Dividing and Conquering

Murray Cole

Department of Computer Science, University of Edinburgh.

e-mail: mic@dcs.ed.ac.uk

Abstract

We suggest that the components of the well known *divide-and-conquer* paradigm can be profitably presented as independent constructs in a skeletal parallel programming model. We investigate this proposal in the context of a rendition of Batcher's bitonic sorting algorithm in which the nested parallel structure of the program is neatly abstracted from the low level detail of a flat presentation. We show that a variant of the conquer construct can make the presentation neater still, and discuss the implications for the design of skeletal models.

1 Introduction

The *skeletal* approach to parallel programming [4, 5] proposes that useful and recurring forms of parallel program structure should be abstracted as second order control constructs (usually expressed in a functional notation) and that the programmer's task should be to compose and nest instances of these constructs to express the parallelism available in a computation in a controlled and well-structured manner. In its strictest form [5], the methodology requires programs to be expressed on two levels, an outer level, dealing exclusively with skeletal forms, and an inner level, specializing the skeletons with application specific, sequential code. The implementation task is then simplified to the composition of the pre-defined structures with ad-hoc, unstructured parallelism explicitly disallowed. The restriction is both a strength (from the perspective of efficient implementability) and a potential weakness, (in that without a suitable set of primitive skeletons, algorithm expression will be hindered). Thus, one of the central challenges in the design of skeletal programming systems concerns the correct choice of skeletal primitives. These should be generic enough to allow the expression of a wide range of algorithms from a small basis of constructs, but should minimize the need for code which is conceptually redundant but syntactically necessary.

Most skeletal programs presented in the literature involve either flat compositions of skeletons or nested structures in which the outer structure is a simple 'map' with the consequent mutual independence of inner level structures (indeed the NESL language [3] restricts nesting so that *only* 'map' parallelism can be expressed at all but the innermost level). In this paper, we present a three level skeletal rendition of Batcher's bitonic sorting algorithm in which the top two levels involve non-trivial parallelism. We believe (albeit subjectively) that such a presentation brings out the algorithmic structure more clearly than its conventional diagrammatic expression or the equivalent flat parallel program. From the skeletal perspective, the example is of interest in that the nested structures are naturally expressed as independent instances of 'divide' and 'conquer' skeletons in which the corresponding 'conquer' and 'divide' phases of a conventional 'divide-and-conquer' presentation would be conceptually null. This suggests that these two structures might usefully be presented as independent constructs in skeletal programming systems. Since 'conquer' is already otherwise known as 'fold' or 'reduce' (given certain algebraic constraints), we believe that some operation analogous to "divide" should have a similar status and exposure. We then present a second rendition of the program, in a similar style, but exploiting a generalized version of the divide primitive admitting a more faithful representation of the original algorithm.

2 Bitonic Sort

Batcher's bitonic sorting network was originally published in [2] and has become one of the best known examples of intricate parallel algorithm design, with reworkings for a variety of networks including shuffle-exchange, hypercube and mesh. In this section we review the key elements of the algorithm and comment on standard presentations thereof. The unfamiliar reader is referred to any standard parallel algorithms text (for example [7]) for more

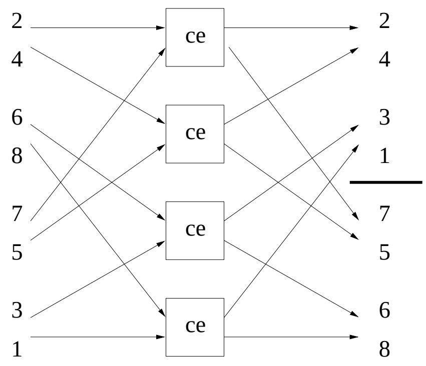

Figure 1: Bitonic merge step

detailed discussion. As is customary, we will assume that the number of items to be sorted is an exact power of two.

A bitonic sequence consists of two portions, one ascending the other descending (or can be cyclically shifted to have this form). Take such a sequence, shuffle it and compare-exchange the adjacent pairs. Now considered the sequence of all ‘losers’ of the comparisons and the sequence of all ‘winners’ independently. The two key observations (illustrated by example in figure 1) are that

1. these sequences are both themselves bitonic

2. none of the losers is larger than any of the winners.

To sort the data, it remains to independently sort the losers and the winners, concatenating the results (because of 2). The independent sorts can be achieved recursively (because of 1), with the obvious base case.

