GENERALIZED PARALLEL PROGRAMMING WITH DIVIDE-AND-CONQUER: THE BEEBLEBROX SYSTEM

A. J. Piper & R. W. Prager

CUED/F-INFENG/TR 132

July 1993

Cambridge University Engineering Department
Trumpington Street
Cambridge CB2 1PZ
England

Email: ajp@eng.cam.ac.uk

Generalized Parallel Programming with Divide-and-Conquer: The Beeblebrox System

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

July 12, 1993

Abstract

Object-oriented programming has been widely proposed as a discipline suited to extracting parallelism from a program. However, object-oriented programming provides no more guarantee of efficient parallelism than procedural programming. The divide-and-conquer (D&C) algorithm does guarantee a measure of efficient parallelism as demonstrated by many researchers. We present a system that uses object-oriented techniques to encapsulate the D&C algorithm and which is much more flexible than our previous work. We go on to describe a stack-based evaluation algorithm that allows the nesting of D&C evaluations. We use this framework to implement a tree-based language model which gives good parallel performance. We then present some techniques for achieving better performance and which are universally applicable to D&C problems. Finally we analyse the performance in theoretical terms and show that a suitable implementation can achieve better speedup than $N/log_2 N$.

1 Introduction

Practical parallel computing has been a reality for an appreciable number of years. Research in this area has matured and significant advances have been made, especially with regard to hardware organisation and architecture. For MIMD computers, however, managing and programming for concurrent performance has proved, with the obvious exception of data-parallel programming, to be a difficult and largely intractable problem [13].

Object-oriented concurrent programming, in various forms and levels of sophistication, has been proposed as an answer to this intractability. The debate continues - passive objects, active objects, threaded message invocations, remote procedure calls [6] - though there is some agreement that concurrency mechanisms make object-oriented languages better at modelling real-world situations.

However, it has been pointed out that the heart of object-oriented programming is orthogonal to concurrency. Although the general object-oriented programming model bears similarities to the concurrent programming model, programming with an object-oriented discipline does not necessarily generate the speedup that marks the holy grail of concurrent programming. If concurrency issues are addressed, most notably locality of reference [11], then object-oriented programming can yield the performance that one requires, but to constrain the programmer in this fashion is to contradict the supposed benefits of object-oriented programming coupled with concurrent programming.

An object-oriented program can be rather like a tangled mess of threads from a concurrency point-of-view. Although each thread exhibits significant concurrency with other threads, the whole is not separable. If the whole is not separable then large-grain evaluation cannot be performed in an efficient way and even fine-grained evaluation has proven tricky [5].

Many concurrent programming systems realise the need for the programmer to play a significant role in identifying concurrency and allowing their design decisions to be influenced by concurrency issues [10]. Yet, all too often, object-oriented concurrent programming is mooted as a universal panacea for the problems of concurrent programming, placing no constraints on the programmer apart from the discipline of object-oriented programming. Experience shows us that taking advantage of the programmer's skill in grasping the structure of a problem invariably reveals levels of concurrency that would have been missed by an automatic system. However, allowing the programmer to make explicit parallelism decisions is not the only way of capitalising on this skill. Instead, constraints can be placed on the programmer which, if satisfied, are guaranteed to yield usable concurrency. Parallel functional languages operate in this way. In a similar manner object-oriented programming can be constrained through the use of the D&C algorithm [16].

1.1 Paper organisation

In section 2 we describe the D&C algorithm and its parallel implementation. Section 3 describes the object-oriented D&C approach and its implementation together with some additional features. We also examine the approach in the light of the actor programming model. Section 4 describes a method of reducing a programmers burden under object-oriented D&C. Section 5 describes tree-based language models and their parallel implementation under object-oriented D&C. Finally we give some performance results for our system in section 6. We also give some performance improvements that should be applicable to any MIMD D&C system.

2 Divide-and-Conquer

In this section we briefly describe the D&C algorithm and its parallel implementation.

D&C is well known for its parallel evaluation properties and has been widely used in a functional language evaluation context [18] as well as a programming paradigm in its own right [2, 15].

The D&C algorithm can be represented in pseudocode as follows:

```
FUNCTION divacon (data)
BEGIN
        IF simple (data)
        THEN RETURN evaluate (data)
        ELSE combine (divacon (divide (data)))
END
```

where *divide()* divides a task into sub-tasks, *simple()* specifies if the task is small enough to compute, *combine()* combines partial results and *evaluate()* yields a partial result from a sub-task.

2.1 D&C as a Means to Parallelism

Interest in D&C as a programming paradigm lies in the potential for parallelism in computing partial results for divided data. Axford [2] 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.

However, as we will describe, the parallelism of stages 1 and 3 can be critical for good performance of parallel D&C systems.

3 Kernel Structure

In this section we describe desirable properties for an object-oriented D&C system. We also describe, in general terms, the implementation of these features in a C++ environment. Finally, we compare object-oriented D&C with the actor model of concurrent computation and describe how various features fit into this model.

3.1 General approach

The basis for object-oriented D&C is to incorporate the primary D&C functions into an object. This design consideration together with normal object-oriented features yields quite a powerful programming paradigm [16].

Previously, we described [16] an object-oriented D&C structure that incorporated various features in a "bolted-on" fashion. In using this structure it became apparent that a more uniform and cleaner approach was needed, so that these features would then be an integral part of the design and so that more advanced features could be incorporated. The goals of this new design were:

Full polymorphism. Our previous implementation consisted of two basic class structures. One represented the D&C functionality required, the other the data on which that functionality would work. The reason for this separation of data and functionality, which appears contrary to the goal of object-oriented programming, was two-fold. Firstly, the data-holding structures needed to be kept simple and small as the number of data objects produced during D&C evaluation was considerable. Secondly, there was no immediately apparent way of representing actions and data which were common to all D&C objects, if all actions and data were defined in a single structure. Hence, this global data was folded into the functional half of the design as this was in itself global to all D&C objects.

The problem with this approach was that the D&C functionality could not be made fully polymorphic as it had to be typed by the data structure it was going to use - in other words this information had to be built into it. The D&C functionality required a polymorphic interface because this interface was to be used by the D&C evaluation code, and we did not want to duplicate this code for every different type of D&C problem! In the end this evaluation code was parameterized upon the D&C data structure using a class template facility, thus putting the onus on the compiler rather than the programmer.

However, this is not a particularly neat solution and runs into all sorts of problems if we want to evaluate more than one type of D&C problem at the same time. The obvious solution is to bundle the two halves into one type of object. A fully polymorphic interface can then be written and all the original problems go away.

"Virtual" object-creation. Once D&C objects are fully polymorphic they can be evaluated without any knowledge of their true type. However, for this anonymity to be transmitted coherently in a message passing environment we must have some way of reconstructing these objects into their actual type.

Multiple evaluations. Provision of the features described above paves the way for concurrently evaluating multiple types of D&C objects. This would then allow programs to be written more flexibly, incorporating a greater degree of parallelism.

Mix-in-based program development. Ideally, most useful D&C functionality should be provided as a library if at all possible. This means that a programmer would be able to build his program from existing building blocks. However, given the four D&C primitives it would be a rather large library that had to provide all possible combinations of different primitives. What is needed is the ability to "mix in" single primitives, thus only requiring these single primitives in the library.

Delayed evaluation. We previously described [17] the advantages of delayed evaluation for creating efficient high-level algebraic operations. However, this feature was rather difficult to use in its previous incarnation.

3.2 Stack-based evaluation

Previous MIMD D&C implementations have generally adopted a tree-like evaluation structure. This has the advantage of maintaining the node orderings as well as being an easy structure to traverse. Another approach is to split the entire task completely and then gradually evaluate the pending sub-tasks. However, this is to perform a breadth-first evaluation which results in high memory overheads. Additionally, if division involves some computational overhead - and in many problems division represents the *only* computational overhead - then efficient parallelism is not possible. This approach can also suffer from incorrect task ordering.

D&C lends itself naturally to recursive evaluation. However, parallel evaluation requires that we have access to tasks that have not yet been evaluated and recursive evaluation does not allow us to do this easily.

Sedgewick [20, p45], gives an algorithm for traversing a binary-tree using a stack rather than by a recursive method. By modifying this algorithm it proves possible to evaluate a D&C task using a stack. We can do this by virtue of the fact that a D&C object holds all information necessary for further division at that point.

Two issues must be considered. Sedgewick's algorithm gives a way to traverse a tree in place, but of course the D&C algorithm necessitates *building* a tree and *ascending* it as well. The former is important to consider since as the tree is not in-place, the ordering of the nodes cannot be fixed rigidly. The latter is important as the original algorithm discards the tree information after it has visited nodes - so the information we require for division is present but that for combination is lost.

The combination problem can be solved by introducing a second stack onto which nodes are placed after they have passed through the divide stage of evaluation. If we do this correctly then we can pop successive pairs of nodes from this stack for combination (figure 1). Unfortunately, this works fine except for a few pathological cases which cause incorrect node ordering or evaluation. A trivial example is shown in 1.

