A HIGH-LEVEL, OBJECT-ORIENTED APPROACH TO DIVIDE-AND-CONQUER

A. J. Piper & R. W. Prager

CUED/F-INFENG/TR 98

April 1992

Cambridge University Engineering Department
Trumpington Street
Cambridge CB2 1PZ
England

Email: ajp/rwp @eng.cam.ac.uk

A High-Level, Object-Oriented Approach to Divide-and-Conquer

A. J. Piper and R. W. Prager
Cambridge University Engineering Department,
Trumpington Street, Cambridge, CB2 1PZ, England
email: ajp@uk.ac.cam.eng

May 27, 1993

Abstract

An object-oriented framework for the divide-and-conquer paradigm is presented. This framework does not require a parallelising compiler and thus provides an environment that is flexible and easily extensible. The framework enables a divide-and-conquer representation of a problem to be built up for subsequent evaluation. Evaluation is delayed until the maximum amount of computation that can be performed in one divide-and-conquer pass has been integrated into the representation. Results are presented for two different implementations of the back-propagation algorithm. Divide-and-conquer provides a flexible framework for the parallel implementation of various algorithms. Object-oriented programming techniques provide a means to encapsulate the divide-and-conquer semantics and provide a uniform interface to the end-user. This results in a reasonably efficient system for parallel problem solving which is easy to use and useful in a wide range of tasks.

1 Introduction

Parallel processing has the potential to deliver cost effective computational power. However, if the benefits of parallel processing are to be experienced by all computer programmers, then the difficulties in implementing parallel programs need to be tackled. Two common approaches to the problems of achieving this accessible parallelism have emerged. The first approach involves producing a custom language for which all language constructs can be easily implemented in parallel - for instance functional languages. The second approach involves taking any serial specification and extracting parallelism from it using a compiler. These two approaches both fail to some degree because of their extremeness. Custom languages that are easily parallelised are not easily programmed. The compiler benefits, the programmer does not. Parallel feature extraction is difficult and inefficient at best, and limited by the number of parallel features present in a program. At worst it is impossible. The programmer benefits at the expense of parallelism.

Divide-and-conquer [1] in conjunction with object-oriented programming [23] may offer a middle ground which achieves the best of both worlds. Divide-and-conquer is a well studied computational paradigm that can be easily and efficiently parallelised. Object-oriented programming allows the complexity of the parallelism provided by divide-and-conquer to be hidden from the programmer. At the same time the programmer can benefit from the features of object-oriented programming in general. This paper presents an object-oriented framework for divide-and-conquer and describes the benefits of the object-oriented approach.

The next section describes the divide-and-conquer paradigm and it parallel implementation.

2 Divide-and-Conquer

Divide-and-conquer has been most widely used in a graph reduction context with functional language evaluation [18]. However, divide-and-conquer can also be used as a programming paradigm in its own right as set out by Axford [3] and Mou [17, 16].

The divide-and-conquer algorithm can be represented in pseudocode as follows:

FUNCTION *Divacon* (data)
BEGIN
    IF *isSimple* (data)
    THEN RETURN *Function* (data)
    ELSE *Combine* (*Divacon* (*Divide* (data)))
END

where *Divide()* divides a task into sub-tasks, *isSimple()* specifies if the task is small enough to compute, *Combine()* combines partial results and *Function()* yields a partial result from a sub-task.

Many common algorithms - merge sort, matrix multiplication, fibonacci sequence etc - have been represented concisely in a divide-and-conquer format and implemented in parallel [3, 17, 22, 2].

2.1 Divide-and-Conquer as a Means to Parallelism

Interest in divide-and-conquer as a programming paradigm lies in the potential for parallelism in computing partial results for divided data. Axford [3] describes it as follows:

Suppose there are P processors available for parallel computation, then $y = Divacon\ (d)$ can be computed by:

1. Compute a set of data values $d_1, d_2, d_3, ..., d_P$ by repeated application of *Divide()* to d until the data is subdivided into P parts.
2. For each of the data values d_i, compute a partial result by sequential computation on the i-th processor.
3. By repeated application of *Combine()* to pairs of partial results, compute the final result y from the set of partial results $y_1, y_2, y_3, ..., y_P$.

These three stages can each be implemented in parallel, although the greatest parallelism is possible for stage 2.

This property of divide-and-conquer makes the solution of general computational problems by a divide-and-conquer method desirable. Axford [3] gives some examples of these. In addition to simple computational problems, many computationally demanding ones have now been solved by divide-and-conquer methods, because it provides a way of implementing them in parallel [10].

The ZAPP project [14, 15] has shown that excellent parallel performance can be achieved by a divide-and-conquer method, although it should be noted that the kernel was written in OCCAM. Many other divide-and-conquer architectures have been implemented with varying degrees of success. [18, 12]

2.2 Discussion

Divide-and-conquer is a computational paradigm that is simply expressed, easily parallelised and easily implemented. However, it provides a programming building block rather than a complete solution. It is therefore essential that divide-and-conquer units be combined a sensible and efficient way. How this can be achieved using object-oriented programming is demonstrated later in this paper.

The next section describes the nature of object-oriented programming and goes on to describe its potential benefits in a parallel context.

3 Object-Oriented Programming : C++ and Divide-and-Conquer

Object oriented languages have been long considered good at encapsulating parallel concepts [9]; many systems exist which attempt to extend, and capitalize on, their ability to express parallelism [4, 13, 11].

C++ is a common choice for object-oriented parallel systems because of its retention of C's efficiency - of obvious importance in parallel systems. Most systems achieve parallelism by augmenting the language features of the language concerned. However C++, as described by Coplien [7], is rich in *meta*-features - simple building blocks which allow the construction of features that are not intrinsically part of the language. This is obviously preferable to additional language features, as it yields programs that are portable across a range of platforms which support the base language.

