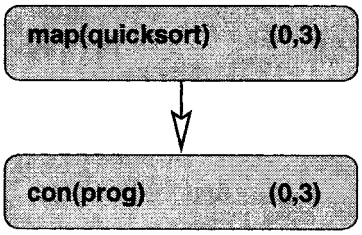

Figure 12. Expansion of **seq**.

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

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

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

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. Compilation Example

The compilation process may be better understood through our running example. Figure 11 shows the abstract syntax tree for the 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 Figure 12 (expansion of **seq**),

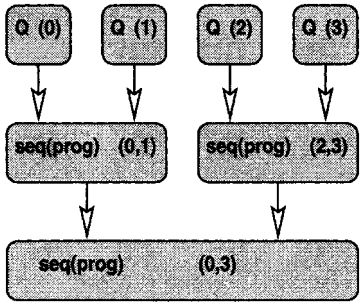

Figure 13. Expansion of **map** and **con**.

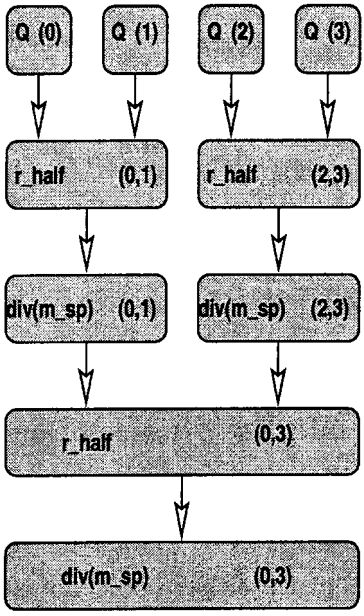

Figure 14. Expansion of **seq**.

Figure 13 (expansion of **map** and **con**) Figure 14, (next expansion of **seq**) and finally Figure 6 (expansion of **div**, to produce the flattened graph).

step 1:

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

step 2:

$v = map\{quicksort(a)\}$

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

step 3:
$v = seq\{prog1, prog2\}$
$prog1 = reverse\_half(a)$
$prog2 = div\{merge\_split(a)\}$
$replace(ag, v, G_{skel}(seq(prog1, prog2), r)))$

step 4:
$v = div\{prog\}$
$prog = merge\_split(a)$
$replace(ag, v, G_{skel}(div(prog)))$

6. Compiling Activity Graphs to MPI Programs

In this section we show how the structure of activity graphs can be exploited to derive a message passing implementation of the bitonic mergesort example. The technique is explained for the case in which modules are written using MPI and the final result of the compilation is an MPI program composed by user-defined code interleaved with *coordination-management* code, automatically generated.

Any such two level language scheme impose a number of requirements on the way in which base-level code may be expressed, in order to be able to give sensible semantics to complete programs. In the case of MPI, an obvious constraint is to require the activities of base level functions to be self-contained (so that processors in one group cannot interfere with those in another).

A simple enforcement mechanism would be to constrain all base level communications to occur within a group specific communicator (say `mycomm`) provided by the implementation, or sub-communicators thereof. We will assume that such a method has been chosen. Then, the operations to establish the corresponding `myrank` and `mygrpsize` will refer to `mycomm`.

An important aspect of the skeletal approach is that it constrains the forms of parallelism (and therefore the structures of the activity graphs) which can arise. For any particular language, we can exploit this knowledge in our compilation strategy. In the case of our very simple language L, the class of possible activity graphs is very simple: the graphs will be layered and within each layer i there will be a unique module f_i in execution on a set of different ranges having the same size s_i.

The focus of implementation in MPI is on using communicators to model the processor group structure required by the program. The corresponding compilation strategy makes use of a stack of communicators `com_stack` so that group contexts can be saved and restored dynamically as execution proceeds. Two higher level functions: `split_n` and `join_n` are defined on top of them. The two functions are

```
if (k[i]>k[i+1])
  split_n(stack, k[i]-k[i+1])
if (k[i+1]>k[i])
  join_n(stack, k[i+1]-k[i])
```

Figure 15. Generating the operations on communicators.

```
com_stack stack;

int main (int argc, char *argv[])
{
  MPI_Init(&argc, &argv);
  mycomm = MPI_COMM_WORLD;
  stack.c=MPI_COMM_WORLD;
  stack.p=1;

  split_n(stack, mycomm, 2);
  quicksort(a[], mycomm);
  join_n(stack, mycomm, 1);
  merge_split(a[], mycomm);
  join_n(stack, mycomm, 1);
  merge_split(a[], mycomm);
  split_n(stack, mycomm, 1);
  merge_split(a[], mycomm);

  MPI_Finalize();
}
```

Figure 16. MPI code for the Bitonic example.

employed to modify the communicator structure in the following way: for each layer i the compiler computes the exponent of subgroups size $k[i]$ such that $s_i = 2^{k[i]}$. The calls to modify the structure of communicators between layer i and layer $i+1$ are generated by the algorithm in Figure 15. Once the communicators structure has been modified the user-defined function f_{i+1} is called. The MPI program derived by the Bitonic Mergesort activity graph is shown in Figure 16, where compilation has assumed that the number of real processors available is 4, in order to match the situation illustrated in earlier figures. Of course, the generated code could easily be made parametric in the number of processors. For completeness, the communicator stack operations are presented in Figure 17.

