Hierarchical Skeletons and *ad hoc* Parallelism

Marcus I. Marr and Murray Cole
Department of Computer Science, University of Edinburgh,
email: marr@dcs.ed.ac.uk

We investigate the controlled introduction of ad hoc, unstructured parallelism into the structured world of skeletal parallel programming. We outline the challenges raised and discuss an approach to their solution in the context of a small case study.

1 Introduction

Algorithmic skeletons[2, 3] promise to provide flexible, portable and efficient structuring of parallel programs. Early work in the area [2] concentrated on one-per-program skeletons, in which the computations "fleshing out" the structure were specified and computed sequentially. Subsequent work has begun to address the complications imposed by allowing composition (so that a program may employ a sequence of skeletons) and nestability (so that one skeleton may be fleshed out by another, and so on) [7, 1, 6, 8].

In this poster we focus on the exploration of a new dimension, in which the fleshing out can be expressed with ad hoc (but self-contained) parallel code, programmed in the base model of the actual machine. In this style, programs remain portable at the skeleton level, but require recoding of the ad hoc base code for each new machine. We are motivated by the likelihood that pure skeletal programming will encounter some situations in which a clear, but non-skeletal solution to some sub-problem is most appropriate. Such solutions can then be encompassed in a controlled way within a wider skeletal framework.

In the following sections we introduce our model more precisely and outline the main challenges it raises. We then present a case study which highlights these issues in the context of a real example.

2 The Model

As in traditional skeletal programming the core of our model is a collection of conventional skeletons such as *pipe*, *map*, *etc.* A program consists of compositions and nestings of these skeletons, with the leaves of the hierarchy expressed in the explicitly parallel base language of the target machine.

The most novel aspect of our approach is the controlled introduction of unstructured explicit parallelism to the skeletal framework and it is at the interface between these two worlds that the main challenges arise. The pure skeletal levels of a program must be independent of the base model, which implies that from the skeletal side, the language mechanism for passing data to the base level computations must be fixed. Meanwhile, from the base language side the mechanisms will necessarily look rather different. In the following case study we illustrate possible mechanisms for both shared memory and distributed memory architectures.

3 A Case Study

To illustrate the division and interface between the skeletal superstructure and the ad hoc leaves, a simple three stage pipeline example will be used. The pipeline takes a list of arrays and performs a series of averaging operations on each array.

The code for the superstructure is written in an architecture independent 'skeleton language', while the code for the leaf groups is written firstly in Fork95[5] for a shared memory PRAM, and secondly in C with MPI[4] for a distributed memory machine. The example illustrates two mechanisms used for passing data between the superstructure and the base level, and also how the data is handled by the base level computations.

3.1 The Pipeline Superstructure

The top level code for the problem declares the input and output data structures for the pipeline, and then specifies the skeleton hierarchy, which in this case is just the pipeline skeleton:

$$
\begin{array}{rcl}
\text{inputList} & : & \text{list of array of integer;} \\
\text{outputList} & : & \text{list of array of integer;} \\
\text{averageFn} & : & \text{array of integer} \rightarrow \text{array of integer;} \\
\text{stageList} & : & \text{list of (array of integer} \rightarrow \text{array of integer);} \\
\text{stageList} & := & \{\text{ averageFn, averageFn, averageFn }\}; \\
\text{outputList} & := & \text{HSM\_Pipeline(inputList, stageList);}
\end{array}
$$

For simplicity, all the arrays will be 1000 elements long, and the boundary problems for the first and last elements in the array will be ignored. The code for the leaf node `averageFn()` will now be described for both base languages.

3.2 The Shared Memory Base Level

Using the shared arrays of the Fork95 language, all the processors in a leaf node of the hierarchy have both read and write access to the data. Data can be passed from the skeleton superstructure to the base levels simply by using parameter passing. It is left to the ad hoc code to partition the work between the processors, which in this example is done using a strided partition on the output array.

```
sh int* average(sh int *ibuf) {
    for (i=HSM_Leafgroup_rank(); i<1000; i+=HSM_Leafgroup_size()) {
        obuf[i]=(ibuf[i-1] + ibuf[i] + ibuf[i+1])/3;
    }
    return obuf;
}
```

Although the code avoids write conflicts, the model allows the programmer to exploit machine dependent conflict resolution strategies in a controlled manner.