Some of the other salient features of object-oriented programming languages are:

Code Reuse The internals of any divide-and-conquer system are usually composed of a collection of well defined structures - trees, lists, stacks. The structured nature of object-oriented languages and their ability to encapsulate generic implementations means that these constructs can be rapidly assembled from existing software which is known to be correct. It also means that system components can be easily replaced in a modular fashion when better implementations are available.

Data Abstraction Data abstraction is particularly useful in this context. Data abstraction allows the internals of code, with some particular functionality, to be hidden from the developer using that functionality. In the case of parallel processing, data abstraction allows the 'hard' bits of parallelism, even the parallelism itself, to be hidden from the programmer. Additionally, data abstraction allows higher-level concepts to be grouped together without being hindered by the detail of implementation. Thus the problem of divide-and-conquer can be addressed at a system, divide-and-conquer or problem level, whichever is appropriate.

Operator Overloading Operator overloading allows the intuitive expression of the message passing paradigm; code like:

```
byte1 >> processA;
```

is possible, meaning "pass a byte of data from my process to process A". This means that the notational convenience of CSP can be simulated if so desired.

Operator overloading also allows divide-and-conquer methods to be hidden within a simple syntax that is easily used. For example, matrix multiplication by divide-and-conquer can be hidden in the "*" operator.

Inheritance and Polymorphism Object-oriented programming, coupled with inheritance, allows the specification and implementation of a basic parallel framework and interface, which is easily augmentable for increased functionality. This framework can be split into functional groups, and even abstract groups to which the functionality is added later. In terms of divide-and-conquer it means that a uniform interface can be developed between the implementation of parallelism and the divide-and-conquer paradigm. Augmentation of this basic interface using inheritance, allows the polymorphic development of divide-and-conquer functionality whilst capitalising on code-sharing.

3.1 Discussion

Many people have obtained good performance results from divide-and-conquer based systems. However, these systems have tended to be functional language based at the user level, and coded in OCCAM at the kernel level. While OCCAM gives the desired performance, it is difficult to see how these systems can be robust, maintainable and extensible; three desirable aspects of software systems [21]. A high level language and operating system would yield these three attributes, but would the performance suffer too much ? In addition to these basic software engineering principles, object-oriented programming has characteristics which are particularly suited to both parallelism in general, and to divide-and-conquer itself.

The next section outlines the development of the divide-and-conquer paradigm in a object-oriented programming context using C++. Data representation and some of the practical issues involved are described and discussed.

4 Object Oriented Design

As shown the divide-and-conquer paradigm is notationally simple, and an initial object-oriented representation is a matter of encapsulating the basic functionality within an object.

4.1 Basic divide-and-conquer

In order to practically implement the primary functions - *isSimple()*, *Function()*, *Divide()* and *Combine()* - it is convenient to split *Divide()* into two functions, `divideUpper()` and `divideLower()`. A call-by-reference scheme could have been adopted to unite the functionality, and for dividing on a non-binary basis this would have probably been necessary. However, the notational aspects are perhaps less intuitive.

4.2 Data Representation

Divide-and-conquer relies on manipulable data structures. For balanced divide-and-conquer where the computational demand is determined by the problem size, a vector structure is most useful. With this structure data sub-units can be directly extracted rather than having to traverse the sub-units, which would be necessary for a list based structure. Additionally, larger sub-units can be constrained to be contiguous in memory so that IO can be performed with a minimum of CPU cycles.

Alternatives are linked lists or binary trees. Linked lists would be especially useful for complicated `combine()` or `divide()` functions as the data sub-units can be rearranged very easily. However, the overhead in traversing the list to extract sub-units and to perform IO, is unnecessary in the problems that will be considered here.

McBurney and Sleep shared matrix structures between divide-and-conquer operations. This meant that data was not duplicated in the `divide()` stage, and so memory was conserved. This is especially important for matrix multiplication and other algorithms which are not data-parallel, as the splitting up of the data involves a certain amount of duplication. In cases such as these, it would seem sensible to use, as much as possible, a description of the data, rather than the data itself. This would also ease the task of passing virtual data descriptors around the processor network, when the problem is initially too big to fit on one processor.

4.3 Implementation

A simple, high-level (C++) interface for the divide-and-conquer paradigm, takes the form:

```
template <class T> class Divacon {
 public:
        virtual T               divideUpper (const T&)          =0;
        virtual T               divideLower (const T&)          =0;
        virtual T               combine (const T&, const T&)    =0;
        virtual T               function (const T&)             =0;
        virtual const boolean   isSimple (const T&) const       =0;
};
```

Although other possibilities, not involving parametrization, are possible the outline above achieves the most flexibility.

Given this initial framework, a number of extensions are necessary. This framework provides an interface to the problem through which the actual graph-reduction routines can operate. As such the interface can provide no information about the underlying problem specification, derived classes will build up this information.

Thus for a problem involving matrices the class derivation would be as follows:

```
struct MatrixData_t {
        unsigned int    rows;
        unsigned int    columns;
        unsigned int    row_position;
        unsigned int    column_position;
        double*         data;
};

class MatrixDivacon : public Divacon<MatrixData_t> {
 public:
        virtual MatrixData_t    divideUpper (const MatrixData_t&);
        virtual MatrixData_t    divideLower (const MatrixData_t&);
        virtual MatrixData_t    combine (const MatrixData_t&, const MatrixData_t&);
        virtual MatrixData_t    function (const MatrixData_t&);
        virtual const boolean   isSimple (const MatrixData_t&);
};
```

Whatever functionality is known at this stage can be added to the `MatrixDivacon` class, and the complete functionality added in derived classes e.g `MatrixMultiplicationDivacon`.

For any non-trivial problem the number of `MatrixData_t`'s created is quite large and therefore the size of `MatrixData_t` must be kept to a minimum. This means reducing data-members and eliminating virtual functions.

