SYNTHESISING EFFICIENT, PORTABLE ALGORITHMS FOR PARALLEL ARCHITECTURES

P G HARRISON

Rapporteur: D McCue

Synthesising efficient, portable algorithms for parallel architectures (with emphasis on communication)

Peter G Harrison[1]
Imperial College
London

Abstract

We use a functional programming formalism to specify algorithms and to abstract the execution characteristics of given parallel machines. In particular, a restricted form of functional program - a higher-order function called a skeleton - is associated with each corresponding parallel machine type on which its implementation is known to be efficient. Program transformation techniques are then used to convert general functional programs, or even higher level, non-executable specifications, into equivalent forms expressed using the appropriate skeletons. Portability is achieved in two different ways: by re-playing the transformational development process targeting a different class of skeletons, and by the availability of transformations between the different skeletal forms. We focus on a new class of communication-intensive algorithms and present in detail a parallel implementation of 'quicksort' that runs in $(log\ n)^2$ time on n communicating processors.

1. Introduction

Application development has been well served in the past by the universality of the sequential von-Neumann model of computation. This has conveyed several significant software advantages that have largely gone unrecognised as there has been no alternative:

- As the program model is one to one with the machine execution model a programmer can have confidence that a program will behave as expected and a program's performance can easily be envisaged at the program level.
- Portability is guaranteed between different machines at the language level. This is important not only because it reduces software effort but also because it maintains the relative predictability of performance. A program moved to a sequential machine with a faster

[1] Joint work with David W N Sharp and John Darlington

processor will, almost certainly, go faster without major changes being needed to the algorithms employed.

This convenient picture has broken down with the arrival of parallel machines with diverse architectures with different execution characteristics. The diversity of parallel machine architectures is causing the application development activity to fragment into different, machine-oriented camps, each with their own languages, algorithms and libraries, at the expense of a deeper understanding of the field and the relative potential of different machines. We therefore seek a methodology for programming parallel machines that is not only uniform across all types of known parallel machines and guarantees portability but is also capable of achieving efficient execution on any known parallel machine.

We use functional programming to serve as a general model of parallel computation and abstract the execution characteristics of a given parallel machine with a particular, restricted form of functional program called a *skeleton*. Thus, each skeleton is associated with a corresponding parallel machine type on which its implementation is known to be efficient and even proved correct using transformational technology. Program transformation techniques are then used to convert general functional programs, or even higher level, non-executable specifications into equivalent forms expressed using the appropriate skeletons. Programs expressed as skeletons are then compiled onto the target machine using C as an intermediate code. This idea of limiting programs to a fixed repertoire of algorithmic forms has much in common with the FP school.

Portability is achieved in two different ways: by re-playing the transformational development process targeting a different class of skeletons, and by the availability of transformations between the different skeletal forms. Hence the issues of correctness, portability and efficient implementation (i.e. resource allocation) are tackled in a structured manner in that all decisions made are explicitly incorporated in the transformation process and ultimately become parameters to the skeletons. This is in contrast to the case of hand-crafted code where the decisions made are implicit in an often obscure final program. Finally we observe that since the skeleton interface between specification and architecture is describable in the source functional language, the appropriateness of transformations and algorithms expressed in terms of skeletons can be validated before any parallel hardware is available, by executing on a trusted sequential implementation.

There are several different types of skeleton, corresponding to the diversity of parallel algorithms and architectures. Our current collection of skeletons includes pipelines, process networks, processor farms, divide-and-conquer algorithms and dynamic message-passing architectures. The present paper focuses on a new class of communication-intensive algorithms, developed by David W N Sharp, for the last skeleton in this list. The synthesis of such an algorithm is described for 'quicksort'.

We start from an initial specification of the problem given as a set of recursive functions which are standardised to correspond directly with the architecture. The architecture specification helps to guide the synthesis towards an algorithm which exploits the communication facilities available. During the synthesis interprocessor communication is introduced by strengthening the problem specification : rules are introduced describing how receipt of one message should give rise to another. The new message contents can then be used by the processor that receives them to replace some local calculation. Thus the amount of computation is reduced and the problem is solved by cooperative message-passing.

Traditionally quicksort [Hoare 62, Sedgewick 78] has been considered to be a highly dynamic algorithm requiring operating system support for run-time dynamic process creation with associated scheduling and load balancing overheads. These requirements have, in the past, discouraged the use of quicksort for parallel sorting. However, the algorithm we synthesise uses interprocessor communication to schedule and load balance itself using algorithm-specific knowledge and thus does not suffer from the overheads of run-time dynamic task creation and efficiently scales up to thousands of processors. The improved algorithm utilises communications whose destination is calculated at run-time using the values of local data. It is a member of a recently discovered class of Communication-Intensive Massively-Parallel (CIMP) algorithms [Sharp 91] which exploit the added power of communications with destinations calculated at run-time. CIMP algorithm communications are more powerful than ordinary communications because, in an n processor machine, when a CIMP communication is received by processor p rather than by one of the other $n-1$ processors, processor p gains $log_2 n$ bits of extra information about the data on the sending processor - even if only a one bit message was sent. This extra information can then be used in conjunction with new local information to calculate another new message destination, and so on. Algorithms which adopt this communication-intensive approach are likely to grow in importance as machines emerge with fast, optical communication paths. The specialisation of each processor to a particular subtask of the algorithm means that communications are not sent to arbitrary destinations determined by a task scheduler and load balancer but to destinations that are calculated using run-time data as part of the algorithm. This approach can be applied to a wide variety of problems that can be converted into message passing form, for example tessellation of the plane and dynamic programming [Sharp 90].

Our strategy is to start the sort with one item on each of a numbered set of processors and to redistribute the items amongst the processors, by repeatedly passing messages, until the items are in ascending order with respect to the enumeration of the processors. This strategy is particularly suitable for sorting multiples of 65536 items using a Thinking Machines Connection Machine [Hillis 85, Thinking 87] which has 65536 processors that can be configured as multiples of 65536 virtual processors. The algorithm synthesised here scales as $time = O(log^{2.7} N)$ sorting N items on N physical processors.

The general idea is the following. Assuming, initially, that the whole list to be sorted is known to every processor, processors compute the position of their item in the sorted list and send the item to the processor indexed by that position. For simplicity, we assume the list elements $X=[X_0, X_1, \dots X_{n-1}]$ are uniquely tagged values (e.g. the list [b, a, c] is represented by [(b, 1), (a, 2), (c, 3)]) and that the comparison operators (<, = , >, etc) automatically compare the values and not the tags. Thus the function posn, which returns the position of an element in a list can be used without ambiguity: posn(X_j, X) returns j even if the value field of other members of the list is equal to the value in X_j. Now, the initial algorithm involves every processor performing a searching operation on the whole list; not very efficient! The strategy of our transformation is to replace *computations* of item positions with messages from other processors that already have information leading to the answer. Thus, the transformation is driven by a desire to make an appropriate "answer message" be delivered to each processor. The *quid pro quo* is that a processor needs to send out information to other processors in order eventually to receive the desired answer.

The soundness of our synthesis relies on an operational semantics based on the routing of messages and their contents. The routing itself depends on the pattern matching semantics of the functional language used. Each step in the synthesis is justified by formal rules but in a more rigourous presentation these would be replaced by statements relating to the interpretation of expressions by term rewriting.

The rules allow us to perform two types of *message folding*. M-folding is the replacement of an expression with the contents of some incoming message and enables local recomputation to be replaced by message passing. R-folding (rule-folding) is the deliberate introduction of one message, which the rules state will give rise to another desired message. This enables a processor to request another to perform some operation instead of doing it locally.

In section 2 we define the structure of the messages that are passed between processors and give an initial recursive program for the quicksort algorithm which exemplifies the type of synthesis we are advocating. In section 3 we present the synthesis of our novel algorithm by repeatedly replacing units of computation by values passed in messages, as discussed above. This yields the final algorithm and a flow chart illustrating the stages in the final algorithm is given in Fig. 3 and an example of the algorithm in operation is given in section 4. We present timings of the algorithm on the Thinking Machines Connection Machine in section 5 and conclude in section 6. First we define our notation.