We now know how to sort a bitonic sequence, but of course we want to sort arbitrary sequences. The final observation is that we can create a bitonic sequence from an arbitrary one by sorting half the items into ascending order and the other half into descending order. Thus, there is a mutual recursion in the full definition - we sort arbitrary sequences with help of a bitonic sorter, which itself requires the assistance of smaller arbitrary sorters. At this point, conventional presentations tend to resort to diagrams which flatten the required communication structure into complex diagrams in the style of of figure 1 and to unintuitive bit-twiddling code fragments to determine the ‘parity’ (low-high or high-low) of the compare-exchange units required to produce appropriate ascending and descending sub-sequences.

Related papers, investigating the expressivity of functional programming notations as sources of parallelism, have noted that the algorithm can be expressed in a nested divide-and-conquer form. In particular Mou and Hudak [9] present the algorithm in terms of their 'divacon' construct, while Misra presents the bitonic merge component of the algorithm in terms of recursive operations on 'powerlists' (in which the constructors, designed with the expression of divide-and-conquer algorithms in mind, ensure that lists are always of length which is a power of two).

In the next section we propose that the phases of a conventional divide and conquer primitive be provided independently and then exploit them to present a skeletal, formulation of Batcher's algorithm, following the style of [9], but with the removal of operations which are conceptually (i.e. at the algorithmic level) redundant.

3 Separating Divide and Conquer

The divide-and-conquer paradigm has a long history parallel programming and algorithm design (for example [1, 6, 9] and many others). Conventional presentations require the programmer to specify both a divide operator and a combine operator, as well as test for the base case and a function to be applied when that case arises.

```
d&c triv base div com x =
      if triv x then base x
      else com (map (d&c triv base div com) (div x))
```

From a skeletal perspective, the flexibility of such a definition is simultaneously attractive and daunting. To the programmer, the unrestricted degree of the division process looks convenient. To the system, it presents a dynamic scheduling problem. Similarly, the need to specify both divide and conquer phases is often, in conceptual terms, an unnecessary chore, essential to make the program consistent, but contributing nothing to the clarity of the exposition. For example, the divide operation in a standard mergesort, serves simply to transform the list into a list singletons for processing by the conquer phase which does the 'real' work. In a parallel skeletal system, this extra 'redundant' computation would at best complicate the compilation process, and at worst impair the achieved run-time, by faithfully scheduling and executing the recursive partitioning.

Our main intention in this paper is to argue that these are overheads we cannot afford in the efficiency conscious context of parallel programming. While there may be a place for fully flexible divide-and-conquer skeleton, it should only be exploited when strictly necessary and a simpler set of primitives can provide equivalent and often more appropriate expressivity without unnecessarily cluttering the code and, more importantly, the compilation and run-time systems.

Our suggestion is as follows. Firstly, following [8], we believe that the 'powerlist' restriction is acceptable for many good parallel algorithms and we will assume that it holds for all the lists in the ensuing discussion. More importantly, we propose that the divide and conquer phases be presented as independent 'skeletal' operations.

3.1 Conquer

Our `conquer` is more or less standard, in that it is essentially a 'fold' (or 'reduce') operation restricted in such a way that the usual parallel tree evaluation is actually enforced (rather than being optional). Semantically, its behaviour can be defined precisely as

```
onelevel f [] = []
onelevel f (x::(y::xs)) = (f x y)::(onelevel f xs)

conquer f [x]        = x
conquer f (x::y::xs) = (conquer f  o  onelevel f) (x::y::xs)
```

with the 'powerlist' assumption ensuring that the 'missing' patterns are not required. We should emphasize that we use this sequential code only as a semantic rather than an operational specification. More intuitively (and informally) it is defined by figure 2, where it is crucial to remember that each instance of `f` may itself be parallel.