Finally, it was found that a number of other functions were necessary in order to conceal the actual data and methods beneath the Divacon interface. The interface then becomes:

```
template <class T> class Divacon {
 public:
    virtual const boolean      isSimple (const T&) const =0;
    virtual T                  divideUpper (const T& )   =0 ;
    virtual T                  divideLower (const T& )   =0 ;
    virtual T                  combine (const T&, const T&) =0 ;
    virtual T                  function (const T& )      =0 ;

    virtual void               outputData (const T&, const Process&)=0;
    virtual void               outputResults (const Process&)=0;
    virtual T                  inputData (const Process&)     =0;
    virtual T                  inputResults (const T&, const Process&)=0;
};
```

where `Process` is a compute node process identifier.

5 The Object Oriented Approach

As shown, object-oriented programming forms a viable framework not only for building a divide-and-conquer system, but also for expressing the divide-and-conquer paradigm elegantly and flexibly. However, object-oriented programming also provides the means for achieving an integrated approach to parallel processing.

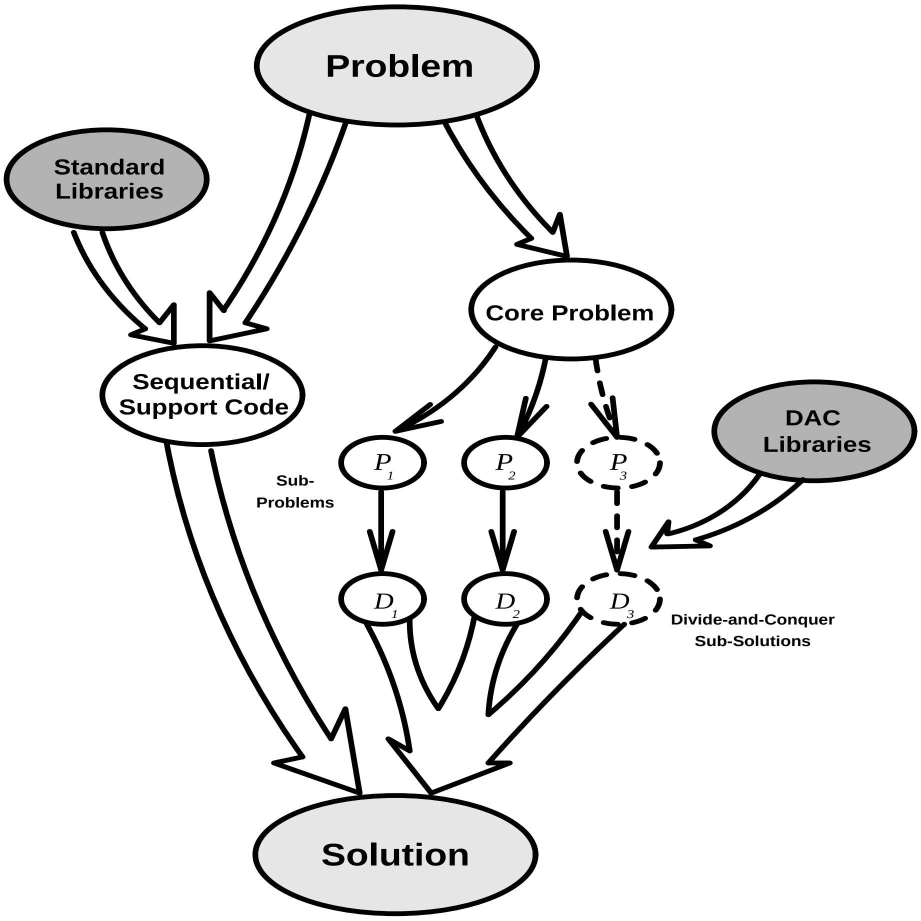

Figure 1: The Design Process

5.1 Motivation

In most engineering disciplines, real problems are characterized by their complexity and their non-uniform nature. Given a problem, there are usually many steps involved in solving that problem. Computationally, the requirements of a solution are usually diverse and multi-faceted. For instance, to recognise human speech usually requires some combination of acquisition, filtering, preprocessing, recognition, postprocessing and lexical-access. With today's problems any computational system must be able to provide sufficient flexibility to accommodate steps such as these, whilst maintaining efficiency as much as possible. It is possible to be overly concerned with the efficient solution of very specific problems, and give insufficient thought to the cost of integration of these solutions into a real system. Efficiency must be measured in software life-cycle terms as well as actual algorithmic efficiency.

Divide-and-conquer provides the flexibility necessary to accommodate a large problem solution set. Object-oriented programming provides the means for integrating efficiently, in all senses of the word, these various solutions to make up a complete solution.

5.2 Integration Using Object-Oriented Programming

It has already been described how the data abstraction and operator overloading aspects of object-oriented programming allow the 'hard' bits of parallel processing in this context to be abstracted from the programmer interface. This abstraction, together with the reusability and encapsulation of functionality, allows libraries of objects to be efficiently created and easily used. However, as described above, the interaction between these various objects and methods is as important as the objects and methods themselves. If this interaction is poor then methods that are individually extremely efficient can be next to useless, leading to poor overall system efficiency.

With the parallel evaluation of divide-and-conquer methods, it is crucial that the number of divide-and-conquer passes [1] is reduced to a minimum so that the communication cost exacted by such a system is minimised. Data dependencies may mean that computation cannot all be performed in one pass, but the number of times this is necessary must be kept low.