If we have an unbalanced tree, where a node has less than two children, then the problem is more serious. If we modify the algorithm to cope with the left node only case then the right node only case fails and vice versa.

The way we solve this problem is by introducing a key to the stack which we use to encode the *level* at which a particular object is in the tree. i.e. the root object is at level 0, its two children are at level 1 and so on. By doing this we can make sure that only pairs with the same level are combined. The modified algorithm is given in program 2.

3.2.1 Advantages of a stack

Stack-based evaluation of D&C problems provides more than just a simple, elegant algorithm. Most MIMD D&C implementations utilise a stack for holding pending tasks [14, 16] because the tasks can be taken from the *bottom* of the stack for parallel evaluation. Since our whole evaluation scheme is stack based, tasks eligible for parallel evaluation are readily available. However, more importantly, we can nest our evaluations.

3.2.2 Nested evaluation

We have argued [16] that the design of efficient parallel algorithms should only be a means to an end, rather than an end in itself. However, many parallel implementations concentrate solely on

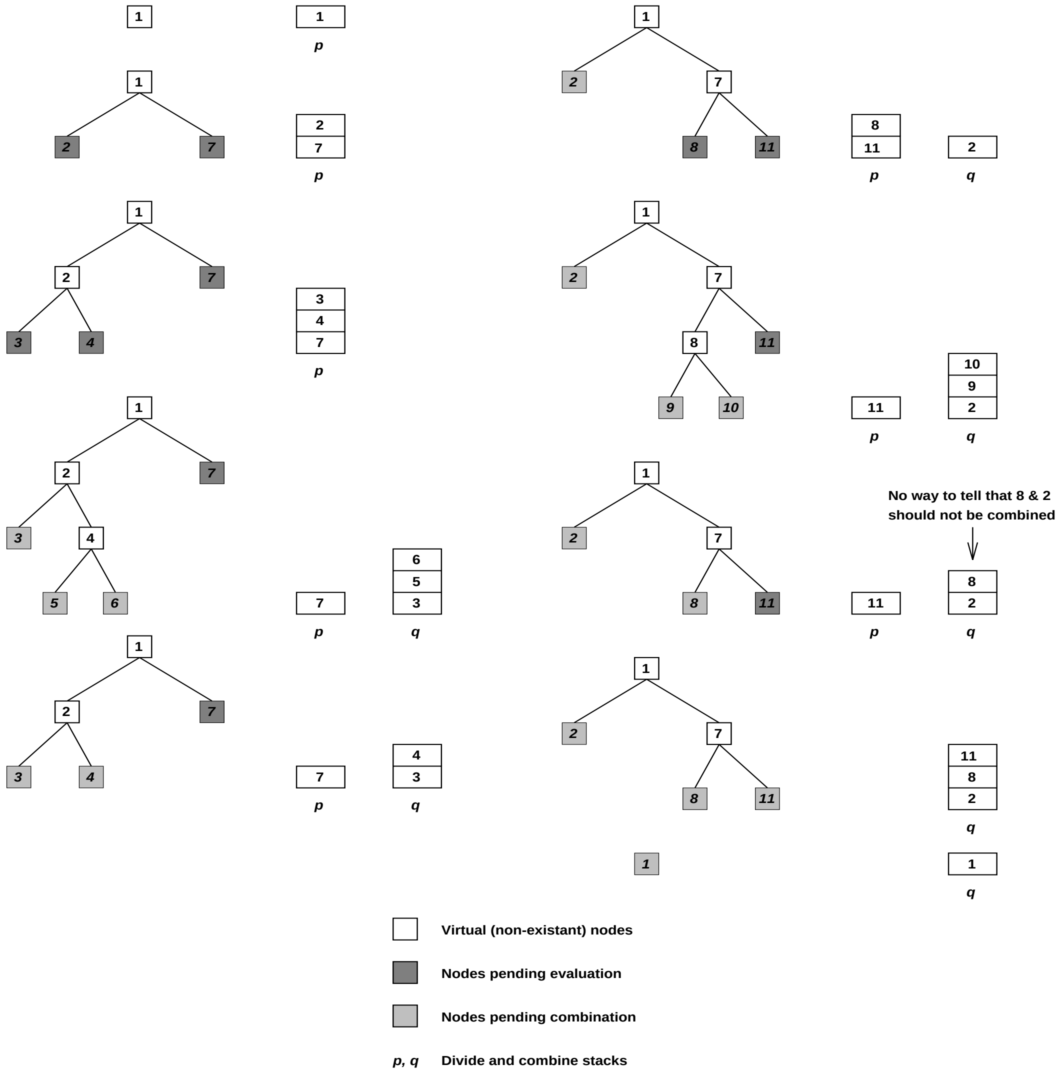
No way to tell that 8 & 2 should not be combined
Virtual (non-existant) nodes
Nodes pending evaluation
Nodes pending combination
p, q Divide and combine stacks

Figure 1: Stack-based evaluation

the efficiency of a single algorithm without much thought as to the operation of this part in a wider whole. This is especially true of D&C implementations. We have argued that D&C gives us the means to integrate together efficient parallel algorithms; nested evaluation makes this more of a reality. By adopting a stack-based evaluation structure, it becomes possible to evaluate multiple D&C task at the same time, i.e. a D&C task can contain other, differing D&C tasks which will be evaluated in parallel as well as the main task.

The beauty of this system is that, although nested D&C tasks imply a finer grain of concurrency, they do not imply a performance loss. This is because the higher level, and therefore larger, tasks will be stacked first. The finer grain tasks will only be evaluated in parallel if there are no larger tasks available on the stack. In theory this should only happen at the very beginning and end of processing.

In case parallel evaluation is *not* required - in debugging D&C applications for example - it is easy to provide a recursive implementation which is interchangeable with the stack-based algorithm. If necessary, it would be trivial to pass a flag to a D&C object, when it is initialised, determining which algorithm to use.

```
Dac Dac::run() const {
     Dac l,r,n;
     int s; // current level
/*
 * This is our exception - when the root node is simple then we simply evaluate
 * it.
 */
     if (simple()) return evaluate();
/*
 * we now save the stack positions in case we recursively call this function
 */
     int ppos = *p;
     int qpos = *q;

     p→push(*this, DacStack::limbo, 1); // musn't spawn this one !

     while (!p→empty(ppos)) { // emptiness is relative to ppos
/*
 * we'll start to divide until we have something to combine
 */
          while (!q→pair(qpos)) {

               s = p→key(); // get the level
               n = p→pop(); // get the object
/*
 * note that n might have suddenly become simple under our noses ...
 */
               if (!n.simple() && !(n = n.divide(l,r))) {
/*
 * if our two child nodes are simple then we evaluate them and push them
 * onto the combine stack. Otherwise we push both nodes onto the evaluation
 * stack. Note that either way we are going down a level.
 */
                    if (!l.simple() || !r.simple()) {
                         p→push(r, DacStack::pending, s+1);
                         p→push(l, DacStack::limbo, s+1);
                    }
                    else {
                         q→push(l.evaluate(), s+1);
                         q→push(r.evaluate(), s+1);
                    }
/*
 * if we couldn't divide then we evaluate the current node and push it for
 * combining. This should cope with the case when the root node is simple as
 * well as the case when we might not think things are simple but we still
 * can't divide ...
 */
               } else {
                    q→push(n.evaluate(), s); // level is constant
               }
          }
/*
 * if we have a pair of operands to combine ...
 */
          while (q→pair(qpos)) {
               s = q→key(); // we are going up a level
               r = q→pop();
               l = q→pop();
               q→push(l.combine(r), s-1);
          }
     }
/*
 * The last object left is the one we want ...
 */
     return q→pop();
}
```

Figure 2: The stack evaluation algorithm

3.3 Virtual object construction

Passing anonymous objects around a message-passing environment necessitates some means of reconstructing objects based on a message. Coplien [8, p290ff] describes a scheme for implementing exemplar-style programming in C++. The techniques he uses allow for arbitrary classes to register in an exemplar list. When an object is to be created, its construction parameters are presented to each registered exemplar in turn to see whether it can consume the input and return an object. By modifying this operation slightly we can assign each D&C class a type code with which registered D&C exemplars can determine whether an incoming object is of its type and act accordingly.

The advantage of Coplien's approach is that the number of exemplars need not be known by the management code and new classes can be added to the exemplar list, simply by linking in the appropriate library, no code modification is necessary.

3.3.1 Nstreams

Coplien's exemplar example consumed characters from a static buffer. Although it would be possible to do this in a message-passing environment, this sort of input/output operation is far more simply performed using C++ streams. These structures dynamically request input from a stream source in a manner transparent to the user [19] and are designed in such a way that the stream source can be modified relatively easily. Thus we have designed message-passing iostreams called nstreams based on the Trollius^{TM} message passing calls `nsend()/nrecv()`. These structures transparently send and receive network packets when sinking and sourcing information as per standard iostreams.

Unfortunately, iostreams are designed to transmit information in ascii format, which is fine for writing data files to disk or scanning for user input, but which leads to data expansion during message-passing! In order to overcome this, we designed binary iostreams which are constructed from normal iostreams but which force binary reads and writes to be performed, thus optimising the volume of data transmitted.