1.1 Notation

In order to save space while synthesising the algorithm, we transform fragments of code in isolation of the context in which they appear in the whole program. Thus some variables that appear to be free in the isolated code actually have values bound in the corresponding outer code. The reader may find it helpful to note the following conventions that apply throughout the synthesis:

- **X** is the list [X_0, ..., X_j ... ,X_{n-1}] that is to be sorted on processors numbered **a, a+1,...,b**.
- **a** / **b** are the lowest / highest numbered processor in the sort ing of the sublist currently being sorted (i.e. **a** and **b** are not global constants.)
- **n** is the total number of items being sorted by the whole algorithm (**n** is a global constant).
- **j** is used as an index.
- $\mathbf{X_j}$ is initially sent to processor **a+j** in a CS message (defined in section 2).
- **x**: Processor **a+j** refers to $\mathbf{X_j}$ locally as **x**.
- $\mathbf{X_j}$ consists of a value and a tag and the initial list **X** to be sorted is in the form [$(v_0, 0)$, $(v_1, 1)$, ... $(v_{n-1}, n-1)$], i.e. $X_j \equiv (v_j,j)$. All comparisons (e.g. $X_j < X_k$) are on the values v_j. The tags are only used when referring to the position of X_j in some list X'. Their main purpose is to distinguish items with equal values. They also allow us to insert items into a list in *tag-order* as well as in *value-order*, i.e. preserving the appropriate ordering.
- **posn**(X_j , X') is used to return the position of an item X_j in a list X' which contains items in ascending order of their tags. **Posn** searches along X' until it finds an item with the same tag as X_j and returns its position. If an item is encountered with a tag greater than X_j 's, then the search stops (because X_j is not in the list in this case). Thus **posn** returns the position where X_j would be if it were in the list.
- X' **upto** X_j returns the front of the tag-ordered list X' consisting of those elements whose tag is less than that of X_j (i.e. less than j).
- $(f \circ g)\, x = f(g(x))$.
- **Lpb**(X, pivot, 0, n-1) is used to find the lengths of the three lists which arise when elements are compared with the pivot and partitioned. The first two parameters are often understood to be present and are not written in explicitly (i.e. Lpb is assumed to have been *partially applied* to X and pivot). The last two parameters are used to select a sublist of X to be partitioned.
- **MSG(DP, contents)** is a message that transfers **contents** to the destination processor **DP**.
- **i** is always an iteration number present in the contents of a message to distinguish messages from different iterations of the algorithm.
- +++ adds triples together: (a,b,c) +++ (d,e,f) = (a+d, b+e, c+f)

2. Message structure and the initial algorithm

2.1 Message structure

All messages take the form MSG(k, e) where the constructor function MSG takes two parameters: the first (k) is the destination processor of the message and the second is the contents of the message. To begin with, we take the message contents to be one of two variants with constructors CS (*continue sort*) and ANS (*answer*). However, as our synthesis proceeds, we will introduce new kinds of message for particular purposes. The message contents therefore takes the form:

CS(x, a, b, i, X) or ANS(x).

CS takes five parameters: the first (x) is a data item X_j from some list $X=[X_0, \dots, X_{n-1}]$ which is to be sent to the processor numbered a+j in the message MSG(a+j, CS(X_j, a, b, i, X)). The second and third parameters (a and b) indicate the lowest and highest numbered processors of a contiguous range involved in the sorting of the list; the fourth parameter, i, is the iteration number of the sort and is used to distinguish this CS message from any others. In the final synthesized version of the algorithm, CS will only have four parameters, however, in order to specify the problem on the architecture initially, it is necessary to have the whole list X present as the final parameter of the the CS message. This parameter will become redundant during the transformation process and will eventually be removed.

For the ANS message constructor, its argument (x) is just the value of x in the CS message. We need iteration numbers in CS-messages because, as quicksort is recursive, care is needed to ensure that it is not possible for two identical messages to be produced. This is because the specification of the routing of messages is defined in terms of sets of messages; there is no guarantee that if two identical messages were produced, both copies would arrive at the input stream of the destination processor. In fact if identical messages were allowed, the maintenance of referential transparency would be impaired. For example suppose one of the processors carrying out the sort were much slower than the others. The other processors may race ahead to subsequent iterations of the algorithm and the slow processor would have no way of distinguishing messages concerning the iteration of the sort that it is working on from messages for the next iteration. An incorrect result could therefore be produced. (No iteration number is needed in the ANS messages because only one ANS message is sent to each processor).

As a result of sending out the CS messages to processors k=a to a+n-1, the sorting process should proceed until the position of each of the elements in the sorted list has been established. Each element is then sent in an answer message to its destination processor to produce the result. Thus, processor a+j must ensure that (eventually) the message

MSG(a + posn(X_j, sort X), ANS(X_j))

is sent to the processor numbered a + posn(X_j, sort X) in response to the incoming message MSG(a+j, CS(X_j, a, a+n-1, i, X)). Processor **a** is therefore sent the smallest item in the list, processor **a+1** the second smallest and so on up to processor **a+n-1** which is sent the largest item in the list.

Any suitable high level specification of sort can be used which satisfies the following condition which is required for a *stable* sort (i.e. one that preserves the relative ordering of equal elements).

$posn(X_j, sort [X_0, ... X_{n-1}] = \#\{X_i < X_j \ | \ 1 \le i \le n\} + \#\{X_i = X_j \ | \ 1 \le i < j\}$

Here we use quicksort as our specification:

```
sort( X ) = sort L <> E <> sort G                           sort([ ]) = [ ]
        where  (L, E, G) = part(X, pivot)                   sort([x]) = [x]
               pivot ∈ X
```

(<> denotes the append function on lists in infix form). The function part partitions the list X into those items less than, equal to and greater than the pivot and satisfies the following law:

LAW 1 $X_j <$ pivot $\Rightarrow X_j \in L$; $X_j =$ pivot $\Rightarrow X_j \in E$; $X_j >$ pivot $\Rightarrow X_j \in G$ $0 \le j \le$ length X -1
InOrder(L), InOrder(E), InOrder(G)

InOrder is a predicate that is true if the tags of the elements of its list argument are in ascending order: its presence in the second line of law 1 is required to produce a stable sort. The following recursive function satisfies law 1:

```
part(Xj::rest, pivot) <=
    if ( Xj < pivot)    then  (Xj::L, E, G)  else
    if ( Xj = pivot)    then  (L, Xj::E, G)  else
                              (L, E, Xj::G)
    where (L, E, G) = part(rest, pivot)

part( [], pivot) <= ([], [], [])
```

The function posn returns the position of an element in a list, with the first element having position zero. Posn is defined by

$posn(X_j, X) = length(X \text{ upto } X_j)$

X_c::rest upto X_j = **if** (c>j) **then** [] **else** X_c::(rest upto X_j)

and it satisfies the following law:

LAW 2 posn(x, A<>B) = **if** x ∈ A **then** posn(x, A) **else** length A + posn(x, B)

The function length returns the length of its list valued argument. Position (posn) is defined in terms of the function upto and upto compares the tags of the items rather than their values and thus can distinguish equal valued items. We formalise the relationship between the CS and ANS messages by the following rule:

RULE 1

MSG(a+j, CS(X_j, a, a-1+length(X), i, X)) $\in$ messages $0 \leq j < \text{length } X,\ i \geq 1$

$\Rightarrow$

MSG(a + posn(X_j, sort X) , ANS(X_j)) $\in$ messages

where messages denotes the set of all messages that exist in the evaluation of quicksort.

This rule, which expresses a property of the messages, operationally implies that when processor a+j receives a message

MSG(a+j , CS(X_j, a, b, i , X)) ($x=X_j$)

it is responsible for ensuring that a message

MSG(a + posn(X_j, sort X), ANS(X_j))

is produced. This is because only processor a+j is aware of the existence of messages whose destination is a+j, and thus it must make rule 1 hold.

Rule 1 has a base case, which is when only a single item is being sorted. In this case j=0, a=b and posn(X_j, sort [X_j]) = 0.