5.3 Sequential-Parallel and Parallel-Parallel Integration

Parallelised object methods can be made transparent to the user using appropriate functions and overloaded infix operators - for instance `MatrixMultiplicationDivacon` can be integrated into the operator "*". This allows the parallel and serial aspects of a problem to be integrated seamlessly without the parallel aspects interfering with the overall solution strategy. This also means that several parallel methods can be grouped together to yield a more complex method. The problem with small sub-units being combined to yield a larger solution is that this can lead to unnecessary inefficiencies. To take a trivial example - it is inappropriate to compute ,using divide-and-conquer many pairs of matrix operations separately - $constant * A + B$ computed as $constant * A$ and then $result + B$ would involve a tremendous amount of unnecessary communication the data would be broadcasted/retrieved twice when really all that is needed is for all data to be broadcast and each result element to be calculated as $constant * A_i + B_i$. As communication is the overriding overhead in MIMD parallel processing this needs to be avoided. This can be achieved by using delayed evaluation whilst still maintaining the notational convenience of operator overloading :

```
class vector {                          // a vector
        unsigned s;                     // size of the vector
        double *e;                      // contents of the vector
        vector(unsigned ss=0) : s(ss),  // constructor with size
                e(new double[ss]) {}
        ~vector() {
                delete[] e;
        }
        operator() (unsigned i) {       // element access operator
                return e[i];
        }
};

class cvector {                         // a vector and constant class
        double c;                       // the constant
        vector v;                       // the vector
        cvector(double cc, vector& vv) : c(cc), v(vv)
                {}                      // constructor from a vector and a constant
};

cvector operator* (double cc, vector& vv) {
                                        // multiplication operator (just returns a cvector)
        return cvector(cc, vv);
}

vector operator+ (cvector cv, vector& vv) {
                                        // plus operator does the actual computation
        vector rr(vv.s);
```

[1] A divide-and-conquer pass is divide-and-conquer evaluation involving some form of data broadcast, with all the data being present on the root processor at the beginning and end of the evaluation.

```
        for(int i = 0 ; i< vv.s; i++)
                rr(i) = cv.c * cv.v(i) + vv(i);
        return rr;
}

{                                       // example usage
        vector a,b,c
        c = 1.2 * a + b;
}
```

This trivial example serves to illustrate the method of delayed evaluation. In order to be useful an additional level of complexity is required.

5.4 Compound Divide-and-Conquer Objects

Delayed evaluation can be extended to perform a whole range of evaluation optimizations. In a sequential context, every object is characterized by its derivation from a single base object which specifies an interface to basic functionality. Compound objects have the basic form:

```
class Compound : public Base {
public:
        Data    evaluate() {    // virtual in the Base class
                                // foo is some arbitrary function
                return foo( bp1->evaluate(), bp2->evaluate() );
        }
protected:                      // constructor is protected to
                                // keep access internal
        Compound(Base* b1, Base* b2) : bp1(b1), bp2(b2) {}

private:
        Base*   bp1;
        Base*   bp2;

        friend class Base;
};
```

The function `foo()` will vary depending on the type of compound object. For instance if the object is an `AddedData` object then `foo()` will be the infix operator "+". The infix operator "+" will then be overloaded for `Base` objects to be:

```
Base& Base::operator+ (Base& b1) {
        return *new AddedData(this, b1);
}
```

with some appropriate garbage collection to release the `AddedData` object when it has been used. Objects that actually contain data are a special case and `evaluate()` simply returns the data itself.

This approach results in the construction of an evaluation tree. The tree can be pruned to eliminate temporaries and unnecessary operations - for example $A^T * B$ can be evaluated without calculating A^T. The tree is then evaluated by overloading the operator "=" so that it calls the root object's `evaluate()` function. This call recurses down the tree until `evaluate()` returns plain `Data`. Then as the recursion unwinds, evaluation takes place.

This is essentially how Davies' Newmat04 [8] package works. This method can be further extended to work with divide-and-conquer . In order to achieve this, every object is derived from `Divacon` as well as some `Base` object. The compound object is constructed so that all the primary and IO functions - `divideUpper(), divideLower()` etc - are recursively called like `evaluate()`. The operator "=" is then overloaded to perform the actual divide-and-conquer evaluation.

Of course, this integration will only work within the constraints of one divide-and-conquer pass and it is essential that the maximum possible computation is performed within each pass. Unlike the sequential example, data dependencies will mean that the evaluation tree can rarely be evaluated in one pass. It is therefore possible to envisage an evaluation scheme in which nodes of the evaluation tree are identified, at which no further single pass evaluation can be performed. When these nodes are identified divide-and-conquer processing takes place, rather than just in the operator "=".

5.5 Problem Partitioning

Rabhi [18] identified problem partitioning as crucial to achieving optimal speedups for balanced divide-and-conquer systems. The compound object scheme as described above, means that partitions can be specified for each divide-and-conquer operation [2]. For the recursed `isSimple()` function, the overall partition can be specified as a suitable combination - currently the *minimum* - of the individual operation partitions.

5.6 Conclusions

Thus integration of parallel methods in this context can be achieved in an efficient and flexible way for some problems. These ideas are denoted schematically by Figure 1.

The next section seeks to illustrate the ideas introduced so far by tracing through the implementation of the back-propagation algorithm in a divide-and-conquer fashion. Results are presented for such an implementation. The speedup gained is obviously not optimal but it is a speedup gained at a relatively low implementation cost.

6 The Back-Propagation Algorithm

Given a three-layer neural network with I input units, J hidden units and K output units, then the output $\underline{\mathbf{o}}$ is related to the input $\underline{\sigma}$ by:

$$\underline{\mathbf{o}}^K = f_K((\mathbf{W}^{JK})^T f_J((\mathbf{W}^{IJ})^T \underline{\mathbf{o}}^I)) \tag{1}$$

This is the feed forward equation. The back-propagation of the error is given by:

$$\Delta\mathbf{W}^{JK} = \eta\underline{\mathbf{o}}^J((\underline{\mathbf{t}}^K - \underline{\mathbf{o}}^K) \otimes f'_K(\underline{\sigma}^K))^T \tag{2}$$

$$\Delta\mathbf{W}^{IJ} = \eta\underline{\mathbf{o}}^I(\mathbf{W}^{JK}[(\underline{\mathbf{t}}^K - \underline{\mathbf{o}}^K) \otimes f'_K(\underline{\sigma}^K)] \otimes f'_J(\underline{\sigma}^J))^T \tag{3}$$

6.1 Approaches

It is necessary now to consider how this problem can be broken down modularly in a divide-and-conquer fashion. There are two options:

- If the network is very large then the algorithm itself can be broken down using divide-and-conquer and basic matrix operations. This is not necessarily highly efficient. This will be denoted as a vertical implementation.