In addition to this optimisation we wanted to make sure that long messages are transmitted directly rather than being intermediately buffered. This is not a problem except for the fact that these un-buffered messages could potentially overtake the buffered messages. For this reason we make sure that the un-buffered messages are only transmitted when the stream is flushed (implicitly or explicitly) at which point we can ensure that the buffered information is sent first.

3.4 Interface design

In this section we build up a object-oriented D&C interface, gradually incorporating desirable features.

3.4.1 The basic interface

```
class Dac {
  public:
    virtual const boolean simple() const;
    virtual Dac divide(Dac& l, Dac& r) const;
    virtual Dac evaluate() const;
    virtual Dac combine(const Dac& d) const;
};
```

Figure 3: Basic interface

To start with we need the D&C primary functions. In order for the interface to be fully polymorphic, all function arguments and return values must be expressed in terms of a generic D&C object, see figure 3.

```
class Dac {
    friend class DacRep;

  public:
    virtual const boolean simple() const {
        return d_rep→simple();
    }
    virtual Dac divide(Dac& l, Dac& r) const {
        return d_rep→divide(l,r);
    }
    virtual Dac evaluate() const {
        return d_rep→evaluate();
    }
    virtual Dac combine(const Dac& d) const {
        return d_rep→combine(d);
    }
        // reference counting
    Dac() {}
    Dac(const Dac& d) : d_rep(d.d_rep) {
        if (d_rep) d_rep→count++;
    }
    const Dac& operator= (const Dac& d) {
        if (d_rep ≠ d.d_rep) {
            if (d.d_rep) d.d_rep→count++;
            if (d_rep && --d_rep→count ≤ 0) delete d_rep;
            d_rep = d.d_rep;
        }
        return *this;
    }
    ~Dac() {
        if (d_rep && --d_rep→count ≤ 0) {
            delete d_rep;
        }
        d_rep = d;
    }

  protected:
    DacRep *d_rep;
};

class DacRep : public Dac {
    friend class Dac;

  public:
    DacRep() : count(1) {}
    unsigned char count;
};
```

Figure 4: Reference counting

3.4.2 Reference counting and labelling

A little thought shows that this immediately creates a problem. All polymorphic behaviour in C++ is exhibited through operations on *references* or *pointers* to objects. If we are to return from functions, objects which exhibit polymorphic behaviour, then it is references or pointers that must be returned. Unfortunately, this creates a memory-management nightmare as all objects referenced in this manner would have to have global lifetime. Fortunately there is a way around this problem; by using Coplien's [8, p133] envelope and letter idiom we can make the interface simply a label for a real object that can be changed readily without affecting the label. So in this instance our label is "D&C object" but what the real object might be is not pre-determined by the interface. The other advantage of this idiom is that we can incorporate reference counting for the actual objects so that memory-management is no longer a problem and object assignment and copying is cheap.

Coplien's original design had class member functions delegating their operation through a pointer to an object of the same class; rather like a klein bottle. However, we require that all D&C objects have some global information - like size - and we do not want to make the "labels" any larger than necessary. For this reason we delegate the D&C interface's functions through a pointer to an object of a class derived from the interface class. See figure 4.

3.4.3 Mix-in support

We now wish to incorporate support for mix-ins. Mix-ins combine class functionality through the use of multiple-inheritance and a common base class. For example a base class X might declare the functions $a()$ and $b()$, and two other classes A and B, derived from X, might define one each of these functions. If we were then to require a class C that required the functionality of $a()$ or $b()$ or both, we could then derive C from X, to make it have the interface of X, and additionally make C "mix in" A or B depending on the functionality required.

In order for mix-ins to work, the base class defining the interface must must be a virtual ancestor of all its derived classes. This means that all descendants of the base class share a single instance. The current D&C design makes this constraint easy to satisfy as we just make the envelope (`Dac`) a virtual ancestor of the letter (`DacRep`). Any mix-in definitions must then be derived from `Dac`.

This raises one other issue. Using mix-ins can be made safe by making the base class functions all pure virtual. This means that the base class does not define these functions, and the compiler forces the programmer to define these functions in derived classes. Unfortunately, our "klein bottle" design means that the D&C base class functions are already defined to delegate their operation to the derived letter. Thus if a programmer happens to forget to really define one of these functions - and the compiler will allow him to do this - any call will default to the base class definition. This obviously gives rise to a never-ending recursion that will only terminate when the process' stack space is exhausted. We can overcome this problem we introducing an intermediate "pure" interface that we place between the envelope and letter in the inheritance hierarchy. All mix-in definitions can then be derived from this pure interface and the compiler will then complain unless all functions are properly defined. See figure 5.

3.4.4 Exemplar support

Finally we must add exemplar support so that objects can be transmitted around a message passing environment. The functionality required by the interface is given in Coplien and comprises three functions: one to find a type match between a list of registered exemplars and an input stream which subsequently calls the relevant object constructor, another - simply a virtual placeholder - to output a type identifier and associated object to a stream and a third to register an exemplar in the exemplar list.

The exact mechanism of these three is not important here, however their effect upon derived classes is. A D&C class with input/output ability is characterised by three associated functions. One a virtual function which returns an object of its type constructed from an input stream, another the associated input stream based constructor, and a third an output function for the

```
class Dac {
    friend class DacRep;

  public:
    virtual const boolean simple() const {
        return d_rep→simple();
    }
    virtual Dac divide(Dac& l, Dac& r) const {
        return d_rep→divide(l,r);
    }
    virtual Dac evaluate() const {
        return d_rep→evaluate();
    }
    virtual Dac combine(const Dac& d) const {
        return d_rep→combine(d);
    }
        // reference counting
    Dac() {}
    Dac(const Dac& d);
    const Dac& operator= (const Dac& d);
    ~Dac();

  protected:
    DacRep *d_rep;
};

class DacPure : public Dac {
  public:
    const boolean simple() const=0;
    Dac divide(Dac& l, Dac& r) const=0;
    Dac evaluate() const=0;
    Dac combine(const Dac& d) const=0;
    DacPure() {}
};

class DacRep : virtual public DacPure {
    friend class Dac;

  public:
    DacRep() : count(1) {}
    unsigned char count;
};
```

Figure 5: Mix-in support

```
class ADacClass : public DacRep {
  protected:
    Obj anotherMember
    int aMember;
        // construction from an input stream
    ADacClass(ibstream& i) : DacRep() , anotherMember(i) {
        i >> aMember;
    }
        // exemplar registration constructor
    ADacClass(ExemplarConstructor e) : DacRep(e) {}
        // "virtual" construction from in input stream
    virtual DacRep* scan(ibstream& i) {
        return new ADacClass(i);
    }
        // output to a stream
    virtual void spawn(obstream& o) {
        o << anotherMember << aMember;
    }
        // registered exemplar
    static ADacClass* exemplar;
};

ADacClass* ADacClass::exemplar = ::new ADacClass(exemplarConstructor);
```

Figure 6: Exemplar support

object. In addition to these each class must have a static member of its own class which serves to register the class in the exemplar list. See figure 6.

Unfortunately, as can be seen although flexible, this approach to object transmission is rather tedious from a programmers point of view. We will examine the relief of this problem in section 4.

3.5 Object oriented D&C and the actor model

This section relates object-oriented D&C to the actor model and shows how delayed evaluation becomes a natural extension to object-oriented D&C in this framework.

The actor [1] object-oriented programming model represents programs as an interacting set of computational agents which map incoming communications to 3-tuples consisting of:

1. a finite set of communications sent to other actors;
2. a new behaviour; and,
3. a finite set of new actors created.

Now that the D&C primary functions (figure 3) are part of a D&C object rather than separate from it, we can see that they constrain D&C objects to a form of these rules. The rules are limited by the requirements of the D&C algorithm [1], namely that incoming communications are mapped to a finite set of new D&C objects created where the set is one of:

- a fixed number N created through division for N-ary D&C;
- a single object created through evaluation;
- a single object created through amortization of other objects, where the incoming communication contains one or more other objects.

Note that the second is a call-by-value form of (2) above. Although a D&C object could conceivably change its type internally - since its interface is purely a "label" - it is intentionally made difficult by the `const`-ness of the primary functions.

[1]This is really what we would expect, the actor model being so powerful.

One might think that not changing type would lead to a proliferation of D&C objects; and for the recursively and tree evaluated cases this is so. However, with the stack based algorithm, the number of D&C objects is kept to be minimum by virtue of the fact that there are no used D&C objects serving as placeholders. Thus the rules above are all extended such that:

- no object changes its behaviour; and,
- an object's existence is terminated upon acceptance of a communication

The only disadvantage that can be envisaged is that of not being able to easily reuse object memory allocations. The advantages are many, not least the simple programming perspective that is realised.

In fact we could rewrite the rules so that changing an object's behaviour became integral to the operation; so that division, for example, would involve an object changing its state and creating $N-1$ new objects. However, this removes the uniformity of the approach by making a distinction between types of children.