2.2 The initial algorithm

The CS and ANS messages can be introduced into the definition of a dynamic-message-passing architecture to yield an architecture-specific problem specification in the form of a top level call which sorts a list X = [X_0, X_1, ..., X_{n-1}] to give a list Y = [Y_0, Y_1, ..., Y_{n-1}] on processors numbered k=0 to n-1. Processor k=a+j receives X_j initially and receives Yj at the end of the algorithm.

```
ParSort( [$X_0$, $X_1$, ..., $X_{n-1}$] ) = [ $y_0$, $y_1$, ..., $y_{n-1}$]
where
    MSG(k, ANS( $y_k$ ) ) ∈ messages

    messages = { MSG( k, CS( $X_k$, 0, n-1, 1, X ) ) | $0 \le k \le n-1$ }   ∪
               { $P_k$ ( filter(k, messages), 1 )   | $0 \le k \le n-1$ }
    filter(k, ms) = { m∈ms | m=MSG(k,_) }

    $P_k$ ( MessagesIn, i ) =
    let { MSG(a+j, CS( x, a, b, i , X) ) } ∪ OtherMessagesIn = MessagesIn in
        if (a=b)
        then { MSG( k, ANS( x ) ) } ∪ $P_k$( OtherMessagesIn, i+1 )
        else FreeMessagesOut ∪ { MSG( destination, ANS(x) ) } ∪ $P_k$ ( OtherMessagesIn, i+1 )
    where destination = a + posn( x, sort X )
```

The processors are represented by the functions P_k ($0 \le k \le n-1$). The filter function is used to define the operation of the dynamic-message-passing architecture and uses the first parameter of the MSG messages as the destination processor number for the message. Thus, a message MSG(destination, contents) appearing on the right hand side of the equation for P_k will become a member of the set of input messages to $P_{destination}$. The last parameter (i) is equal to the number of the iteration of the current call to P_k and is incremented on each new recursive call to P_k. In the definition of P_k the variable i appears in the parameter list and on the left hand side of the let-expression. The pattern matching system must ensure that the right hand side is used only when the value of i in the message equals the value of i which is the last parameter. The initial program contains the free variable FreeMessagesOut in the message stream emerging from P_k; *any* value of FreeMessagesOut that is consistent with rule 1 provides a correct specification for quicksort; for example {}. To prove that this specification satisfies rule 1, messages can be instantiated to MSG(a+j, CS(X_j , a, a-1+length(X), 1, X)) and the program code can be unfolded until MSG(a+posn(X_j, sort X), ANS(X_j)) appears in the messages as well.

The function P_k relies on access to the whole of the list to be sorted (X) and this is initially present in the CS message sent to processor k in addition to the value of one of the items in the list. X will be removed during the transformation. Moreover, each processor individually calculates the position of its item in the final list and sends out a corresponding answer message. Clearly this is not a very efficient parallel algorithm as each processor duplicates all of the sorting work. The aim of the transformation process is to remove this redundancy, replacing it by inter-processor communication whereby useful results computed by one processor are transmitted to the processor that needs them. Thus, in our initial algorithm, processor k would like to arrange for some other processor to compute the value destination and send the result ANS(x) there. It therefore wants to pass on the value x to another processor *closer* to destination. It is by the instantiation of the free variable FreeMessagesOut that message passing is introduced. The synthesis is realised by successively taking a sub-expression from the body of P_K and replacing it with the contents of a message. The message (which was created by instantiating the free variable FreeMessagesOut in the definition of some $P_{K'}$) is extracted from MessagesIn using a let-clause.

For example, suppose the set of messages already contains the message **M1=MSG(dest1, contents1)**. To introduce some message **M2=MSG(dest2, contents2)** into the set of messages, a new rule is added which states that **M1∈ messages ⇒ M2∈ messages.** Processor **dest1** (which received **M1**) is then charged with ensuring that **M2** appears. This is achieved by instantiating $\mathbf{P_{dest1}}$'s free variable *FreeMessagesOut* to **M2** ∪ **FreeMessagesOut2**. Processor **dest2** will then receive **contents2** in a message. If **contents2** appears as a sub-expression in the body of $\mathbf{P_{dest2}}$, for example in a where clause, its calculation can be replaced by the contents of the message. The message is extracted from **MessagesIn** using a let-clause. Thus, for example, in the body of the message recipient, we can replace the expression **E(x) where x = y+z** with
let MSG(k, ValueOfxIs(x)) ∪ rest = MessagesIn in E(x)
providing we have introduced the message "ValueOfx" by instantiating *FreeMessagesOut* of some other processor. Proceeding this way the synthesis begins thus:

3 Synthesis of Parallel Quicksort

3.1 Step 1: Introducing Recursion

Unfolding the function sort in our specification changes only the last line of the where-clause:

```
destination = a+ posn( x, sort L <> E<> sort G )
where       (L, E, G) = part(X, pivot)
            pivot ∈ X
```

Interprocessor cooperation would be facilitated during the calculation of the partition if each processor were using the same pivot. The pivot must be one of the items being sorted and the quicksort algorithm operates more efficiently if the pivot chosen is one near the median of the elements. Many different algorithms could be used to choose the pivot. In order to simplify the transformation, the item on processor numbered **a** will be used as the pivot and this will be broadcast to all of the processors (including processor **a**). A better strategy using a tree of processors is described in [Sharp 90].

Thus processor **a** will send a message to each of processors **a** to **a+n-1** to inform them of the pivot. P_k must now be transformed to make use of these extra messages so that all processors use a common pivot. The new messages must satisfy the following rule:

RULE 2

MSG(a, CS(x, a, b, i, X)) $\in$ messages

$\Rightarrow$

{ MSG(k', PIVOT(x, i)) | a≤k'≤b } $\in$ messages

Thus following receipt of the message MSG(a, CS(x, a, b, i, X)), processor **a** will send out messages { MSG(k', PIVOT(x, i)) | a≤k'≤b }. To modify P_k to conform to rule 2, its free variable FreeMessagesOut can be instantiated as follows (recall that i is the iteration number):

FreeMessagesOut = (**if** k=a **then** { MSG(k', PIVOT(x, i)) | a≤k'≤b } **else** {}) $\cup$ FreeMessages2

A new free variable FreeMessages2 has been introduced to provide scope for introducing further messages later in the transformation. The new PIVOT messages are routed by the filter function and arrive in the input set of the appropriate processors. All processors can then use the value of x in the PIVOT message as the pivot. By using Rule 2 and M-folding, we can instantiate OtherMessagesIn:

OtherMessagesIn = MSG(k, PIVOT(pivot, i)) $\cup$ OthersIn2

Applying laws 1 and 2 gives a more refined version of P_k within the initial program:

```
P_k ( MessagesIn, i ) =
let MSG( a+j, CS( x, a, b, i , X) ) ∪ OtherMessagesIn = MessagesIn in
( if k=a then { MSG(k', PIVOT(x, i))   | a≤k'≤b } else {} )
∪
let MSG(k, PIVOT(pivot, i)) ∪ OthersIn2 = OtherMessagesIn in
            FreeMessages2 ∪ MSG( destination, ANS(x) ) ∪ P_k ( OthersIn2, i+1 )
where
        destination =  a +   (if x<pivot  then (posn( x, sort L)
                              else   if x=pivot  then length L + posn( x, E)
                                     else length L + length E + posn(x, sort G))
        (L, E, G) = part(X, pivot)
```

This function has reduced the sort of n items to three smaller sorts of the lists of items indexed by L, E and G. As each processor is using the same pivot, each will have calculated the same lists L, E and G. It would be better if the processors cooperated: rule 1 can be used to make this happen. R-folding the sub-expression MSG(destination, ANS(x)) using rule 1 when x < pivot, sending the message

```
        MSG( a + posn( x, L), CS(x, a, a+ length L - 1, i+1, L) )
```

will eventually cause the message

```
        MSG( a + posn( x,  sort L), ANS(x) )
```

to be sent. A similar result holds for the case x > pivot, with **a** replaced by **a + length L + length E**. Thus we obtain

```
P_k ( MessagesIn, i ) =
        let MSG( a+j, CS( x, a, b, i, X)  ) ∪ OtherMessagesIn = MessagesIn     in
        ( if k=a then { MSG(k', PIVOT(x, i))    |  a≤k'≤b } else {} )
        ∪
        let MSG(k, PIVOT(pivot, i)) ∪ OthersIn2 = OtherMessagesIn  in
        FreeMessages2
        ∪
        ( if x < pivot   then   MSG( a + PL, CS(x, a, a+ NL-1, i+1, L) )  else
          if x = pivot   then   MSG( a + NL + PE, ANS(x) )  else
          if x > pivot   then   MSG(a + NL + NE + PG, CS(x, a+NL+NE, i+1, G)
        ∪ P_k ( OtherMessagesIn, i+1 )
        where        (PL, PE, PG) = (posn( x, L), posn( x, E), posn( x, G)
                     (NL, NE, NG) = (length L, length E, length G)
                     (L, E, G) = part(X, pivot)
```

The recursive call to sort is now distributed over the processors by the **CS** messages and the calculation of PL, PE and PG has been abstracted out of nested **if** statements to make it more uniform over the processors. However, the calculation of the partition is not yet distributed.