---

[2] For instance matrix multiplication

- If the network is small then the whole algorithm can be kept on each node and different weight updates can be calculated on each node and the results combined. This is a common way to implement back-propagation - to run the entire training set through the network and to sum the weight updates. This has the advantage of eliminating random noise. This will be denoted as a horizontal implementation.

Let us consider the second possibility first as this is likely to be the easiest one to implement.

6.2 Horizontal Back-Propagation

Given a training set of n frames $\mathbf{T_S}(n) = \{f_1, \ldots, f_n\}$, this can be split recursively into groups of frames. Thus

$$\begin{aligned} Divide() : \ \mathbf{T_S}(n_s) \ \ \rightarrow \ \ & \{\mathbf{T_{S_1}}(n_s/2), \mathbf{T_{S_2}}(n_s/2)\} \ WHERE \\ & \mathbf{T_{S_1}}(n_s/2) = \{f_1, \ldots, f_{n_s/2}\} \\ & \mathbf{T_{S_2}}(n_s/2) = \{f_{n_s/2+1}, \ldots, f_{n_s}\} \end{aligned} \tag{4}$$

If n_s defines the number of frames in a given set then a straightforward $isSimple()$ function is when n_s is equal to some suitable $partition$.

$$isSimple() : \ \mathbf{T_S}(n_s) \rightarrow n_s = partition \tag{5}$$

The $Function()$ function then becomes a cycle of the feed-forward / error back-propagation equations.

The combination function is equally straightforward and involves an addition of the returned weight updates. Thus the result set $\mathbf{R_S}$ for each node is defined by:

$$\begin{aligned}
Combine() : \{\mathbf{R}_{\mathbf{S}_1}, \mathbf{R}_{\mathbf{S}_2}\} &\rightarrow \mathbf{R_S}\ WHERE \\
&\quad \mathbf{R_S} = \{\Delta\mathbf{W}^{IJ}, \Delta\mathbf{W}^{JK}\}\ WHERE \\
&\quad \Delta\mathbf{W}^{IJ} = \Delta\mathbf{W}_1^{IJ} + \Delta\mathbf{W}_2^{IJ} \\
&\quad \Delta\mathbf{W}^{JK} = \Delta\mathbf{W}_1^{JK} + \Delta\mathbf{W}_2^{JK}
\end{aligned} \tag{6}$$

In addition to these primary functions it is also necessary to define the sets for input/output functions. These will be called $\mathbf{B_S}$ the Broadcast Set and $\mathbf{Re_S}$ the Reel Set. These are:

$$\mathbf{B_S} = \{\mathbf{W}^{JK}, \mathbf{W}^{IJ}, \underline{\mathbf{o}}^I, \underline{\mathbf{t}}^K\} \tag{7}$$

$$\mathbf{Re_S} = \{\Delta\mathbf{W}^{IJ}, \Delta\mathbf{W}^{JK}\} \tag{8}$$

Note that this approach runs into the problem of lack of memory. With a simplistic implementation the entire training set would need to be initially held on the root node together with all the other associated data. A more sophisticated approach might use some through routing technique so that data was not actually transmitted until needed. Additionally the idea of using the host as the root and initially employing divide-and-conquer as a scheduling strategy would be helpful here. This would involve dividing the data until the sub-divided parts are small enough to run on the first node, and the evaluation function becomes that of scheduling this load onto the first node for computation.

6.3 Vertical Back-Propagation

The second approach is to break down the algorithm itself into divide-and-conquer format, rather than the data. Only the feed-forward equation will be considered to illustrate the method. It is necessary at this stage to define some new terminology to specify the state of data structures. Principally whether the leaf result of a divide-and-conquer operation is in a column-wise segmented or row-wise segmented state. These will be denoted by the subscripts c and r respectively. Unsubscripted data structures are unsegmented.

By definition $f(\underline{\sigma})$ is a data parallel operation, and thus can be computed by divide-and-conquer by simply splitting the vector along its length. This gives:

$$\begin{aligned}
Divide() : \underline{\mathbf{v}} &\rightarrow \{\underline{\mathbf{v}}_{1_r}, \underline{\mathbf{v}}_{2_r}\}\ WHERE \\
&\quad \underline{\mathbf{v}}_1 = (v_1, \ldots, v_{n/2})^T \\
&\quad \underline{\mathbf{v}}_1 = (v_{n/2+1}, \ldots, v_n)^T \\
isSimple() : \underline{\mathbf{v}} &\rightarrow elems(\underline{\mathbf{v}}) \leq const \\
Function() : \underline{\mathbf{v}}_h &\rightarrow \underline{\mathbf{v}}'_r\ WHERE \\
&\quad v'_i = f(v_i) \\
Combine() : \{\underline{\mathbf{v}}_{1_h}, \underline{\mathbf{v}}_{2_h}\} &\rightarrow \underline{\mathbf{v}}\ WHERE \\
&\quad \underline{\mathbf{v}} = (\underline{\mathbf{v}}_{1_r}^T, \underline{\mathbf{v}}_{2_r}^T)^T
\end{aligned} \tag{9}$$

Note that functions of this nature can be recursively applied so that only one divide-and-conquer pass is necessary.

Matrix-Vector multiplications can be performed by segmenting the matrix row-wise and keeping the vector intact. Thus:

$$Divide() : \{\mathbf{M}, \underline{\mathbf{v}}\} \rightarrow \{\mathbf{M}_{1_r}, \mathbf{M}_{2_r}, \underline{\mathbf{v}}\}\ WHERE$$