3.5.1 A homogeneous approach

In our previous implementation D&C objects had to be clumsily represented in terms of evaluation objects and result objects. Now the types of object are homogeneous: all are D&C objects but the type of D&C object can be transformed when a change of functionality is required. This eliminates the need for functional baggage that would clutter the definition of D&C objects. It also means that changing form can be accommodated easily.

For example if we have some operation that involves a matrix changing into a scalar, we can separate the two distinct types by defining a D&C object that deals with matrices, a D&C object that deals with scalars and a mapping between the two. If we then want to define an operation that involves only matrices we are not denied the possibility of using the matrix type object. Previously we would have had to define a single D&C object that knew about matrices and scalars and defined operations on both - and the object would be specific to that single operation.

3.5.2 Delayed evaluation

In viewing object-oriented D&C as a specialised actor system we have made no reference to mapping (1) above for general actors. Allowing D&C objects to be replicated, amortized or transformed purely fits within the confines of creating a finite set of new actors (3). However, delayed D&C evaluation features, described in [16], operate by combining objects of *differing* types prior to D&C evaluation, and subsequently performing this evaluation upon the aggregate object using delegation [4]. This possibility fits neatly into mapping (1), as an aggregate D&C object would first create a finite set of children and then pass on the communication to its constituent members.

The obvious application for this is in evaluating arithmetic expressions where, for example, we might wish to evaluate the matrix expression $A+B+C$. Delaying the evaluation of this expression means producing an aggregate - in our case D&C - object, and delegating calls to the aggregate's interface to the individual object's functions. See figure 7.

3.5.3 Envelope/letter advantages for delayed evaluation

In using delayed evaluation we have two objectives:

- constructing the aggregate; and,
- evaluating the aggregate.

In the example given above the aggregate is constructed using overloaded arithmetic operators. However, previously there was no clear way in which to organize an object hierarchy that allowed the interactions of evaluation and construction to be separated. For example in the expression $A = B + C * D$ where $A - D$ are general objects. The evaluation sequence would have been:

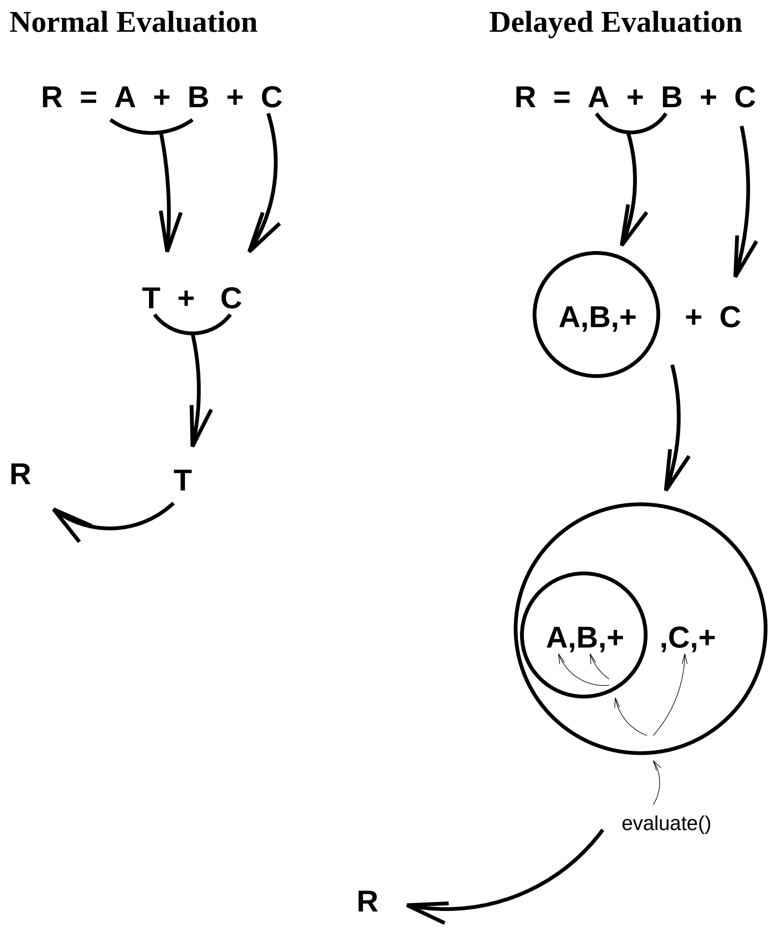

Figure 7: Delayed evaluation

$$
\begin{aligned}
A &= B + C * D \\
A &= B + \mathit{TimesObj}[C, D] \\
A &= \mathit{AddObj}[B, \mathit{TimesObj}[C, D]]
\end{aligned}
$$

where $O[P_1, \ldots, P_n]$ denotes an object O containing objects $P_1, \ldots, P_n$.

Whereupon the compound *AddObj* is evaluated and assigned to A. However, in order for this sequence of events to operate correctly, *AddObj* and *TimesObj* must be derived from some generic *ArithObj* which defines the addition and multiplication operators; otherwise *TimesObj* and *AddObj* would need to define every possible operator combination. During subsequent D&C evaluation these functions are completely redundant. In addition, defining any new operators would necessitate recompilation of all the arithmetic classes.

With the envelope/letter idiom all these problems go away. The aggregate constructors can be defined within a wrapper derived from the envelope class `Dac`, while the aggregates themselves can be defined normally - derived from the letter class `DacRep`. The wrappers then serve purely to build an aggregate object, old wrappers being discarded when they are redundant:

$$
\begin{aligned}
\mathit{ArithWrap}[A] &= \mathit{ArithWrap}[B] + \mathit{ArithWrap}[C] * \mathit{Arithwrap}[D] \\
\mathit{ArithWrap}[A] &= \mathit{ArithWrap}[B] + \mathit{ArithWrap}[\mathit{TimesObj}[C, D]] \\
\mathit{ArithWrap}[A] &= \mathit{ArithWrap}[\mathit{AddObj}[B, [\mathit{TimesObj}[C, D]]]]
\end{aligned}
$$

This also means that the assignment operator can be completely generic and included in the definition of the wrapper.

Of course these features are a subset of the possibilities under the actor regime and thus delayed evaluation represents a natural usage of object-oriented D&C rather than an added feature.

4 Translation versus code insertion

In this section we briefly describe an approach to overcome some of the tediousness of programming with D&C objects.

Many parallel object-oriented systems [9], with some exceptions [3], use a language translator to parse their particular flavour of the target language. For C++ at least, this is not particularly desirable as one has to have the translator as well as the target language compiler *and* ensure that the two are compatible. In addition the programmer has to learn the new constructs and - as shown by the C++ standardization effort - these constructs may well just be unnecessary syntactic sugar.

Such is true of our object-oriented D&C additions to C++ - we can incorporate these constructs using existing language features. We could augment the C++ language definition in order to make the programming of these constructs easier, but with the portability problems given above. In addition it is not always clear what a programmer's intentions for a D&C object are, and enforcing a translator-based regime could yield an undesirable degree of inflexibility.

However, making C++ objects into D&C objects *is* slightly cumbersome and some sort of automation would be desirable. If we are not going to write a translator then the obvious solution is to automatically insert the required code directly into the source - and we can achieve this by using the GNU emacs editor.

4.1 Dac-mode

GNU emacs differs from most editors in that it is almost infinitely reconfigurable through the use of its internal lisp interpreter. It also has a powerful regular expression library. These two features, used in conjunction with each other, mean that quite complicated language constructs can be parsed by emacs. It is even possible to execute lisp commands non-interactively so that emacs can be used as a pseudo-translator if so desired.

Thus we have written an emacs "mode" - `dac-mode` - that allows a user to modify a C++ file by inserting text relevant to D&C evaluation. The only constraint is that data members of D&C objects, that require transmission, be delimited by special comments. Since the inserted text is editable, any wrong assumptions made by the lisp code can be corrected by the programmer. The lisp code will not attempt to update D&C-relevant constructs that are already in existence.

The whole environment manages to maintain programming flexibility whilst removing repetitious work.

5 Application to grammar modelling for speech recognition

In this section we describe the use of tree language models for speech recognition and the implementation of one particular tree growing algorithm using object-oriented D&C. We then consider a parallel improvement to the algorithm using nested D&C.

In the field of automatic speech recognition (ASR), language models, which attempt to provide an accurate prediction of the next word in a sequence, are important for good overall performance of any ASR system.