In order to achieve cooperative calculation of the partition, we now consider the evaluation of PL, PE and PG. In the following well known auxiliary functions and laws, * denotes the post-fix form of map (defined on tuples, i.e. f* (a, b, c) = (f a, f b, f c)) and ∘ denotes function composition. Let PA2partpivot be the partial application of part to its second argument pivot and PA2uptox be the partial application of upto to its second argument x. I.e. PA2uptox = λ X. (X upto x) and PA2partpivot = λ X. part(X, pivot).

LAW 3 (Distributivity of map**)**

$(f \circ g)^* = f^* \circ g^*$

LAW 4 (Promotion of upto**)**

PA2uptox* ∘ PA2partpivot = PA2partpivot ∘ PA2uptox

Now, (PL, PE, PG) = (PA1posnx* ∘ PA2partpivot) X where PA1posnx is the partial application of posn to its first argument x. We therefore have

PA1posnx = length ∘ (PA2uptox

(abstracting the second argument from the definition of posn)

PA1posnx* = length* ∘ PA2uptox* (by law 3)

PA1posnx* ∘ PA2partpivot = length* ∘ PA2uptox* ∘ PA2partpivot

= length* ∘ PA2partpivot ∘ PA2uptox (by law 4)

Therefore, applying this expression to X, we obtain

(PL, PE, PG) = length* (part(X upto x , pivot))

But, we already have

(NL, NE, NG) = length* (part(X, pivot))

Consequently the *where abstraction*
(L, E, G) = part(X, pivot) is no longer needed and can be removed. Some way is needed to calculate length* part(X upto x, pivot) and length* part(X, pivot) in parallel. The quickest way of calculating lengths in parallel is to use a divide and conquer method that sums up items using a tree of additions.

3.2 Step 2: A divide and conquer algorithm for length*

In order to convert functions which sequentially traverse lists into divide and conquer functions, a way is needed to refer to sublists within a list. An infix function between can be used to do this. The function between : list alpha X num X num -> list alpha takes a list of items ($[x_0, .. x_{n-1}]$) and two numbers (p and q, $0 \leq p \leq q \leq n-1$) and returns the sublist $[x_p, ..., x_q]$. We make the following abbreviations (where PA1betweenX is the partial application of between to its first argument X):

Lpb = length* ∘ PA2partitionpivot ∘ PA1betweenX

(a, b, c) +++ (d, e, f) = (a+d, b+e, c+f)

Lpb can be read as "lengths of partition between". Note that **Lpb(j, j)** can easily be calculated by processor **a+j** because it knows the value of item X_j. We have (by unfolding):

Lpb(j, j) = if X_j<pivot then (1, 0, 0) else if X_j=pivot then (0, 1, 0) else (0, 0, 1)

This special case of Lpb will be very useful so we define a function triple based upon it:

triple(x, pivot) = if x<pivot then (1, 0, 0) else if x=pivot then (0, 1, 0) else (0, 0, 1)

A divide and conquer law can now be expressed in terms of Lpb:

LAW 5 (Divide and Conquer)

Lpb(j, j+1) = Lpb(j, j) +++ Lpb(j+1, j+1))

Lpb(p, r) = Lpb(p, p) +++ Lpb(p+1, q) +++ Lpb(q+1, r) $\quad$ $a < q < b$, $p>0$, $q \geq 0$, $r \geq 0$

The value of q has been left loosely defined so that different values of q can be used as necessary. Now from the definition of P_k it can be seen that each processor needs to know the value of length* part(X upto X_j, pivot) which is the same as Lpb(0, j). Unfolding this expression for list length seven (for example) using Law 5 above with
q = a + (b-a) div 2 yields the tree of computations shown in Fig. 1.

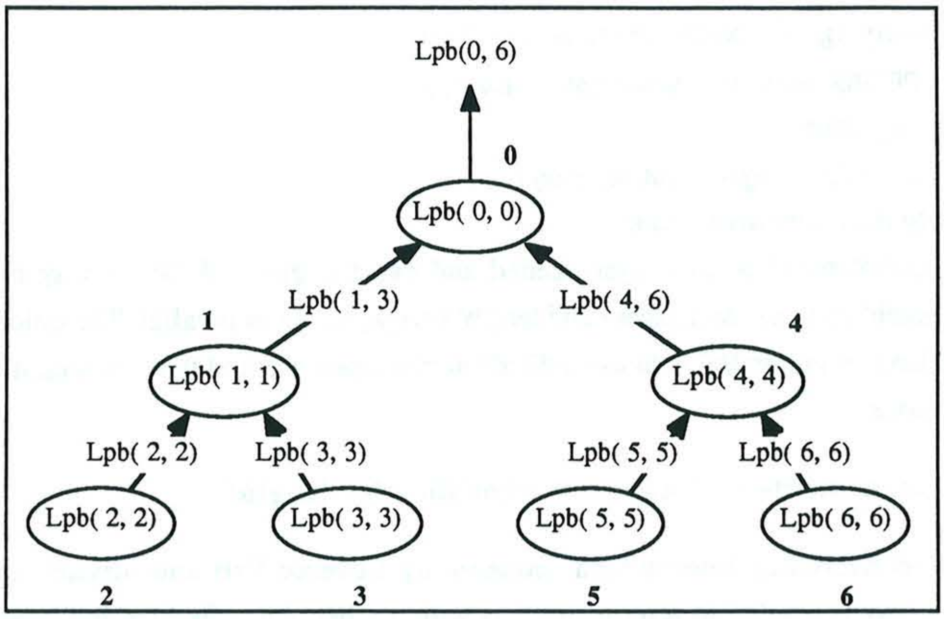


Fig. 1 Calculation of Lpb(0, 6)

In Fig. 1, at each oval, the triple inside the oval is added to the sum of the triples from the left subtree and the right subtree. Currently P_k instructs each processor to duplicate this summation.

Each oval has been labelled with a number k, (k=0 to 6). If processor k performed the additions of oval k and sent out an appropriate message to its parent (in the tree), then the result Lpb(0, 6) would emerge from processor one after a time O(log n). The result could be communicated to the rest of the processors in O(log n) time by sending it back down the tree. However, to simplify the transformation we assume the presence of a broadcast channel, and hence O(1) time.

A suitable rule to make this happen follows the definition of some auxiliary functions which we assume are given for tree navigation. The definitions refer to a depth first numbered tree containing m elements numbered a, a+1, ..., a+(m-1).

BiggestDesc(k, a, m) = Largest numbered descendant of processor k.
e.g. BiggestDesc(0, 0, 6) = 6.
If processor k is a leaf then BiggestDesc(k, a, m) = k.

Parent(k, a, m) = The parent of processor k.
(if k is a left child, the parent is k-1. If k is a right child, the parent is k-q-1 where q is the number of items in the left subtree of the parent.)

LeftChild(k, a, m) = k+1 (for a depth first tree)

RightChild(k, a, m) = BiggestDesc(k+1, a, m) + 1

IHaveARightChild(k, a, m) = True, if processor k has a right child, otherwise false.
= (BiggestDesc(k, a, m) > BiggestDesc(k+1, a, m))

RULE 3:

MSG(a, PIVOT(pivot, i)) $\in$ messages,

$\Rightarrow$

{ MSG(k', LENGTHS(Lpb(0, n-1), i)) | $a \le k' \le a+n-1$ } $\in$ messages

MSG(k, PIVOT(pivot, i)) $\in$ messages, $k \neq a$

$\Rightarrow$

MSG(parent(k, a, n), AT(Lpb(j, BiggestDesc(k, a, n) - a), k, i)) $\in$ messages

Operationally, following receipt of the message MSG(k, PIVOT(pivot, i)), processor k=a+j will:

a) If $k \neq a$, send an "Add Triples" message
MSG(parent(k, a, n), AT(Lpb(j, BiggestDesc(k, a, n) - a), k, i))
to its parent.

b) If k=a, broadcast the message
MSG(k', LENGTHS(Lpb(0, n-1), i)) to all processors k' = a to a+n-1.