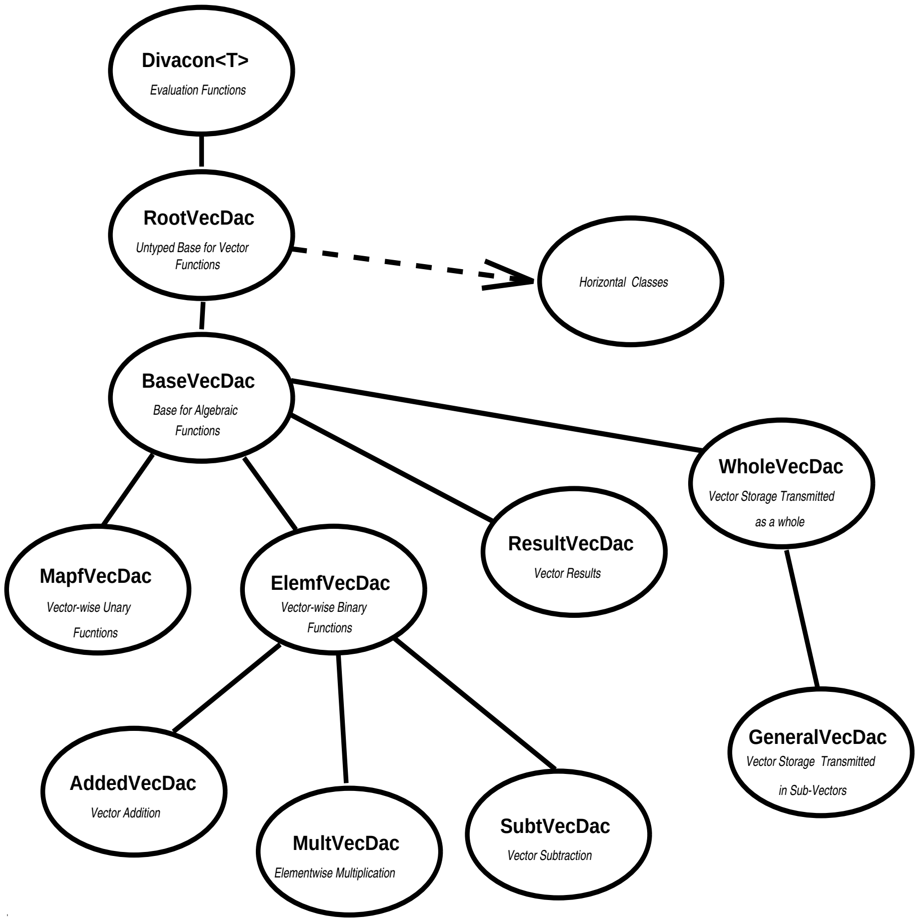

Figure 2: Object Hierarchy

$$
\begin{aligned}
& & \mathbf{M} = \begin{pmatrix} \mathbf{M}_{1_r} \\ \mathbf{M}_{2_r} \end{pmatrix} \\
isSimple() : \ \mathbf{M} & \rightarrow & elems(\mathbf{M}) \leq partition \\
Function() : \ \{\mathbf{M}_r, \underline{\mathbf{v}}\} & \rightarrow & \underline{\mathbf{u}}_r \ \ WHERE \\
& & \underline{\mathbf{u}}_r = \mathbf{M}_r \underline{\mathbf{v}} \\
Combine() : \ \{\mathbf{M}_{1_r}, \mathbf{M}_{2_r}, \underline{\mathbf{v}}\} & \rightarrow & \{\mathbf{M}, \underline{\mathbf{v}}\} \ \ WHERE \\
& & \mathbf{M} = \begin{pmatrix} \mathbf{M}_{1_r} \\ \mathbf{M}_{2_r} \end{pmatrix}
\end{aligned} \tag{10}
$$

The vector $\underline{\mathbf{u}}$ generated by this function is in the row-wise segmented state when $Function()$ is applied. Thus, data-parallel vector operations can be applied to this result in the same divide-and-conquer pass. For example:

$$f(\mathbf{M}_r \underline{\mathbf{v}})$$

Since the resultant vector is row-wise segmented it is cannot be re-used by equations 10. Thus the calculation of $\underline{\mathbf{o}}^K$ requires two divide-and-conquer passes.

6.4 Implementation Issues

An attempt was made to implement the two schemes described above, on a network of transputers using the Trollius^{TM} [5] operating system and AT&T C++ v3.0.1. During the course of implementation a divide-and-conquer object hierarchy evolved as illustrated by figure 2. This hierarchy was influenced by the delayed evaluation techniques described in section 7, and embodied the algebraic manipulations necessary to implement the vertical scheme. Programs could then be written in simple algebraic style:

```
main()
{
        const SZ = 128;
        VectorDac<float> A(SZ), B(SZ), C(SZ), R(SZ);
        MatrixDac<float> M(SZ,SZ);

        R = A + B * exp(M*C);
}
```

As discussed previously, the operator "=" actually performs the divide-and-conquer operations, with the other operators simply building up the compound divide-and-conquer object.

The feed forward equations 21 & 22 were implemented for the vertical scheme. The execution time for evaluating the first layer of one frame on one processor was found to be so low (< 1s) that parallel implementation yielded very little improvement and was in some cases slower! This was due to the parallelising overhead for this case being comparable to the actual computation being performed. No further implementation was made as the overheads involved would only multiply with further divide-and-conquer passes. However, the insight gained into combining divide-and-conquer objects was invaluable in implementing the horizontal approach.

6.4.1 Parameterized Typing

The parameterized typing facility of C++ proved to be invaluable in crossing from the vertical implementation to the horizontal implementation. This facility, which allowed `VectorDac`'s of `float`s or `double`s to be created, was flexible enough for `VectorDac`s of `Vector`s of `float`s to be created easily using the existing code. This meant that implementing `Vector`s of training frames, for example, was a simple matter of providing more generalised IO routines.