Decision trees (figure 8) are one possible type of language model [21]. A tree T may be viewed as a set of nodes $T = \{t_0, t_1, \ldots, t_n\}$, with t_0 reserved as the root node. The input to the tree $w_{j-n}, w_{j-n+1}, \ldots, w_{j-1}$ is a string of the previous n words, and the output from a leaf of the tree is a probability distribution over the possible predicted words $\hat{W}$. Starting from the root node

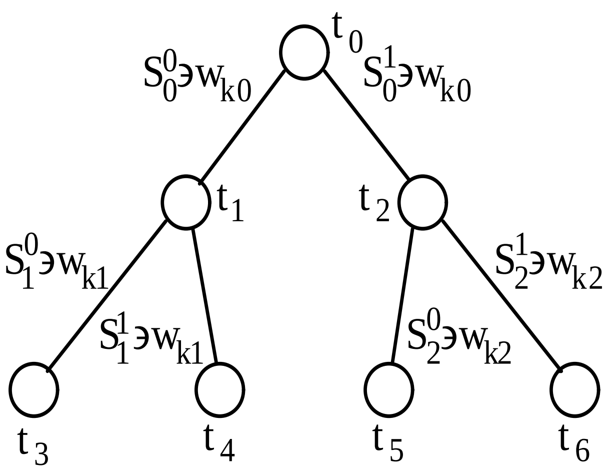

Figure 8: Tree

of the tree, at each non-terminal node t_c, a binary function Q_c is performed on w_k, one of the n words of the input. This function takes the form: 'is $w_k \in S_c^j$?', where S_c^j is the jth set of words (for binary branching $0 \leq j < 2$). If the result is true, the left branch from the node is followed otherwise the right branch. This is repeated until a terminal node is encountered.

In growing a tree, the objective lies in minimizing risk for a given cost constraint. Each step consists of splitting some terminal node t into two children in order to maximize a merit function, merit(.), for some test Q, which will lead to a tree which satisfies the global constraints.

Due to the computational complexity of growing optimal trees, practical design procedures (deciding on the best splits and hence the sets S_c) are invariably steepest-descent based. However, even practical approaches are extremely computationally intensive and a good candidate for parallel evaluation.

For the purposes of this work, a clustering-based algorithm due to Chou [7] and implemented serially by Waegner and Young [21] was employed. Vectors consisting of the conditional probabilities of the predicted words, conditioned on a particular instantiation of w_k, are iteratively clustered into bins. The clustering algorithm is classically D&C with *all* computation being done in the divide phase of the algorithm.

5.1 Parallel implementation of Chou's algorithm

The strategy we employed in implementing a parallel version of Chou's algorithm is important as it reflects a general strategy for writing programs using object-oriented D&C. Our initial strategy entailed the following:

1. Identify how the algorithm is "divide-and-conquer-able".
2. Identify data that child nodes require from their parents and vice versa and encapsulate these in an object definition (class) derived from `DacRep`.
3. Identify data required by all nodes and add these to the class as static members.
4. Add the required primary D&C functions to the class definition through: inheritance if they are standard, or by writing them if they are not.
5. Format the class in the emacs editor with `dac-mode` so that objects of that class are usable in a message passing environment.
6. Define any initial conditions required.

Our original aim was to use much of the serial code, which was written in C, as it stood. Much of the original code was concerned with actual D&C evaluation, so this could be thrown out in favour of our parallel framework; leaving only the guts of the partitioning algorithm itself. This, we hoped, could be left as a C module that only required linking in to the parallel framework. For an initial attempt this proved possible - if slightly clumsy - by defining appropriate conversion operators from C++-world array objects and the like to C-world arrays. However, when it came to more complicated implementations, it was found far easier to recompile the C-code with a C++ compiler thus giving the C-module easy access to all the C++-world data structures. Although this approach does not work with programs for which the source is unavailable, it is vastly preferable from a flexibility point-of-view.

The clustering algorithm made up the flesh of the `divide()` phase of the algorithm and in terms of writing functionality there was little else to do. Since we are *growing* these trees and therefore have no interest in combining anything, `evaluate()` was made to return a null D&C object. This object did nothing except satisfy the requirements for successful completion of the evaluation algorithm; i.e. that it could be transmitted and combined.

5.1.1 Tree dumping

One other important issue needs considering. Data is generated from each non-terminal and terminal node in the tree, but how can we guarantee that the tree constructed on disk is ordered correctly ? The data cannot be written to a single file as there is no way to tell size or shape of a tree before evaluation - it is an unbalanced D&C computation. The only obvious solution is to write a separate file for each sub-task evaluated. Several methods were tried of naming these files so that their order could be determined easily, but this required preorder node-number information which would have been available under a serial implementation but was indeterminate under the parallel implementation.

In the end the nodes were numbered in level order - a numbering scheme which is independent of the size of the tree. The output files were identified by the root node that was processed in a sub-task, and then the files were parsed using Sedgewick's stack-based pre-order tree-traversal algorithm in order to discover their concatenation order.

Since tree growing is more generally applicable to D&C than just for language modelling, this technique should be more widely applicable.

The speedup results for the initial implementation are given in section 6.

5.2 Improved parallel performance using nested D&C

In this particular tree-growing application we are trying to partition a corpus into two at each non-terminal node. The partitioning algorithm is of time complexity $\mathcal{O}(N)$. So it can be seen that at the start of evaluation, when there is a single large context, the (serial) partitioning of this will dominate the execution time of the whole. At the end of execution the partitioning is of many small contexts but these are done in parallel.

Thus it would be desirable to parallelise the initial partitioning somewhat to achieve better processor utilisation. The partitioning algorithm involves looping over the entire corpus for each member of a given context. Thus, if we were to parallelise across the context elements we should be able to improve the performance of the partitioning by a factor close to the number of elements in a context.

5.2.1 Nested Implementation

In designing this extra level of parallelism we adopt an approach similar to that given above. However, this time we are merely iterating over an integer so a general D&C iterator can be used to do much of the programming leg-work. From a functionality point-of-view we merely have to separate the loop and the partitioning algorithm, and assign each to a class.

At this point we can make use of inheritance to simplify the class definition. The tight coupling of the outer loop with the partitioning algorithm also implies a tight coupling of data and we can thus factor the data common to both control structures into a single base class. It is worth considering this property for a moment, as it is an important one for our object-oriented approach. In general, once a D&C strategy has been decided for a particular problem, then the next candidates for parallelism are any outer loops in the D&C primary functions. However, the only required additions to the locality of reference *already provided* by the D&C partitioning, will be data local to the D&C primary functions. Thus, the factoring together of common data will be a common occurrence in making use of nested D&C, and it is only by virtue of the object-oriented approach that this is made simple.

The fact that the parallelised looping is a sub-task of the parallelised clustering is coped with is because of the generality of the stack-based approach described in section 3.2.

A comparison of the performance of the two approaches is given in section 6.

6 Performance of the system

In this section we give performance results for the simple and nested implementations of the tree-growing algorithm. We then consider some implications of these results and give some improved results through an understanding of D&C systems in general.

In the ensuing discussion it is important to realise that tree-growing is a true D&C application, in other words the appropriate method of serial evaluation is D&C. For this reason there is no need to compare the parallel implementation with a control as the uniprocessor case is the same, performance wise, as the serial case; neglecting processor usage due to scheduling. It is also important to realise that partition size [18] is an integral part of the algorithm, thus finding an optimal partition for parallel evaluation must be entirely separate from specifying a D&C partition.

6.1 Non-nested implementation

Figure 9 gives the speedup results for the non-nested implementation using a small corpus of 16384 words (16376 contexts) and a cluster size of 64 words. The experiments were conducted on a toroidal mesh of T800 transputers running the Trollius^{TM} operating system.

As can be seen performance is good though, as noted in [16], there are some discontinuities in the curve. As we explained [16] these discontinuities can be attributed to the connectivity of a transputer node. The depth-first evaluation scheme that we employ means that the size of task offloaded to neighbouring processors decreases in size as $1/2^n$, where n is the task number. So the first processor receives 1/2 the problem, the second 1/4 and so on. However, once the nearest neighbours have been exhausted due to the *single-steal* rule [14], then the speedup is limited to some extent by the size of problem remaining on the root processor. Thus we would expect the speedup to be limited to $2^{n_{links}}$; however many processors there are. This is borne out by figure 9. The processor numbering scheme we have adopted means that there is only one link available for 1-4 processors, two for 5-12 processors and three for 12-16 processors. The breakpoints for the number of links correspond to the discrepancies in the graph. The sudden drop in performance after each breakpoint can be attributed to a large proportion of the problem being scheduled to a solitary processor. This processor is connected to the newly available link with no other neighbours to offload to.

However, the results are inconclusive and we will present more compelling evidence below.

6.2 Nested implementation

In figure 9 the speedup results for the nested implementation are presented, together with the non-nested implementation. As can be seen, the nested implementation gives a significant improvement over the simple implementation. We would also expect this improvement to increase as the problem size is increased and parallel overheads become less significant. However, we would expect the maximum performance increase to have an upper bound equal to the maximum speedup given by the nested performance only - in this case 8.