The processor number k in the add triples message ensures that no two identical AT messages can be produced. The first AT messages come from the leaf processors. The parents of the leaf processors receive these messages and use the information in them to calculate further AT messages for their parents. Finally the root processor sends out the LENGTHS messages. Thus we instantiate FreeMessages2:

```
FreeMessages2 =
(if leaf( k, a, n )
then MSG( parent( k, a, n ), AT( triple( x ), k, i) )
else   if k≠a
       then MSG( parent( k, a, n ), AT( Lpb( k-a, BiggestDesc( k, a, n ) - a ), k, i ) )
       else { MSG( k', LENGTHS(Lpb( 0, n-1 ), i) )  |    $0 \le k' \le n-1$ }     )
∪ FreeMessages3
```

Law 5 can be used in the calculation of Lpb(k-a, BiggestDesc(k, a, n) - a): applying the second case of law five for processors with two children, taking q = q' - a, gives:

```
Lpb( k-a, BiggestDesc(k, a, n) - a ) =
       Lpb(k-a, k-a) +++ Lpb( (k+1) - a , q' - a ) +++ Lpb( q'+1 - a, BiggestDesc(k, a, n) - a )
       where k < q' < BiggestDesc(k, a, n)
```

The Lpb function has been highlighted because it calculates values that can be deduced from incoming messages. From rule 3 we can deduce that processor k receives AT messages containing

- Lpb(k'- a , BiggestDesc(k', a, n) - a)   from its left child
- Lpb(k"- a, BiggestDesc(k", a, n) - a)   from its right child

The letter k has been dashed because relative to the parent (processor k), k'=k+1 and k"= right child of processor k = BiggestDesc(k', a, n) + 1.

Choosing q'= BiggestDesc(k', a, n) and observing that

BiggestDesc(k, a, n) = BiggestDesc(k", a, n)

enables the AT messages to be used instead of calculating Lpb((k+1) - a , q' - a) and Lpb(q'+1 - a, BiggestDesc(k, a, n) - a) locally. Thus we can write equivalently (j=k-a and x is the first value received in the CS message) :

```
FreeMessages2 =
(if leaf( k, a, n )
then let OthersIn3 == OthersIn2 in MSG( parent( k, a, n ), AT( triple(x, pivot), k, i) )
else    if not IHaveARightChild(k, a, n)
        then    let MSG(k, AT( T1, _, i )) ∪ OthersIn3 == OthersIn2 in
                if k≠a  then  MSG( parent(k, a, n), AT( T1 +++ triple(x, pivot), k, i) )
                        else  { MSG( k', LENGTHS(T1 +++triple(x, pivot) , i) )  |     0 ≤ k' ≤ n-1 }
        else    let MSG(k, AT( T1, LeftChild(k, a, n) , i )) ∪ MSG(k, AT(T2, RightChild(k, a, n) , i) )
                                                                 ∪ OthersIn3 == OthersIn2 in
                if k≠a  then  MSG( parent(k, a, n), AT( T1 +++ triple(x , pivot) +++ T2, k, i) )
                        else  {  MSG( k', LENGTHS(T1 +++ triple(x , pivot) +++ T2, i) )  | 0 ≤ k' ≤ n-1}
)
∪ FreeMessages3
```

(Notice that for direct implementation we would require pattern matching against non-constructors, e.g applications of LeftChild.)

Rule 3b now enables OthersIn3 to be instantiated to

MSG(k, LENGTHS((NL, NE, NG), i)) ∪ OthersIn4

and the contents of the LENGTHS message can be used instead of calculating length* part(X, pivot) locally. Thus only (PL, PE, PG) = length* (part(X upto x , pivot)) remains in the final where-abstraction and the next step of the transformation is to eliminate it by introducing further messages.

3.3 Step 3: Determining Destination Processors

Once again the divide and conquer laws can be applied and a depth first tree connection of the processors is suitable. However, a more efficient algorithm is produced if different values of m are used for left children and right children. Further specialisations of law 5 are obtained by taking q = k–a and q = parent(k, a, n) - a respectively:

LAW 5a (Divide and Conquer Law for Left Children):

Lpb(a, k-a) = Lpb(a, k-a-1, pivot) +++ Lpb(k - a, k-a)

LAW 5b (Divide and Conquer Law for Right Children):

Lpb(a, k-a) = Lpb(a, parent(k, a, n) - a)
+++ Lpb(parent(k, a, n) + 1 - a, k-a-1)
+++ Lpb(k - a, k-a)

Applying these laws leads to the network of computations in Fig. 2 which illustrates the case when a=0 and n=6. To visualize law 5b, consider processor 4: it is trying to calculate Lpb(0, 4). It is a right child so it uses the right child divide and conquer law (5b) to get

Lpb(0, 4) = Lpb(0, 0) +++ Lpb(1, 3) +++ Lpb(4, 4)

Now processor 0 can easily calculate Lpb(0, 0) as it has item 0 on it. Processor 0 also knows the value of Lpb(1, 3) because this was sent to it by its left child (processor 1) in an AT message in Fig. 1. Thus if a rule is introduced to make processor 0 produce a "Determine Destination" message MSG(5, DD(Lpb(0, 0) +++ Lpb(1, 3), i)) then processor 4 will not have to recompute these values locally but can use the information in the message.

Now consider processor 5. It uses the law of divide and conquer for left children (law 5a) to determine that Lpb(0, 5) = Lpb(0, 4) +++ Lpb(5, 5). It will not have to recalculate Lpb(0, 4) locally if processor five sends it a message MSG(5, DD(Lpb(0, 4), i)). Note that the DD messages consist of three integers and are thus quick to send. The following rule will thus dramatically reduce the amount of redundant computation:

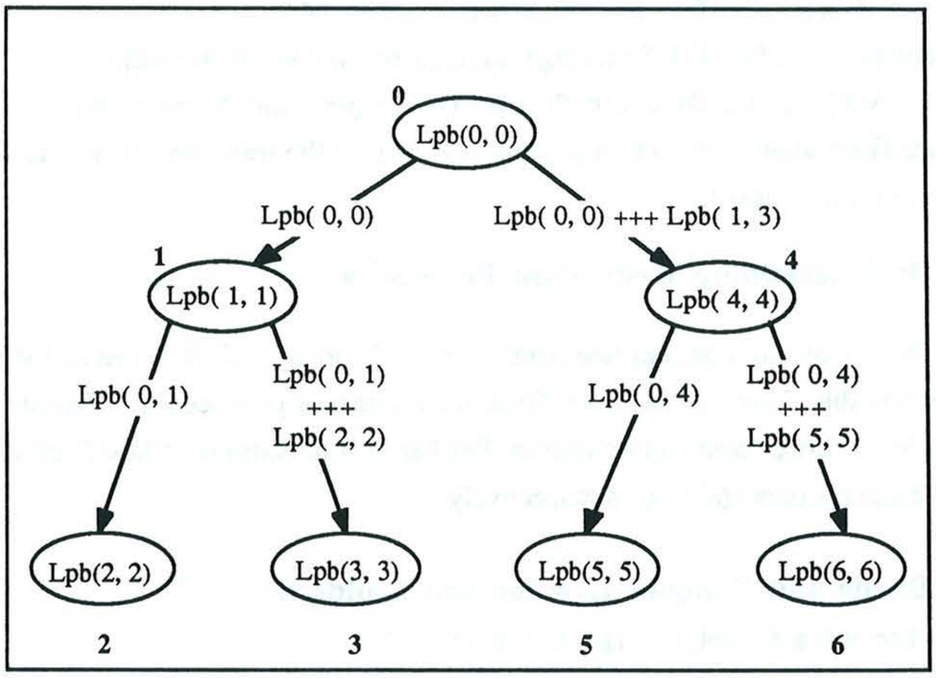


Fig. 2 Determination of Destination of Continue Sort Message

RULE 4:

```
{  MSG( parent(k, a, n), AT( T1, k, i) )          | a+1≤k'+1≤a+n-1 } ∈ messages,
{  MSG( k', LENGTHS(  Lpb( 0, n-1 ) ,  i)  )       | a≤k'≤a+n-1 }  ∈ messages
⇒
{  MSG( k'+1, DD( Lpb( 0, k'-a ), i ) )                       | a≤k'+1≤a+n-1 }  ∈ messages      ,
{  MSG( RightChild(k', a, n),  DD( Lpb( 0, k'-a ) +++ T1, i ) )  |  a≤k'+1≤a+n-1 }  ∈ messages
```