6.4.2 Object Combination

The horizontal implementation is currently coded as:

```
void main()
{
    // ************* setup data structures *************

    const TRAINING_FRAMES = 512;
    const I = 32;              // nn parameters
    const J = 136;
    const K = 10;

    VectorDac< Vector<float> >  inputs(TRAINING_FRAMES),
                                outputs(TRAINING_FRAMES);

    inputs      = Vector<float>(I); // resize the data
    outputs     = Vector<float>(K);

    Matrix<float>       d_Wij(I,J), Wij(I,J),
                        d_Wjk(J,K), Wjk(J,K);
    CombVectorDac< Matrix<float> >      delta_Wij(d_Wij, TRAINING_FRAMES),
                                        delta_Wjk(d_Wjk, TRAINING_FRAMES);
```

```
    // ********** initialisation for root node *********
    if (getnodeid() == 1){
        inputs = outputs = Wij = Wjk = Random; // randomize everything
                                               // in lieu of actual data
    }

    // ********* Divide-and-Conquer operations *********

    BackpropDac<float> bp(inputs, outputs, delta_Wij, delta_Wjk, Wij,Wjk);
    bp.evaluateExpression();

    // ********* post-processing for root node *********
    if (getnodeid() == 1) {
        Wij += d_Wij;          // update weights
        Wjk += d_Wjk;
    }
}
```

The divide-and-conquer object `BackpropDac<T>` simply applies the training algorithm to its component divide-and-conquer objects. The training algorithm is simply expressed using standard C++ Vector / Matrix objects and infix operators. It is apparent that in this case the actual combination of the divide-and-conquer objects is rather cumbersome - a six parameter function call - as opposed to the elegance of the vertical implementation. Thus a more intuitive combination scheme would be of much value.

6.4.3 Memory Exhaustion

The largest problem size that could be tested was $I = 32, J = 136, K = 10, Frames = 512$. Any larger than this and the memory was exhausted. This limitation was partially due to the large code size generated by the C++ compiler, but mainly due to the fact the *all* the problem data starts out on the root node, and as the job diffuses the data thins out. When actual evaluation is performed the data required for this evaluation is relatively small in comparison with the total data. Thus, some sort of virtual data transmission scheme would be necessary in order to implement larger problems, and make efficient use of the available resources. This would mean that data was only present on a node when it was actually needed and that a virtual representation was transmitted at other times.

7 Results

Figure 3 shows the speedup gained for the horizontal scheme with $I = 32, J = 136, K = 10, Frames = 512$.

The experiments were conducted on a 4x4 mesh of T800 transputers. The mesh was wrapped into a torus. The processor numbers used was increased rasterwise - left to right, top to bottom - the root node being at the beginning of the raster.

A number of features are worth noting.

- Because of the way the processor numbers were increased, the root transputer uses one of its links for processors 1-4, two for processors 5-12 and three for processors 13-16. The fourth link is connected to the host. In the worst case the expected speedup will be $2^{\#links}$, as there will be $\frac{1}{2^{\#links}}$ of the problem left on the root processor. This observation is borne out by the results, speedup break points coming at 2,5 and 13 processors. This constitutes a strong argument for using a hypercube connected topology.

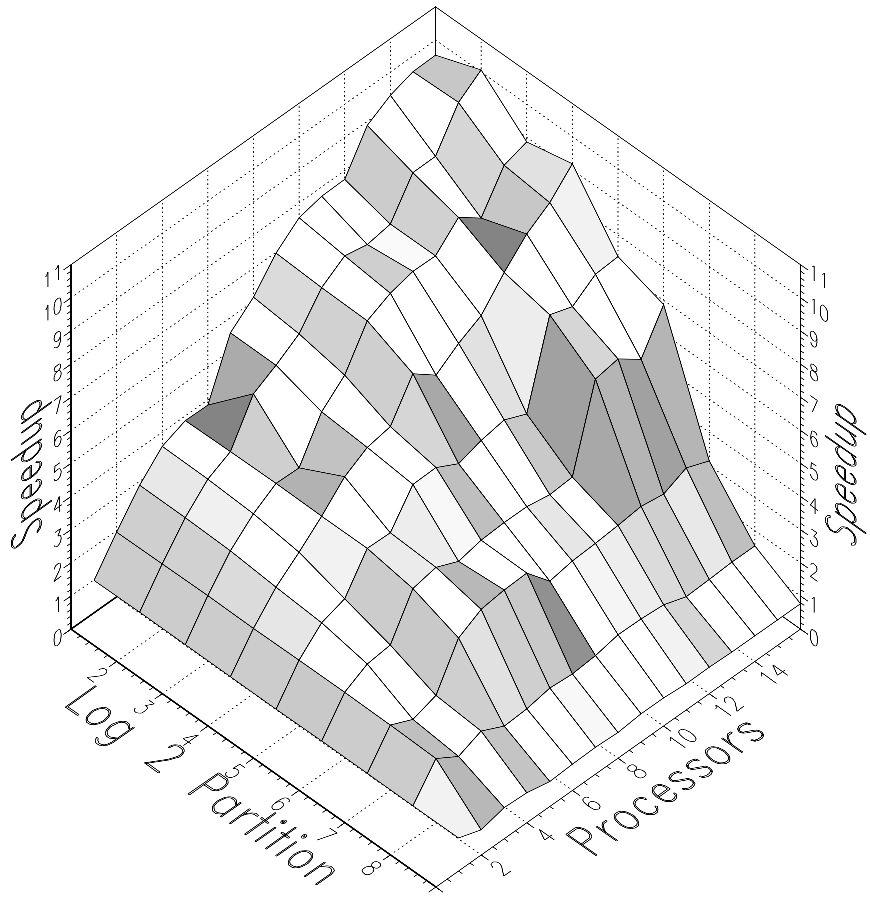

Figure 3: Results for Horizontal Back-Propagation

- For large partitions, the number of available problems decreases until it is less than the number of available processors. At this point performance tails of rapidly. In the limit, the speedup tends to that of the single processor case.
- When more than one processor is available, but the partition size constrains the system to use only one, the speedup gained is slightly less than the single node case. This is probably because of the overhead introduced by the bidding scheduling scheme - scheduling requests are still made to the root node, it just cannot satisfy any of them.
- Rhabi's [18] partitioning results - optimal speedups for partitions matching the sub-task size to the number of processors - are not observable here. This may be because of a difference in scheduling strategy between the two systems. The strategy employed here means that with more sub-tasks, more processors will be able to be used towards the end of a processing pass. With less sub-tasks, the root processor will be the only processor left computing, towards the end of the computation, when the sub-task queue has been exhausted.