3.2 Divide

Our `divide` skeleton is less conventional. Its semantics are defined formally as

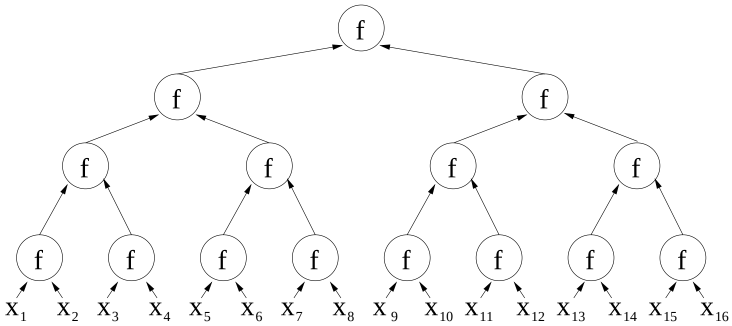

Figure 2: Operation of conquer

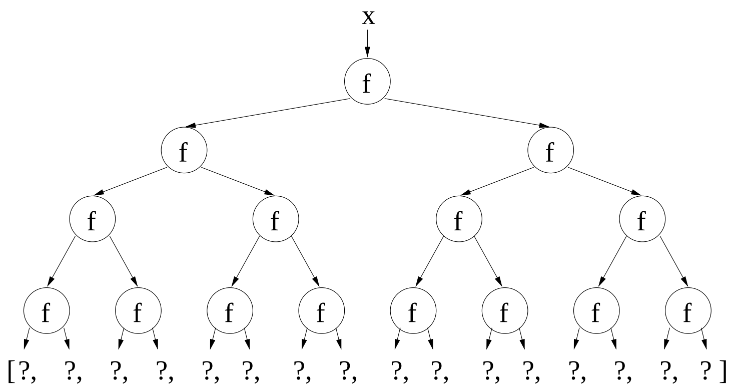

Figure 3: Operation of divide

```
divide f x = if singleton x then [x]
             else (( flatten o (map (divide f)) o ptol o f) x

ptol (x,y) = [x,y]
```

where `flatten` removes one level of sequencing, and informally by figure 3. Notice that we use the type system to require the dividing operation `f` to be of type $\alpha \to (\alpha, \alpha)$.

4 A Nested Parallel Rendition of Bitonic Sort

We now present Batcher's algorithm in terms of the new primitives. At this level, the program essentially follows that of [9], but with the isolation of redundant divide and conquer phases. We begin at the outer level with the simple observation that the process involved is that of the conquer phase of a traditional divide and conquer algorithm. The conquer step is to produce a single sorted sequence from two sorted sequences of half the length[1]. The fact that one of these sequences should be reversed is not significant here and will be handled at an inner level (although as we will address later, this represents a slight divergence from the real essence of Batcher's algorithm). Thus, at this level our program has the form

```
bitsort = conquer bitmerge o map (fn x => [x])
```

In Mou's presentation, the `map` phase (which as we have noted is conceptually redundant) is coded into the divide-and-conquer call and given similar apparent status to the `bitmerge` which does the real work. It would be implicit in a directly programmed imperative version. We speculate (though do not pursue here) that it might be possible to infer, in a suitably restricted skeletal system, rather in the manner of an implicit type cast, thereby leaving it altogether unstated, or at the very least, to have it as a routine predefined data manipulation (like ML's `explode`).

As we have seen, the utility of the algorithm is that `bitmerge` can also be parallelized. The key here is to view the inner level computation described in section 2 as a 'dividing' operation which creates two sequences from one. Applied recursively (with the recursion wrapped up in our second skeletal construct), it turns a sequence into a sequence of singleton sequences, with property 2 above showing that these are sorted. It only remains to flatten out one level of sequencing (as above, this would be implicit in an imperative formulation). At the third and innermost level, the dividing operation itself is expressed through a simple map.

Thus, we can instantiate 'divide' to describe the behaviour of the bitonic merge algorithm as

```
bitmerge xs ys
 = (flatten o divide (bmstep ce)) (xs@(reverse ys))

ce (x:int,y) = if x<y then (x,y) else (y,x)
```

[1] Just like mergesort, and indeed, the original algorithm is often referred to as bitonic mergesort.

```
bmstep ce = unzip o (map ce) o zip o split
```

where `split` splits a sequence in two and `unzip` is effectively an inverse of zip, but returning the two zipped sequences concatenated as a single sequence.

Notice that the inner level requires some pre and post processing to reorganize the data - some of this is algorithmic (for example the shuffle, expressed here as a zip), some (for example the flattening of the resulting list and the `split`) is again essentially trivial.

The essence of the algorithm, that it is a conquer with an operator which divides with an operator which maps, is now clear from the definitions of `bitsort`, `bitmerge` and `bmstep`, and would be even clearer if some of the data manipulations programmed explicitly in this version were provided as built-in operations.