Thus following receipt of the LENGTHS message, processor k sends messages

- MSG(k+1, DD(Lpb(0, k-a), i)) to its left child, if it has one
- MSG(RightChild(k, a, n), DD(Lpb(0, k-a) +++ T1, i)) to its right child, if it has one

T1 is the triple that it received from its left child in an AT message. The contents of these messages are used by the children in their subsequent calculations. Rule 4 can now be used to instantiate FreeMessages3:

```
FreeMessages3 =
(     if leaf(k, a, n)   then {}
                         else    MSG( k+1, DD( Lpb( 0, k-a ), i ) ) ∪
                                 if not IHaveARightChild(k, a, n) then {}
                                 else MSG( RightChild(k, a, n), DD( Lpb( 0, k-a ) +++ T1, i ) ) )
) ∪ FreeMessages4
```

For the root processor k=a, so the determine destination message contains Lpb(0, 0) which is the same as triple(X_0). For the other processors (except the leaves) the calculation of Lpb(0, k-a) can utilize the DD(Lpb(0, (k-1) - a), i) message sent to it from its parent (using laws 5a and 5b and rule 4) because of the following relationship:

```
Lpb( 0, k-a )      =  Lpb( 0, (k-1) - a) +++ Lpb(k-a, k-a)
                   =  (N1, N2, N3) +++ Lpb(k-a, k-a)
                      where   MSG(k, DD(N1, N2, N3), i) ∪ Others5   = OthersIn4
```

Thus FreeMessages3 can handle the root separately from the other processors:

```
FreeMessages3 =
if (k = a )
then    let OthersIn5 == OthersIn4
        in MSG( k+1, DD( triple(x), i ) ) ∪
                if not IHaveARightChild(k, a, n)  then {}
                else MSG( RightChild(k, a, n), DD( triple(x , pivot) +++ T1, i ) )
else     let MSG( k, DD( (N1, N2, N3), i ) )  ∪ OthersIn5 ==  OthersIn4
        in      if leaf(k, a, n) then {}
                else        if not IHaveARightChild(k, a, n) then {}
                            else MSG( RightChild(k, a, n), DD( triple(x, pivot ) +++ T1, i )
∪ FreeMessages4
```

There is no longer any reference to X because it has been m-folded out by replacing Lpb expressions with the contents of an incoming DD message. Thus a distributed parallel algorithm has been synthesized. The remaining free messages (FreeMessages4) can be instantiated to {} and the CS message no longer needs its fifth parameter , X. We therefore have the final algorithm:

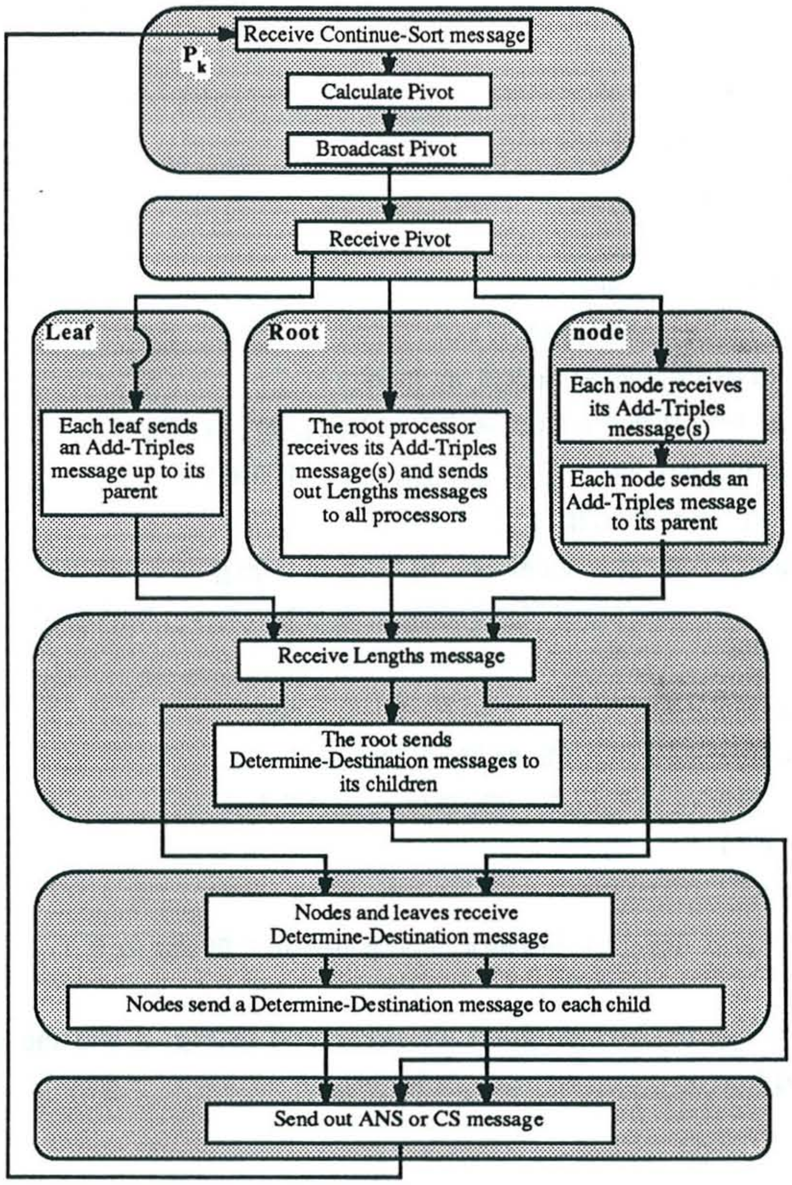
```
ParSort( [$x_0$, $x_1$, ..., $x_{n-1}$]  ) = [ $y_0$, $y_1$, ..., $y_{n-1}$]
where
        MSG(k, ANS( $y_k$ ) ) ∈ messages
        messages = {  MSG( k, CS( $x_k$, 0, n-1, 1, X ) )  |   0≤k≤ n-1 }     ∪
                   {  $P_k$ ( filter(k, messages), 1   )    |   0≤k≤ n-1 }
        filter(k, ms) = { m∈ms | m=MSG(k,_) }

$P_k$ ( MessagesIn, i ) =
let MSG( a+j, CS( x, a, b, i)  ) ∪ OtherMessagesIn = MessagesIn     in
( if k=a then { MSG(k', PIVOT(x, i))    |  a≤k'≤b } else {} )
∪
let MSG(k, PIVOT(pivot, i)) ∪ OthersIn2 = OtherMessagesIn  in
! FreeMessages2 !
(if leaf( k, a, n )
then  let  OthersIn3 == OthersIn2 in MSG( parent( k, a, n ), AT( triple( x, pivot  ), k, i) )
else    if not IHaveARightChild(k, a, n)
        then      let  MSG(k, AT( T1, _, i  )) ∪ OthersIn3 == OthersIn2  in
                  if k≠ a   then   MSG( parent(k, a, n), AT( T1 +++ triple(x, pivot ), k, i) )
                          else    { MSG( k', LENGTHS(T1 +++ triple(x, pivot ), i) )  |      0 ≤ k' ≤ n-1 }
        else
```

```
                let MSG(k, AT( T1, LeftChild(k, a, n) , i )) ∪ MSG(k, AT(T2, RightChild(k, a, n) , i) )
                                                                    ∪ OthersIn3 == OthersIn2  in
                if k≠ a   then  MSG( parent(k, a, n), AT( T1 +++ triple(x , pivot ) +++ T2, k, i) )
                          else  { MSG( k', LENGTHS(T1 +++ triple(x , pivot ) +++ T2, i) )  | 0≤ k'≤ n-1 } )
∪   ! FreeMessages3  !
if k = a
then    let OthersIn5 == OthersIn4
        in  MSG( k+1, DD( triple(x, pivot), i ) ) ∪
                if not IHaveARightChild(k, a, n)  then {}
                else MSG( RightChild(k, a, n), DD( triple(x, pivot ) +++ T1, i ) )
else     let MSG( k, DD( (N1, N2, N3), i ) )  ∪ OthersIn5 ==  OthersIn4
        in      if leaf(k, a, n) then {}
                else        if not IHaveARightChild(k, a, n) then {}
                            else MSG( RightChild(k, a, n), DD( triple(x , pivot ) +++ T1, i )
∪
( if x < pivot   then   MSG( a + PL, CS(x, a, a+ NL-1, i+1) )  else
  if x = pivot   then   MSG( a + NL + PE, ANS(x) )  else
                        MSG(a + NL + NE + PG, CS(x, a+NL+NE, i+1)
 ∪ P_k ( OthersIn5, i+1 )
```

triple(x, pivot) = if x< pivot then (1, 0, 0) else if x= pivot then (0, 1, 0) else (0, 0, 1)

The stages in the synthesized program are illustrated in Fig. 3 and the operation of the synthesized algorithm on a worked example is given in section 4.