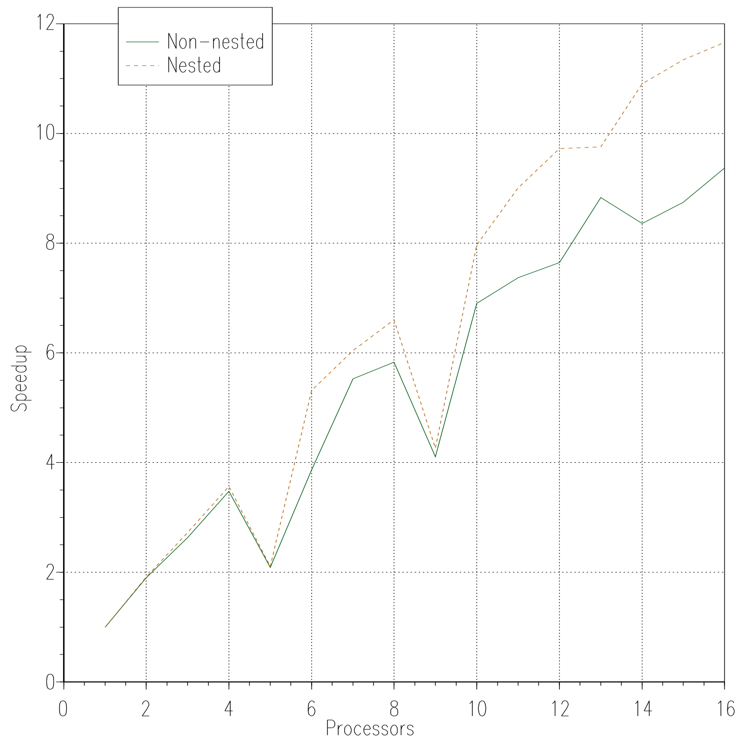

Figure 9: Nested and non-nested performance

6.3 Performance improvements

We have surmised that the performance of a depth-first D&C system will be limited by the connectivity of the processors, and we have provided some scanty evidence for this. Therefore, for a 4-link processor, we would expect the performance to be largely unaffected by an increase in the number of processors past 16, if the connectivity of the processors remained unchanged. If we increase the number of processors from 16 to 32, we obtain the results given in figure 10.

Obviously the system as it stands is not at all scalable, and scalability is highly desirable. However, if our connectivity argument is right then we can achieve better scalability by utilising a processor network with a higher degree of connectivity. Specifically, if we require each processor to have access to a larger number of neighbours, then a hypercube architecture is the obvious candidate. Furthermore, if speedup is limited to $2^{n_{links}}$ and speedup is also limited to $n_{processors}$ then we will attain maximum speedup for:

$$\begin{aligned} 2^{n_{links}} &= n_{processors} \\ n_{links} &= log_2(n_{processors}) \end{aligned}$$

which is true for all hypercubes.

6.3.1 Virtual hypercubes

It is clear that a hypercube interconnection network is desirable for D&C systems, but a transputer only has four links! Ideally we would use a network of TI C40's which would yield a physical hypercube of maximum degree 6; but is this really necessary? We can achieve higher dimensionality hypercubes we forming groups of physically connected hypercubes and connecting these to each other by means of "virtual" - or multi-hop - links. Arranging for this to happen is simple if the

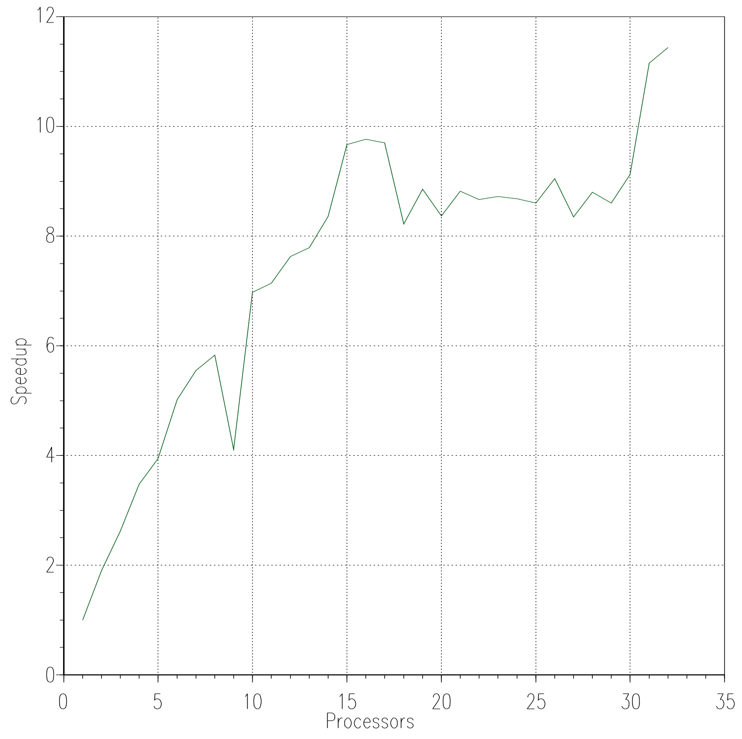

Figure 10: 32 processors non-nested performance

real hypercubes are numbered in units of 16. In this case a processor will have virtual neighbours at processor identifiers:

$$(procid + 2^{i+4})\ MOD\ 2^{i+5}$$

where $i = 0, 1, \ldots$.

Applying this to the 32 processor case yields the results given by figure 11. As can be seen the speedup is now much more linear with a greater maximum speedup.

Parallel overheads could possibly account for the decreasing efficiency. If this is the case then increasing the problem size should result in increased efficiency.

6.3.2 Problem scaling to reduce overheads

By increasing the size of problem we obtain the results given by figure 12. As can be seen increasing the problem size *does* increase the efficiency of the system. However, a corpus of 100000 words was the largest problem that could be tried with the memory available, and it is not clear that the maximum efficiency has been obtained with a problem of this size.

Interestingly enough adding a further dimension to the hypercube (figure 13) for the 100000 word case yields little further speedup, performance actually decreasing after 46 processors. It would therefore appear that tuning the D&C partition would be a sensible thing to do, to limit the number of processors used, as well as evaluating still larger problems.

7 Theoretical speedup

In this section we present some theoretical speedup results for D&C and relate them to the practical results presented in section 6. Lewis et al [12] gave the maximum possible speedup for D&C