We note that second input to the inner phase must be reversed before processing. This both complicates the code (and thereby the language and implementation process) and is a clumsy expression of one of the key ideas in the algorithm, that the input to the merge operation should be bitonic. In the next section, we present an alternative `conquer` skeleton which allows this idea to be expressed more naturally.

5 A Generalized Conquer

Our second conquer construct (from which the `conquer` above can be specialized) allows left and right sub-trees to (recursively) apply distinct combining operators. In bitonic sort, this can be exploited (in place of the standard 'merging' structure employed above) to capture the idea that matching pairs of sequences within the merge should be alternately sorted into ascending and descending order (so that their concatenation is bitonic). Informally, this `conquer'` is described by figure 4, and can be more formally specified as

```
onelevel' f g [] = []
onelevel' f g (x::(y::xs)) = (f x y)::(onelevel' g f xs)

conquer' f g [x]        = x
conquer' f g (x::y::xs) = (conquer' f g o  onelevel' f g) (x::y::xs)
```

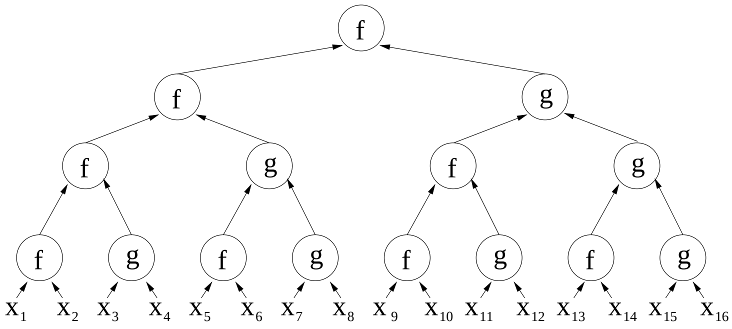

Figure 4: Operation of `conquer'`

We can now re-express the sorting algorithm as

```
bitsort'
  = conquer' (bitmerge' (bmstep ceup)) (bitmerge' (bmstep cedown))
    o (fn x => [x])


bitmerge' f xs ys = ( flatten o divide f) (xs@ys)
```

where `ceup` is just the `ce` we had earlier and `cedown` produces descending outputs, so that when passed into the two instantiations of `bitmerge'`, we obtain ascending and descending mergers as appropriate. Notice that the unnatural 'reverse' is no longer required.

6 Observations

Since we have earlier argued against over-generality in skeletal constructs, we should comment on the desirability of introducing our own generalized `conquer'`. To the programmer, the fact that `conquer f = conquer' f f` means that the more specialized form could easily be predefined as a specialization of the latter. Of course the same could be said of a pre-packaged `d&c` primitive specialized to the cases in which one or other phase is trivial. More importantly, definition of `conquer` in terms of `conquer'` implies no additional complication to the implementation mechanism - it simply determines which operations should be applied at which predictable points in the evaluation tree. In contrast, source level specialization of full divide-and-conquer risks extra work unless accompanied by a complementary specialization of the implementation.

References

[1] Axford, T. *"The divide-and-conquer paradigm as a basis for parallel language design"*, in Advances in Parallel Algorithms, Kronsjo L. and Shumsheruddin D. (Eds), Blackwell Scientific, 1992.

[2] Batcher, K. *"Sorting Networks and their Applications"*, in Proc. AFIPS Spring Joint Computer Conference, volume 32, pages 307-314, 1968.

[3] Blelloch G.E.. *"Programming Parallel Algorithms"*, in Communications of the ACM, volume 39, number 3, 1996.

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

[5] Darlington, J. and Guo, Y. and To, H.W. and Yang, J., *"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] Horowitz, E. and Zorat, A., *"Divide-and-Conquer for Parallel Processing"* , IEEE Transactions on Computers, volume 32, number 6, pages 582-585, 1983.

[7] Kumar, V. and Grama, A. and Gupta, A. and Karypis G. *"Introduction to Parallel Computing: Design and Analysis of Algorithms"*, Benjamin/Cummings, 1994.

[8] Misra, J. *"Powerlist: A Structure for Parallel Recursion"*, in A Classical Mind: Essays in Honour of C. A. R. Hoare, Prentice Hall, 1994.

[9] Mou Z.G. and Hudak, P. *"An Algebraic Model for Divide and Conquer and its Parallelism"*, Journal of Supercomputing, volume 2, pages 257-278, 1988.