7. Compiling Activity Graphs to BSP programs

In this section we show how the structure of activity graphs can be transformed to a BSP [15] program written using the Paderborn University BSP library (PUB) [2].

```
#include <mpi.h>
typedef struct {
          MPI_Comm c[MAXSTACK];
          int p;
   } com_stack;

void push(com_stack s, MPI_Comm myc){
          s.c[p]=myc;
          s.p++;
}
void pop(com_stack s, MPI_Comm myc){
          myc=s.c[p];
          s.p--;
}
void split_n(com_stack s, MPI_Comm myc,
                                    int n){
 int i;
 for (i=0;i<n;i++){
     MPI_Comm_rank(myc,&myrank);
     MPI_Comm_size(myc,&mysize);
     if(myrank<mysize/2)
        color=0
     else color=1;
     push(s,myc);
     MPI_Comm_Split(&myc,color,0,&myc)
 }
}
void join_n(com_stack s, MPI_Comm myc,
                                    int n){
 int i;
 for (i=0;i<n;i++)
      pop(s,myc);
}
```

Figure 17. MPI stack operations.

PUB is a BSP library extending the standard BSP model with the possibility of decomposing the parallel computers into subgroups of processors each operating independently. This feature is obtained by introducing BSP objects to distinguish the context in which the primitives are executed (similar to MPIs communicators). BSP objects can be created using the `bsp_partition` operation. After the execution of a `bsp_partition` each subgroup operates as an autonomous BSP computer (i.e., with its own processor numbering). The management of a BSP object is simpler than that of MPI communicators, indeed a stack of BSP objects is automatically maintained by the system and we can use `bsp_done` to resume the

```
void split_n(tbsp obj, tbsp newobj, int n){

int part[2],i;
for (i=0;i<n;i++){
    part[0]=bsp_nprocs(obj)/2;
    part[1]=bsp_nprocs(obj);
    bsp_partition(obj, newobj, 2, part);
    obj=newobj;
    }
}

void join_n(tbsp obj, int n){

int i;
for (i=0;i<n;i++){
    bsp_done(obj);
    }
}
```

Figure 18. PUB stack operations.

previous object from it. Therefore we can write the *split_n* and *join_n* primitives as illustrated in Figure 18:

The PUB library allows the programmer to write the base functions using C and the DRMA (Direct Remote Memory Access). Using the DRMA style is particularly suitable because a structure can be registered or deregistered as a shared area using the `bsp_push_reg` and `bsp_pop_reg` operations. A shared area can be written using `bsp_put` and `bsp_get`.

8. 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, as demonstrated, we have already made a preliminary study of the back-end process of compiling activity graphs to concrete low-level code, using MPI and BSP as a target. Obvious extensions are to automate this work and to target further implementation layers, for example OpenMP, 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 modeling and optimization in this presentation, since we believe them to be orthogonal to our primary semantic purpose. However, we are aware that the structural information captured by our activity graphs should be useful in this respect.

We are interested in the application of the Activity Graphs to the coordination of heterogeneous systems placed within a computational grid. In such a context we envisage compiling the graph generated from our skeletal program to some meta-description of the available resources. Assuming that a meta-communication layer is provided (as in the Nexus [7] framework) we can use it as target for the compilation. This approach seems to be suitable for managing the two-level heterogeneity arising from multiple communications layers (e.g., MPI, TCP/IP) and multiple computational resources (SMPs, MPPs and clusters) while maintaining portability and extensibility.

References

1. K. Batcher, "Sorting networks and their applications," in *Proc. AFIPS Spring Joint Computer Conference*, pp. 307–314, 1968.
2. O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping, "The Paderborn University BSP (PUB) Library—design, implementation, and performance," in *Proceeding of 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP), 1999.*
3. G. Botorog and H. Kuchen, "Efficient parallel programming with algorithmic skeltons," in L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert (eds.), *Proceedings of EuroPar '96*, Vol. 1123 of *LNCS* pp. 718–731, 1996.
4. M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation.* Pitman, London, 1989.
5. M. Cole, "On dividing and conquering independently," in *Lecture Notes in Computer Science 1300*, pp. 634–637, 1997.
6. J. Darlington, Y. Guo, H. To, and J. Yang, "Parallel skeletons for structured composition," in *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 19–28, 1995.
7. I. Foster, J. Geisler, W. Gropp, N. Karonis, E. Lusk, G. Thiruvathukal, and S. Tuecke, "A wide-area implementation of the message passing interface." *Parallel Computing* 24(12), 1998.
8. I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit." *International Journal of Supercomputing Applications* 11(2), pp. 115–128, 1997.
9. I. Foster and C. Kesselman, *The Grid*: *Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, San Francisco, 1998.
10. J. Keller, C. Kessler, and J. Traff, *Practical PRAM Programming*. Wiley, New York, 2001 (to appear).
11. V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing*. Benjamin Cummings, Redwood City, 1994.
12. S. Orlando, P. Palmerini, and R. Perego, "Coordinating HPF programs to mix task and data parallelism," in *Proceedings of 15th ACM Symposium on Applied Computing*, 1, pp. 240–247, 2000.
13. S. Pelagatti and M. Danelutto, *Structured Development of Parallel Programs*. Taylor and Francis, 1997.
14. H. To, "Optimising the parallel behaviour of combinations of program components," Ph.D. thesis, Department of Computing, Imperial College, 1995.
15. L. G. Valiant, "A bridging model for parallel computation." *Communications of the ACM* 33(8), p. 103, 1990.