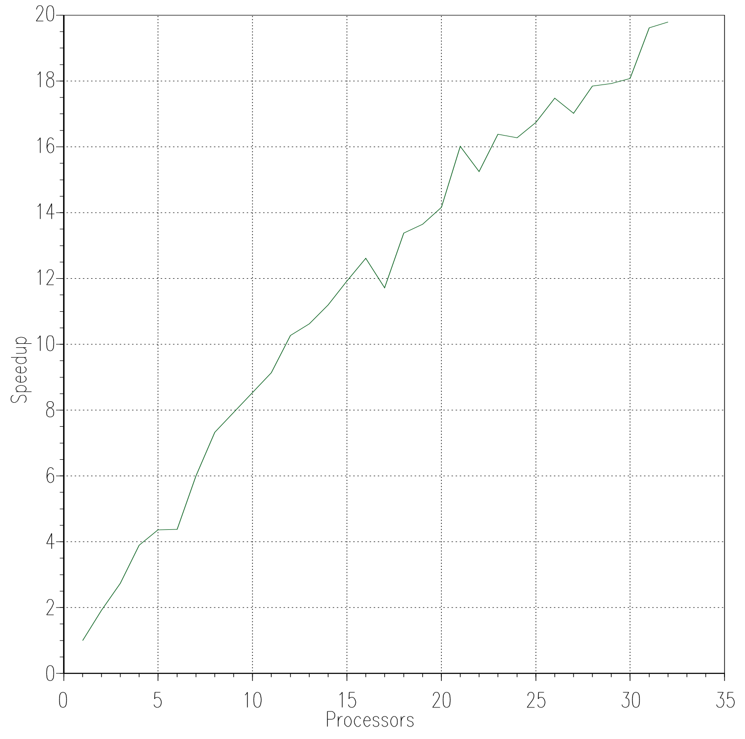

Figure 11: Virtual hypercube performance

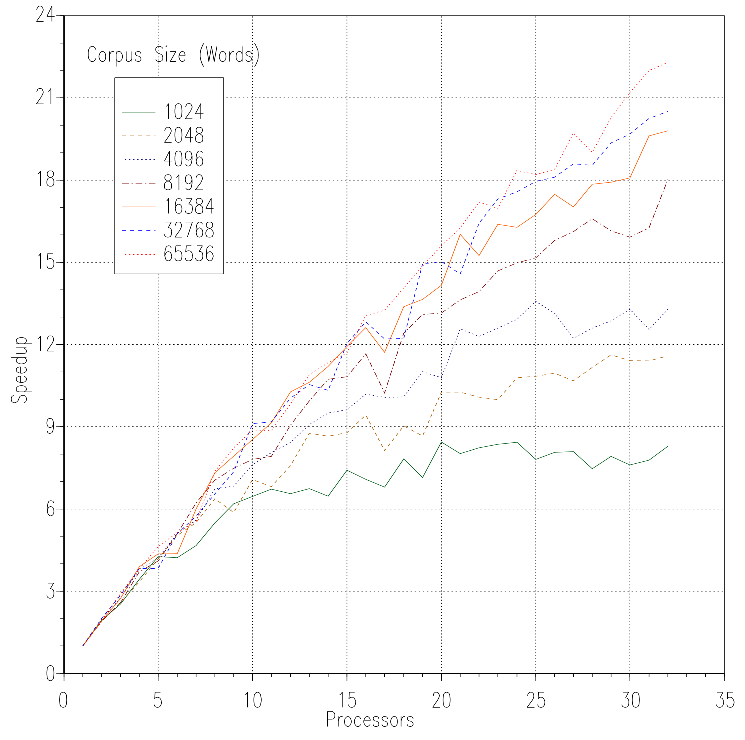

Figure 12: Virtual hypercube performance with increasing problem size

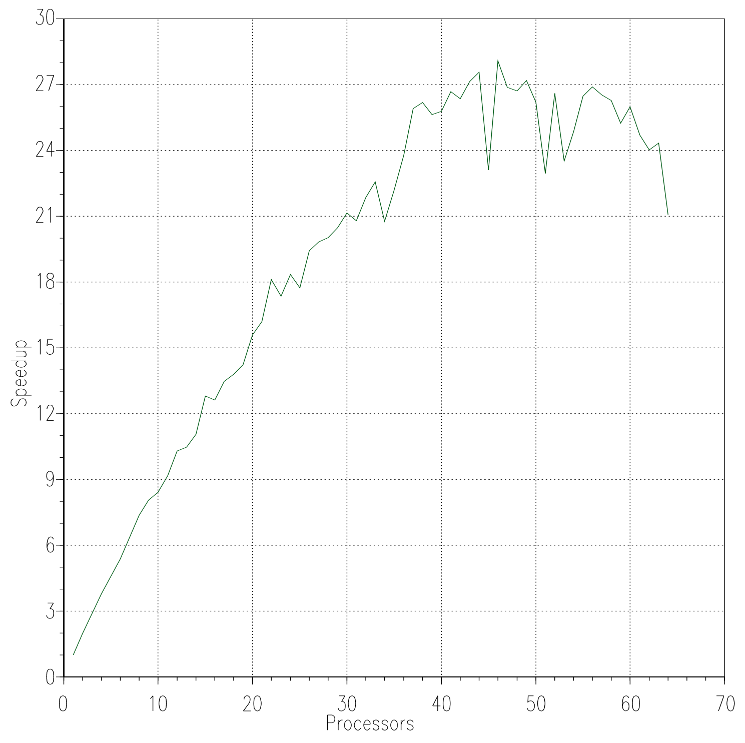

Figure 13: Degree 6 virtual hypercube performance

problems as N/log_2N. However, our results exhibit rather better performance than this for an algorithm with classical D&C properties. Therefore, we develop a better model of D&C which takes into account the effects of problem scaling. These results tie up well with the practical results of section 6. We then develop a more complex model which takes into account the possibility of variable divide time; a property which the tree algorithm should exhibit. However, the theoretical results are much worse than our experimental results and so we conclude that this property is not significant for the tree algorithm.

7.0.3 Framework

In the execution of a D&C problem, we define $d = 1 \ldots D$ as the depth down the D&C tree so that there are:

$$n_D = 2^D \tag{1}$$

terminal nodes in the tree if the tree is balanced. This also means that there are:

$$N = 2^{D+1} - 1 \tag{2}$$

nodes in the tree in total, and

$$\begin{aligned} 2^D - 1 &= n_D - 1 & (3)\\ &= n_t & (4) \end{aligned}$$

non-terminal nodes. We note also that the maximum depth:

$$\begin{aligned} D &= log(n_D)/log(2) \\ &= log_2(n_D) & (5) \end{aligned}$$

We define:

$$\begin{aligned}
t_d &= \text{time to perform a single divide} \\
t_c &= \text{time to perform a single combine} \\
t_f &= \text{time to perform a single evaluation}
\end{aligned}$$

but we will often assume that $t_d = t_c$.

We assume that communication time is zero and that there is no limit to the availability if processors.

7.1 Basic D&C

We first investigate a simple theoretical model of D&C, where the divide time dominates, which yields the result given by Lewis et al [12].

We note that it is not possible to make use of more than n_D processors. Thus if we wish to evaluate all nodes in parallel the execution time will be:

$$\begin{aligned}
T(n_D) &= Dt_d + t_f + Dt_c \\
&= 2Dt_d + t_f \\
&= 2t_d log_2(n_D) + t_f \qquad (6)
\end{aligned}$$

if $t_d = t_c$. This is because the time to evaluate in parallel all nodes at depth $= k$ will be t_d and all nodes at depth k cannot be evaluated before nodes at depth $k-1$.

The serial execution time for this problem is given by:

$$\begin{aligned}
T(1) &= n_t t_d + n_t t_c + n_D t_f \\
&= (n_D - 1)(t_d + t_c) + n_D t_f \\
&= 2t_d(n_D - 1) + n_D t_f \qquad (7)
\end{aligned}$$

thus the speedup,

$$\begin{aligned}
S &= T(1)/T(N) \\
&= \frac{2t_d(n_D - 1) + n_D t_f}{2t_d log_2(n_D) + t_f} \qquad (8)
\end{aligned}$$

We note that the speedup is n_D for the best case of $t_d = 0$. The worst case is for $t_f \to 0$, which gives:

$$\begin{aligned}
\lim_{t_f \to 0} S &= \frac{2t_d(n_D - 1)}{2t_d log_2(n_D)} \\
&= \frac{n_D - 1}{log_2(n_D)} \qquad (9)
\end{aligned}$$

which is that given by Lewis et al [12].

7.2 D&C with problem scaling

The result obtained above obviously scales very badly with increasing numbers of processors. However, as demonstrated by Gustaffason et al, the sensible thing to do is to scale the *problem* relative to the number of processors. Thus we develop a theoretical model based on this premise.

If we scale the problem relative to the number of processors then we can view the execution on $n_p = 2^p$ processors as a purely parallel evaluation up to depth p, and a serial evaluation of n_D/n_p nodes.

In this case the serial execution time is the same as above, but the parallel execution time is given by:

$$\begin{aligned} T(N) &= 2t_d log_2(n_p) + \frac{n_D}{n_p} t_f + 2t_d \left(\frac{n_D}{n_p} - 1 \right) \\ &= 2t_d log_2(n_p) + \frac{n_D}{n_p}(t_f + 2t_d) - 2t_d \end{aligned} \tag{10}$$

so that the speedup is:

$$S = \frac{2t_d(n_D - 1) + n_D t_f}{2t_d log_2(n_p) + \frac{n_D}{n_p}(t_f + 2t_d) - 2t_d} \tag{11}$$

If we introduce a *scaling ratio*:

$$\alpha = \frac{n_D}{n_p} \tag{12}$$

$$n_D = \alpha n_p \tag{13}$$

then the speedup becomes:

$$S = \frac{2t_d(\alpha n_p - 1) + \alpha n_p t_f}{2t_d log_2(n_p) + \alpha(t_f + 2t_d) - 2t_d} \tag{14}$$

$$\begin{aligned} \lim_{t_f \to 0} S &= \frac{2t_d(\alpha n_p - 1)}{2t_d log_2(n_p) + \alpha 2t_d - 2t_d} \\ &= \frac{\alpha n_p - 1}{log_2(n_p) + \alpha - 1} \end{aligned} \tag{15}$$

Thus by increasing α we can produce a more linear speedup than the simple D&C case of section 7.1. Figure 14 shows theoretical speedups for varying α. These results tie in nicely with the practical results presented in figure 12. The case $\alpha = 1$ is equivalent to the simple case, given above. It can be seen, therefore, that problem scaling is a good thing to do with D&C systems, as this yields more efficient speedups.

Note that these are worst case results, in practice $t_f \neq 0$ and therefore the $n_p t_f$ term will be significant, increasing the linearity of the speedup.

Intuitively this is the result we would expect; for the case $\alpha = 1$ processor utilisation is only at a maximum when nodes at maximum depth are being evaluated. However, if $\alpha > 1$ then processor utilisation is at a maximum from depth p onwards.

7.3 D&C with variable divide time

The tree algorithm presented in section 5 processes a context of ever decreasing size. The algorithm involves looping over each member of the context; so we would expect the iteration time - the divide time - to decrease as the algorithm progresses. We now analyse the theoretical speedup expected from systems with this property.