Receive Continue-Sort message
P_k
Calculate Pivot
Broadcast Pivot
Receive Pivot
Leaf
Root
node
Each leaf sends an Add-Triples message up to its parent
The root processor receives its Add-Triples message(s) and sends out Lengths messages to all processors
Each node receives its Add-Triples message(s)
Each node sends an Add-Triples message to its parent
Receive Lengths message
The root sends Determine-Destination messages to its children
Nodes and leaves receive Determine-Destination message
Nodes send a Determine-Destination message to each child
Send out ANS or CS message

Fig. 3 Parallel Quicksort

4. The Synthesized Parallel Quicksort Algorithm

We now summarise the operation of the synthesised quicksort algorithm. Initially the items are distributed one per processor as shown in Fig. 4. Suppose item d on processor 6 is chosen as the pivot. The pivot is broadcast to all the processors using a binary tree connection in O(log n) time and each processor compares its item with the pivot in O(1) time. The number of items less than, equal to and greater than the pivot is added up in O(log n) time using the same tree connection of the processors as before.

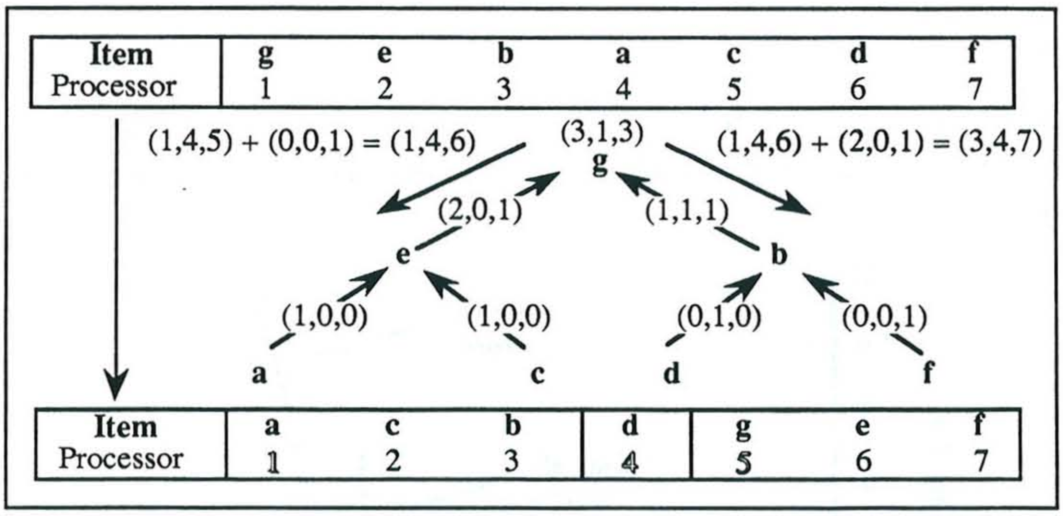


Fig. 4 Massively Parallel Quicksort

The number of items in each group is broadcast to all the processors using the tree connections in the opposite direction in O(log n) time. The processors are split so that lower numbered processors (i.e. processors 1 to 3) sort items less than the pivot and higher numbered processors (i.e. processors 5 to 7) sort items greater than the pivot. However, the items need to be moved to new processors. For example, processor 1 (which will sort items less than the pivot) contains g which is greater than the pivot. The g needs to be sent to one of processors 5, 6 or 7. Ordinarily a load balancing and scheduling system would be used to map the new sorting processes to processors, however, these systems tend to dominate the performance of the algorithm. Our algorithm self-schedules in O(log n) time: processor 1 gets first choice and chooses the lowest numbered available processor in the appropriate group. Thus processor 1 will send the g to processor 5. Processor 1 can now calculates that processors (1,4,5)+(0,0,1) = (1, 4, 6) are available to processor two for sorting items less than, equal to, and greater than the pivot respectively. Now processor one knows the number of items in each group that processor two and its descendents have because this is the information it received during the tree addition in Fig. 4. As processor two and its descendents will be using the lower numbered processors up first, processor one can calculate the lowest numbered processors available for processor three and its descendents. Thus processor one sends to processor three a message (2, 0, 1) + (1, 4, 6) = (3, 4, 7) which takes into consideration that the left subtree of processor 1 contains two items less than the pivot and one item greater than the pivot. Processors two and three can then inform their descendants which processors are still available to be used. Each processor then knows where to send its item and, after the items have been redistributed the same algorithm can be used again on the subgroups until the list is sorted. The whole algorithm thus has an expected execution time of O(log n) iterations each taking O(log n) time, i.e.O((log n) * (log n)).

5. Execution Time On Connection Machine

The quicksort has been executed on 8192 processors of a Thinking Machines Connection Machine CM2 and gives excellent results in comparison with enumeration sort and bubble sort (Fig. 5).

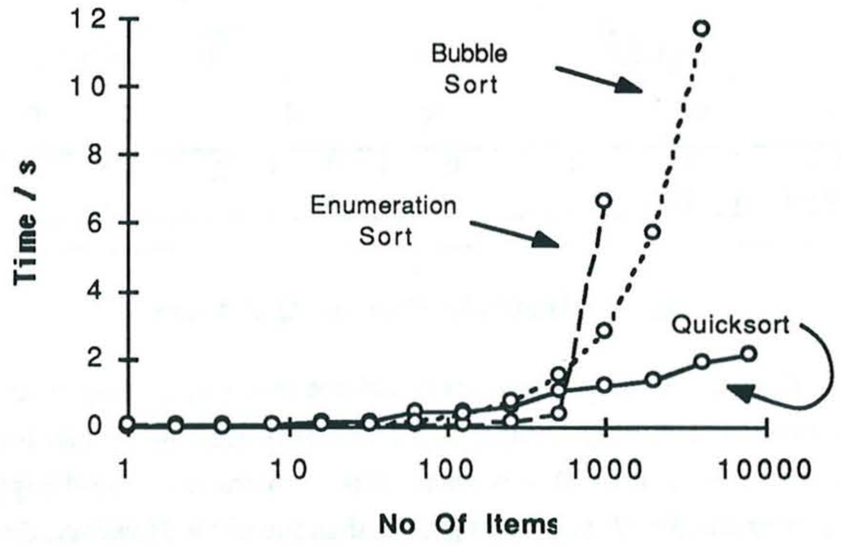


Fig. 5 Quicksort , Bubble Sort and Enumeration Sort on The Connection Machine. ***(No of Items = No of Processors Used)***

The theoretical complexity of our parallel quicksort is **time = k (log n)^p** where p is 2. This is because we expect **O(log n)** recursions of each partitioning about the pivot and the communications to choose a pivot and then redistribute the items take **O(log n)** time. In practise we would expect there to be some constant set-up time **c**; **time = k (log n)^p + c**, however, on the Connection Machine c=0.0006 seconds and is thus insignificant compared with **k (log n)^p** which is three orders of magnitude bigger than **c**. We therefore ignore **c** in the following analysis.

Taking logs we have **log time = log k + p log(log n)**. Thus plotting **log time** against **log log n** should yield a straight line of slope **p**. Fig. 6 shows that, for the Connection Machine, **p=2.7**. This is good agreement with the theoretical prediction: the hypercube hardware and routing software of the Connection Machine has added an extra **O(log n)$^{0.7}$** factor onto the execution time of the **O(log n)^2** algorithm.