3.3 The Distributed Memory Base Level

Unlike the shared memory implementation, a distributed memory implementation must take into account the location and ownership of the data passed from the skeleton superstructure to the ad hoc code. The simplest mechanism involves passing all the data to one 'root' processor in the node and leaving it to the programmer to distribute the data to the other processors. Before completion of the node all results must be gathered to the root processor.

To perform the averaging operation, the input array is divided into contiguous blocks which are then distributed by the root to the other processors using a collective *scatter* communication. All the processors then send and receive the boundary values to their neighbours using point-to-point *send-receive* communications. The output blocks are then calculated and collected back at the root processor using a collective *gather* communication.

```
int* average(int *ibuf) {
    MPI_Scatter(ibuf, ..., HSM_Leafgroup_comm());
    MPI_SendRecv(ibuf[0], &leftval, HSM_Leafgroup_rank()-1, ...);
    MPI_SendRecv(ibuf[last-1], &rightval, HSM_Leafgroup_rank()+1, ...);
    obuf[0]=(leftval + ibuf[0] + ibuf[1])/3;
    for (i=1; i<last-1; i++) {
        obuf[i]=(ibuf[i-1] + ibuf[i] + ibuf[i+1])/3;
    }
    obuf[i]=(ibuf[i-1] + ibuf[i] + rightval)/3;
    MPI_Gather(obuf, ..., HSM_Leafgroup_comm());
    return obuf;
}
```

The disadvantage of passing data from the skeleton superstructure to only the root processor is that the programmer's task is made more complicated. If the nature of the distribution required by the function could be included in the function's type definition, the skeleton could re-distribute the data automatically. There would also be scope for optimising the re-distribution based on each target architecture.

4 Summary & Conclusion

We have shown that ad-hoc parallelism can be integrated at the lowest level of a skeletal programming hierarchy in a language independent way. In a message passing environment the programmer can use the full set of explicit communication tools to distribute data to where it is needed in the leaf nodes, but there is also scope for automatically distributing the data.

The *pipeline* skeleton described here is one of the simplest skeletons to consider with only a single direction of flow between sub-groups. The inheritance of input and output data in the *map* and *divide-and-conquer* skeleton is very similar to that in the *pipeline* skeleton, but there are many more skeletons to consider.

The nesting of skeletons also raises interesting questions on the optimal partitioning of processors among the subgroups since it is likely that there will be fewer processors than there is parallelism to exploit. A machine independent performance model for each skeleton, parameterised by machine computation and communication properties, would be able to suggest possible partitioning strategies.

References

[1] Tore Andreas Bratvold. *Skeleton-Based Parallelisation of Functional Programs.* PhD thesis, Heriot-Watt University, November 1995.

[2] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, 1989.

[3] J. Darlington *et al.* Parallel Programming using Skeleton Functions. In *Proc. of PARLE'93*, volume 694 of *LNCS*, pages 146–160. Springer-Verlag, 1993.

[4] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, May 5 1994.

[5] C W Kessler and H Seidl. Fork95 language and compiler for the SB-PRAM. In *5th International Workshop on Compilers for Parallel Computers*, June 28-30 1995.

[6] S. Pelagatti. *A Methodology for the Development and the Support of Massively Parallel Programs.* PhD thesis, TD-11/93, Department of Computer Science, University of Pisa (Italy), 1993.

[7] Roopa Rangaswami. HOPP - a Higher Order Parallel Programming Model. In Marc Moonen, editor, *Algorithms and Parallel VLSI Architectures.* Elsevier, 1995.

[8] David B. Skillicorn. *Foundations of Parallel Programming.* Cambridge University Press, 1994.