7.3.1 Basic algorithm

The divide time should vary as $1/n_d$ where n_d is the number of nodes at depth d. If we therefore take t_d as the maximum divide time, then the actual divide time will be t_d/n_d. Thus the serial execution time for this case will be:

$$\begin{aligned} T(1) &= \left[\sum_{i=0}^{D-1} \frac{t_d 2^i}{2^i} + \sum_{i=0}^{D-1} \frac{t_c 2^i}{2^i} \right] + 2^D t_f \\ &= D(t_d + t_c) + 2^D t_f \\ &= 2t_d log_2(n_d) + n_d t_f \end{aligned} \tag{16}$$

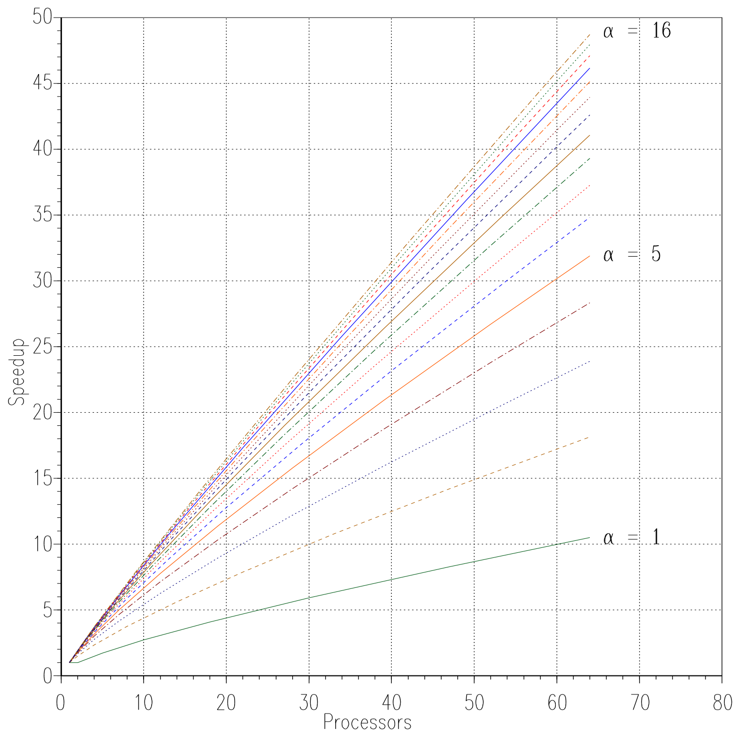

Figure 14: Performance with varying scaling ratio

and the parallel execution time will be:

$$
\begin{aligned}
T(N) &= \left[\sum_{i=0}^{D-1} \frac{t_d}{2^i} + \sum_{i=0}^{D-1} \frac{t_c}{2^i} \right] + t_f \\
&= 2t_d \sum_{i=0}^{D-1} \frac{1}{2^i} + t_f \\
&= 2t_d \left(\frac{1 - 1/2^D}{1 - 1/2} \right) + t_f \\
&= 4t_d(1 - 1/n_d) + t_f \qquad (17)
\end{aligned}
$$

So the speedup will be:

$$
\begin{aligned}
S &= T(1)/T(N) \\
&= \frac{2t_d log_2(n_d) + n_d t_f}{4t_d(1 - 1/n_d) + t_f} \\
\lim_{t_f \to 0} S &= \frac{log_2(n_d)}{2(1 - 1/n_d)} \qquad (18)
\end{aligned}
$$

This is a very poor result as for large n_d the speedup is $log_2(n_d)$. Thus, as previously, we must look at variable divide time with problem scaling.

7.3.2 Variable divide time with problem scaling

In this instance we can again view the parallel execution as purely parallel up to n_p nodes and then serial for n_d/n_p nodes, ignoring any t_f terms at depth n_p since they are not applicable. We note also that the maximum divide time at depth p will be t_d/n_p. Thus:

$$\begin{aligned}
T(N) &= 4t_d(1-1/n_p) + 2\frac{t_d}{n_p}log_2(n_d/n_p) + \frac{n_d}{n_p}t_f \\
&= 4t_d(1-1/n_p) + 2\frac{t_d}{n_p}log_2(\alpha) + \alpha t_f \\
&= 2\frac{t_d}{n_p}\left(log_2(\alpha) - 2\right) + 4t_d + \alpha t_f \qquad (19)
\end{aligned}$$

The serial execution time is the same as above so the speedup is:

$$\begin{aligned}
S &= T(1)/T(N) \\
&= \frac{2t_d log_2(n_d) + n_d t_f}{2\frac{t_d}{n_p}\left(log_2(\alpha) - 2\right) + 4t_d + \alpha t_f} \\
&= \frac{2t_d log_2(\alpha n_p) + \alpha n_p t_f}{2\frac{t_d}{n_p}\left(log_2(\alpha) - 2\right) + 4t_d + \alpha t_f} \qquad (20) \\
\lim_{t_f \to 0} S &= \frac{2t_d log_2(\alpha n_p)}{2\frac{t_d}{n_p}(log_2(\alpha) - 2) + 4t_d} \\
&= \frac{n_p log_2(\alpha n_p)}{log_2(\alpha) - 2 + 2n_p} \qquad (21)
\end{aligned}$$

Again increasing α will yield better performance but not nearly as good as that for the fixed divide time case. In practice the t_f terms will be significant and performance will be better.

7.4 Application to the tree implementation

The tree growing implementation presented in section 5 is one in which the divide time *is* variable. However, the results presented in section 6 are considerably better than those indicated by equation 21. This is partially due to the nested implementation but also must be due to the fact that $t_f \neq 0$ and that there must be some constant element in the divide time. Presumably this constant element will be less dominant for larger problems so the effect of increasing the scaling ratio will become less effective for larger problems. For the largest problem tried:

$$\begin{aligned}
\alpha &= n_d/n_p \\
&= \frac{100000/64}{64} \\
&= 24.41
\end{aligned}$$

Thus for a fixed divide-time and $t_f = 0$ the maximum expected speedup on 64 processors is:

$$\begin{aligned}
\lim_{t_f \to} S &= \frac{24.41 \times 64 - 1}{log_2 64 + 24.41 - 1} \\
&= 53.09
\end{aligned}$$

and for the variable divide time case:

$$\begin{aligned}
\lim_{t_f \to} S &= \frac{64 log_2(24.41 \times 64)}{log_2(24.41) - 2 + 2 \times 64} \\
&= 5.20
\end{aligned}$$

Which is far less than the results actually obtained (figure 12). So obviously the variable divide time effect cannot be particularly significant for this particular problem.

We may conclude then that D&C can provide much better speedups than that presented by Lewis et al [12], if the problem is scaled relative to the number of processors. However, we note that variable divide time leads to very poor performance even if the problem is scaled. Fortunately, few problems will *only* display properties of variable divide time.

8 Conclusions

We have described the implementation of an object-oriented D&C system together with some of its finer details, and given some results which demonstrate the system's ability to achieve useful speedups.

We can conclude therefore that object-oriented D&C provides a viable way of achieving parallel performance. However, any conclusions about the usability of the system are necessarily subjective - though we have given reasons as to why our approach is preferable to defining yet another dialect of C++.

Within the context of our system, nested D&C evaluation proves to be a useful addition which yields appreciably improved performance. We would hope to increase the applicability of this technique, and others, by increasing the scope of our D&C class library. Some of the technical challenges of implementing this have yet to be addressed, but we believe that the C++ language is powerful enough for our demands.

In conclusion we would say that D&C works and suitable object-oriented programming techniques can increase its applicability and usability.

References

[1] Gul A. Agha. *Actors.* MIT Press, 1986.

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

[3] N. Brian et al. PRESTO: A System for Object-Oriented Parallel Programming. *Software - Practice and Experience*, 18(8):713–732, August 1988.

[4] A. A. Chien and W. J. Dally. Concurrent Aggregates (CA). In *Second Symposium on Principles and Practice of Parallel Programming.* ACM, March 1990.

[5] A. A. Chien, W. Feng, V. Karamcheti, and J. Plevyak. Techniques for Efficient Execution of Fine-Grained Concurrent Programs. In *Yale Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science. Springer-Verlag, August 1992.

[6] Roger S. Chien and Samuel T. Chanson. Distributed Object-Based Programming Systems. *ACM Computing Surveys*, 23(1):91–124, March 1991.

[7] P. A. Chou. Optimal Partitioning for Classification and Regression Trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(4):340–354, April 1991.

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

[9] Andrew S. 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.

[10] Andrew S. Grimshaw. An Introduction to Parallel Object-Oriented Programming with MENTAT. Technical Report TR-91-07, Department of Computer Science, University of Virginia, 1991.

[11] T. J. LeBlanc and E. P. Markatos. Shared Memory vs. Message Passing in Shared-Memory Multiprocessors. In *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing.* IEEE Computer Society Press, December 1992.

[12] T. G. Lewis and H. El-Rewini. *Introduction to Parallel Computing.* Prentice-Hall, 1992.

[13] J. Lim and R. E. Johnson. The Heart of Object-Oriented Concurrent Programming. *SIGPLAN Notices*, 24(4), April 1989.

[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] 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.

[16] A. J. Piper and R. J. Prager. A High-Level, Object-Oriented Approach to Divide-and-Conquer. Technical Report CUED/F-INFENG/TR 98, Cambridge University Engineering Department, 1992.

[17] A. J. Piper and R. J. Prager. A High-Level, Object-Oriented Approach to Divide-and-Conquer. In *The Fourth IEEE Symposium on Parallel and Distributed Processing*. IEEE Computer Society Press, December 1992.

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

[19] Jerry Schwartz. Iostreams Examples. In *C++ Language System Release 3.0 Library Manual*. Unix Systems Laboratories, 1991.

[20] R. Sedgewick. *Algorithms in C*. Addison-Wesley, 1990.

[21] Nick Waegner and Steve Young. A Trellis-Based Language Model for Speech Recognition. In *Proceedings of the 2nd International Conference on Spoken Langauge Processing*, 1992.