8 Conclusions

It has been shown that divide-and-conquer provides a general means to parallelism constrained by the expressive power of divide-and-conquer paradigm. The limitations imposed by this constraint restricting in many ways. However, divide-and-conquer allows a shift of emphasis from that of solving parallel problems to that of re-expressing algorithms within a set of clearly defined boundaries.

Object-oriented programming does not provide any 'magic' means to parallelism through divide-and-conquer. However, it does allow the divide-and-conquer paradigm to be represented conveniently and flexibly in a declarative environment. It also allows divide-and-conquer solutions to be combined in a uniform fashion. Again, this combination does not magically happen; the combination of divide-and-conquer solutions requires a good understanding of the overall problem and has to be specifically defined. However, object-oriented programming provides the framework for its implementation and allows this implementation to be hidden from subsequent users.

From an implementation standpoint, it is known that divide-and-conquer can provide a usable degree of efficiency. However, this paper has shown that even an extremely high-level approach, using object-oriented programming does not have to be less efficient. The advantage of a high-level approach is that it paves the road for more complex algorithms and implementations.

This approach in no-way presents a magic formula that automatically parallelises every problem encountered; the C++ code is compiled to standard C-code and as such the actual functionality could be written using C. However, the approach does sufficiently alter and abstract one's viewpoint of parallel processing problems so that their solution becomes achievable in a structured, sensible and efficient - in every sense - way.

9 Further Work

Providing a suitable framework for divide-and-conquer in C++ has proved relatively straightforward. Providing this framework whilst maintaining efficiency has proved less than straightforward and there is much scope for improving the implementation to achieve better efficiency, especially in the scheduling scheme.

The choice of representing the data as vectors has been suitable in this context. However, for different classes of problem some other data structure might be preferable - for instance a linked list. The generic nature of C++ will make an implementation based on this straightforward.

The compound divide-and-conquer representation could be more subtly evaluated. Unnecessary operations could be pruned from the tree. Objects which were candidates for single divide-and-conquer passes, could be identified and evaluated automatically.

A virtual data transmission scheme would ease memory problems. If data was only actually transmitted when it was required for computation, then all the data would not need to be held initially on the root processor.

References

[1] A. V. Aho, J. E. Hopcraft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983. p2,3 C++ is rich in meta-features which allow functionality to be built in to the language.

[2] M. J. Atallah, R. Cole, and M. T. Goodrich. Cascading Divide-and-Conquer: A technique for Designing Parallel Algorithms. *SIAM Journal of Computing*, 18(3):499–532, June 1989.

[3] T. H. Axford. An Elementary Langauge Construct for Parallel Programming. *ACM Sigplan Notices*, 25(7):72–80, 1990.

[4] M. Ben-Ari. *Principles of Concurrent and Distributed Programming.* Computer Science. Prentice-Hall International, 1990.

[5] Greg Burns et al. All About Trollius. *Occam User's Group Newsletter*, 1990.

[6] A. R. Clare. A Generic Divide-and-Conquer Kernel for the Meiko Computing Surface. 1990.

[7] J. O. Coplien. *Advanced C++ Programming Styles and Idioms.* Addison-Wesley, 1992.

[8] R. B. Davies. Newmat04, An Experimental Matrix Package In C++. 1991.

[9] S. A. Dobson and A. J. Wellings. Programming Highly Parallel General-Purpose Applications. In *Workshop on Abstract Machine Models for Highly Parallel Computers*, volume 2, pages 49–. Univeristy of Leeds, 1991.

[10] D. Gill and E. Tadmor. An $O(N^2)$ Method for Computing the Eigensystem of $N \times N$ Symmetric Tridiagonal Matrices by the Divide and Conquer Approach. *SIAM Journal of Statistical Computing*, 11(1):161–173, January 1990.

[11] A. Grimshaw. The Mentat Run-Time System: Support for Medium Grain Parallel Computation. In D. W. Walker and Q. F. Stout, editors, *The Fifth Distributed Memory Computing Conference*, pages 1064–1073. Computer Society Press, 1990.

[12] D. H. Grit and J. R. McGraw. Programming Divide and Conquer for a MIMD Machine. *Software-Practice and Experience*, 15(1):41–53, January 1985.

[13] A. Khanna. On Managing Classes in a Distributed Object-Oriented Operating System. In D. W. Walker and Q. F. Stout, editors, *The Fifth Distributed Memory Computing Conference*, pages 1056–1063. Computer Society Press, 1990.

[14] D. L. Mcburney and M. R. Sleep. Transputer-Based Experiments with the ZAPP Architecture. In de Bakker et al., editors, *PARLE*. Springer-Verlag LNCS, Vol. 258, 1987.

[15] D. L. Mcburney and M. R. Sleep. Transputers + Virtual Tree Kernel = Real Speedups. In G. C. Fox, editor, *The Fourth DMCC*. Association for Computing Machinery, 1988.

[16] Z. G. Mou. Divacon: A Parallel Language for Scientific Computing Based on Divide-and-Conquer. In *1990 Third Symposium On The Frontiers Of Massively Parallel Computation*, pages 451–461, 1990.

[17] Z. G. Mou and P. Hudak. An Algebraic Model for Divide-and-Conquer and Its Parallelism. *The Journal of Supercomputing*, (2):257–278, 1988.

[18] F. A. Rabhi and G. A. Manson. Divide-and-Conquer and Parallel Graph Reduction. *Parallel Computing*, (17):189–205, 1991.

[19] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning Internal Representations by Error Propogation. In Anderson and Rosenfield, editors, *Neurocomputing*. MIT Press, 1988.

[20] D. R. Smith. Top-Down Synthesis of Divide-and-Conquer Algorithms. *Artificial Intelligence*, 27:43–96, 1985.

[21] I. Sommerville. *Software Engineering*. International Computer Science Series. Addison-Wesley, 1985.

[22] V. Strassen. Gaussian Elimination is not Optimal. *Numerische Mathemtik*, 13:354–356, 1969.

[23] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.