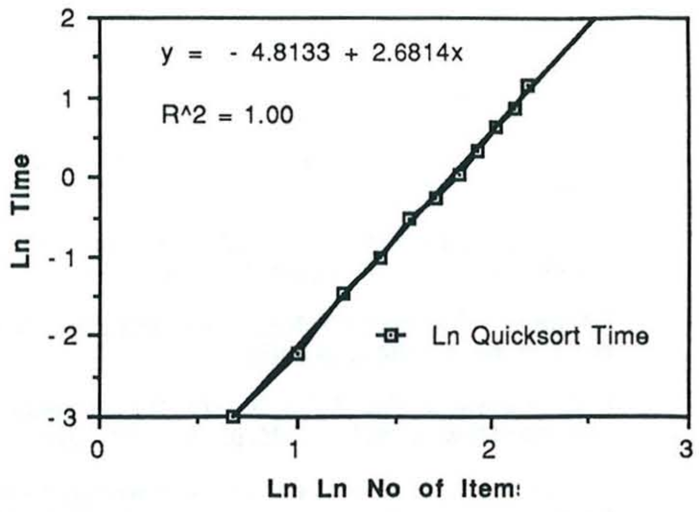


Fig. 6 The Slope of This Graph is p in O(log n)^p

6. Conclusions

We have applied a methodology for systematically synthesising algorithms for message passing parallel machines to the formal derivation of a novel parallel sorting algorithm based on Quicksort. We have demonstrated that communication can be used beneficially and that efficient algorithms can be synthesized from high level specifications. The decomposition of the algorithm into processor-specific tasks has played an integral part in the creation of the novel algorithm with its run-time data dependent interprocessor communication scheme.

Despite the success of other parallel sorting algorithms [Akl 85, Bitton 84, Gibbons 88, Hirschberg 78, Lakshmivarahan 84, Preparata 78], previous attempts to parallelize quicksort [Moller-Nilesen 63, Evans 85, Deminet 82] have not achieved a speedup greater than six [Francis 88]. This is because they did not use a communication-intensive strategy and thus were crippled by the overheads of fine grain dynamic task allocation and load balancing.

A O(log n) time quicksort could theoretically be achieved [Martel 89] but it requires a nonexistent concurrent-write machine to execute it. Thus the O($\log^2$ n) quicksort synthesized here, and related algorithms [Sharp 89] have the lowest complexity and fastest execution speed for parallel quicksort on commercially available parallel hardware. The quicksort program is effectively finding the fixpoint of a list of elements when a partial sorting function is repeatedly applied. Fixpoint methods arise in many branches of science and engineering and so the communication-intensive massively-parallel approach is a promising technique for solving large problems of this type.

7. References

[Akl 85] S. G. Akl, "Parallel Sorting Algorithms," Academic Press, 1985.

[Bitton 84] D. Bitton, D. Dewitt, D. Hsiao, J. Menon, "A Taxonomy of Parallel Sorting Algorithms," Computing Surveys, Vol. 16, No. 3, 1984, p.287.

[Clark 80] K. L. Clark, J. Darlington, "Algorithm Classification Through Synthesis," The Computer Journal, Vol. 23, No. 1., 1980, pp.61-65.

[Deminet 82] J. Deminet, "Experience With Multiprocessor Algorithms," IEEE Trans. Comput., vol C-31, pp. 278-288, Apr. 1982

[Evans 85] D. J. Evans and Y. Yousif, "Analysis Of The Performance Of The Parallel Quicksort Method," BIT, vol. 25, pp. 106-112, 1985

[Francis 88] R. S. Francis, I. D. Mathieson, "A Benchmark Parallel Sort For Shared Memory Multiprocessors," IEEE Trans. Comput., Vol. 37, No. 12, Dec. 1988, pp.1619-1626.

[Gibbons 88] A. Gibbons, W. Rytter, "Efficient Parallel Algorithms," Cambridge University Press, 1988.

[Hillis 85] W. D. Hillis, "The Connection Machine," MIT Press 1985.

[Hirschberg 78] D. S. Hirschberg, "Fast Parallel Sorting Algorithms," Commun. ACM Vol. 21, No. 8, Aug. 1978, pp.657-666.

[Hoare 62] C. A. R. Hoare, "Quicksort," The Computer Journal vol. 5, pp.10-15, 1962.

[Knuth 73] D.E. Knuth, "The Art Of Computer Programming Volume 3 : Sorting and Searching," Reading, MA : Addison-Wesley, 1973.

[Lakshmivarahan 84] S. Lakshmivarahan, S. K. Dhall, L. L. Miller, "Parallel Sorting Algorithms," Advances in Computers, Vol. 13. New York: Academic, 1984, pp.295-354.

[Martel 89] C. U. Martel, D. Gusfield, "A Fast Parallel Quicksort Algorithm," Information Processing Letters, Jan. 1989, pp97-102.

[Moller-Nilesen 87] P. Moller-Nilesen and J. Staunstrup, "Problem-heap : A Paradigm For Multiprocessor Algorithms," Parallel Computing 4 (1987) , pp 63-74.

[Preparata 78] F. P. Preparata, "New Parallel Sorting Schemes," IEEE Trans. Comput., Vol. C-27, No. 7, July 1978, pp.669-673.

[Sedgewick 78] R. Sedgewick, "Implementing Quicksort Programs," Comm. ACM, vol. 21, pp.847-856, Oct. 78.

[Sharp 89] D. W. N. Sharp, M. D. Cripps, "A Parallel Implementation Strategy For Quicksort," Proc. 1989 International Symposium on Computer Architecture and Digital Signal Processing, Vol. 1, Hong Kong, Oct. 89, pp.305-309.

[Sharp 90] D.W.N. Sharp, "Functional Language Program Transformation For Parallel Computer Architectures," Ph.D. thesis, Dept. of Computing, Imperial College, London Univ., 1990.

[Sharp 91] D.W.N. Sharp, M.D. Cripps, "Parallel Algorithms That Solve Problems by Communication," Internal Report DoC 91/23, Imperial College, May '91. (Also to appear in Proc. Third IEEE Symposium on Parallel and Distributed Processing, Dallas, Texas, Dec 91.)

[Thinking 87] Thinking Machines, "Connection Machine Model CM-2 Technical Summary," Thinking Machines Corporation Technical Report Series, HA87-4, 1987.

DISCUSSION

Rapporteur: Daniel McCue

Professor Randell observed that one technique for generating one-liners in APL, using the iota operator, was considered bad form because it generated huge amounts of temporary data. These constructs expose much parallelism but at a great cost in space. He went on to ask, "How does your transformation system compare to optimized APL with respect to space utilisation?"

Dr. Harrison acknowledged that it is true that the price of referential transparency in functional programming is that lots of data is generated and lots of copying occurs. He explained that this is generally true in all functional programming languages - not particular to parallel systems. Initial implementations of the program transformation system will not be concerned with optimising space although clearly more work is needed here. Initial approaches to addressing this problem in functional languages relied on garbage collection (in sequential implementations). More sophisticated techniques are used now in conjunction with GC.

Professor A. J. Cole remarked that In the example given [a graphics application averaging intensity over a set of neighboring points], Dr. Harrison had ignored the boundary points. "Isn't that a serious oversimplification?", he asked.

Dr. Harrison agreed that the example was somewhat simplified for presentation purposes, but argued that as in any language, boundary conditions can be addressed at a cost in program clarity. He claimed that functional programming provides a significant improvement in clarity over other programming styles not only in the main, simpler parts of the application, but in addressing boundary problems as well. For example, higher-order functions could be used to address the boundary problem here.

Professor B. Randell asked, "Have you tried this technique on an algorithm that would interest a real user?"

Dr. Harrison replied that he has applied it to ray tracing - the most complicated problem he have attempted.

Professor Dr. D. Swierstra quipped, "This approach to program construction carries with it a proof of correctness - a very great benefit - but this necessarily excludes "interesting" problems!"

Professor W. D. Shepherd asked, "Have you thought of applying this technique to Programmable Gate Arrays?"

Dr. Harrison replied that PGAs look interesting, but have not been attempted yet.

Professor Drs. C. Bron asked, "What happens if the transformation maps onto a nesting of skeletons? One cannot nest the hardware!"

Dr. Harrison responded by saying that there is no problem with nesting. If all of the skeletons that result from the transformation can be efficiently mapped onto the target hardware, there is no problem. Otherwise, some inefficiencies may result.