Reducing Graph Copying Costs

Time to Wrap it Up

Marco Kesseler
Faculty of Mathematics and Computer Science,
University of Nijmegen,
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands

Abstract: Graph copying is used in parallel implementations of functional languages on architectures with distributed memory. This paper will explore the costs of graph copying in detail. It will make clear that these costs can form a bottleneck for a class of serious parallel programs. This not only comprises practical divide and conquer style programs - of which parallel matrix multiplication is an example -, but also programs that use pipelines, such as the sieve of Erathostenes. One can observe that copying costs vary widely for different data structures. We will show how arrays can be used to reduce copying costs considerably. This resulted in significant speed-ups for the examples above.

Keywords: parallel functional programming, distributed memory, graph copying, graph conversions, delays

1. Introduction

Graph rewriting systems have been studied as a computational model for functional languages and their implementation. The lazy functional language Concurrent Clean, for instance, is based on such a graph rewriting system and the Clean compiler is known for the efficient code it produces [1, 2, 3, 4]. Graph rewriting systems represent a program as a set of rewrite rules. Starting with an initial graph, rewriting (reduction) takes place according to these rules. At any moment in time there will be multiple redexes, and therefore a reduction strategy is needed. Sequential strategies will at any point in time reduce a single redex, as opposed to parallel ones that will reduce multiple redexes at a time.

Parallel implementations of functional languages can roughly be divided into two classes: those aimed at shared memory architectures, and those suited for architectures with distributed memory. The main difference between the two is that shared memory implementations will have multiple processors working on a globally accessible shared graph, whereas distributed implementations have to distribute the graph so that processors can reduce a private copy. Important issues for shared memory implementations are the use of caches and locks. Conversely, implementations for distributed memory have to deal with the costs of graph copying. These areas of research are related in the sense that they focus on providing the processors efficient access to graphs. On the other hand, the notion of copying sharply contrasts with the notion of sharing.

This paper will focus on the costs of copying of graphs on general purpose distributed memory machines. More precisely, we will not concern ourselves with the pure transmission costs, but with costs that surround the act of transportation. In general, efficient reduction requires a graph representation that differs from the one needed for efficient communication. During reduction a node typically contains local memory addresses to efficiently access other nodes on the same processor, whereas a graph that is being transported will contain some sort of globally usable addresses, such as offsets within a message. This introduces the need for graph conversions during copying. We will not assume any special conversion hardware, so the processor itself has to perform this task. Additionally, administrative tasks - possibly employing control messages - are needed to regulate copying. Administration and conversion introduce delays that are unacceptable for some parallel programs.

These protocol costs have been largely ignored by the implementors of parallel functional languages. In the first place, of all implementations, there are not many aimed at distributed memory architectures. Most recent research in parallel functional programming focuses at implementations for (virtual) shared memory architectures (<υ,G>, AMPGR, GAML, GRIP, Flagship, and HyperM) [5, 6, 7, 8, 9, 10, 11, 12, 13]. And secondly, the exact costs of copying have not been made explicit for distributed memory implementations. Sometimes the use of an interpreter causes uncertainty about realistic copying costs (PAM, π-RED$^+$) [14, 15]. In other cases only some simple divide-and-conquer programs have been tested that are not very conclusive with respect to overall communication overheads (HDG, SkelML) [16,17]. In particular pipelines of processes are not considered at all.

This paper will explore the costs of graph copying in detail. It will make clear that these costs can form a bottleneck for a class of serious parallel programs. We will show how one can avoid problems by using appropriate data structures. These can either be used implicitly, via skeletons, or explicitly. Considerable speed-ups can be obtained this way.

Our work has some relation with that presented in [18], where a set of skeletons on special data structures has been proposed to exploit data-parallelism. This is based on the assumption that some data structures are cheaper than others for certain problems. The effect of different data structures on efficiency of communication is not examined in detail. In contrast, we will not limit ourselves to data parallelism in any way, nor will we present a set of skeletons. We will not examine algorithmic properties of data structures, such as the ability to efficiently access elements in a random way (as in [18]). Instead, we will focus on efficiency of communication alone and explain why certain data structures are cheaper than others in this respect.

We have structured this paper as follows. The first section will point out two directions of interest for reducing copying costs. The following sections will show for each direction that serious programs exist that suffer from high copying costs and how this problem can be solved.

2. Copying Costs

Reports on speed-ups for parallel implementations of a functional languages on machines with distributed memory are rare and strikingly similar. Either abstract machine code has been interpreted, resulting in performance figures that are not conclusive with respect to the possible performance of compiled code, or one does compile, but then only a small set of programs turns out to perform well. These programs all have a comparable structure that makes them less sensitive to delays in communication.

A closer look at this sort of program reveals a number of similarities. In the first place, they all exploit a simple divide-and-conquer mechanism. Computation unfolds into a tree of processes. These trees are not very deep, but they contain a huge amount of processes. This means that data does not have to travel far. At the same time there will be many processes per processor, so that a waiting process does not imply the processor becomes idle. In addition, the complexity of copying data is very small, compared to the complexity of the computation. As a result, communication overheads can easily be ‘suppressed’ by increasing the problem size. Altogether, this means that communication latencies do not (need to) have much effect on efficiency. The table below lists some typical examples.

table 1. Execution times of Divide-and-Conquer Programs in Concurrent Clean on a network of transputers. nfib computes nfib 30. queens computes the number of solutions of the queens problem on a board of 10 by 10. mandelbrot computes and draws a well known fractal in a resolution of 560 by 320 pixels and depth 128. raytrace computes and draws a scene of polygons in a resolution of 300 by 300 pixels. The mandelbrot program and the raytrace program consist of ‘sequential’ functions that describe the colour of each pixel on the screen. These have been passed - using currying - to a parallel pixel-plotting function. The raytrace program is the most complex of the ones listed. It - rather implicitly - copies a database of objects to all processors.

	1 T800	16 T800’s	Speedup
nfib	12.2 sec.	1.2 sec.	10.2
queens	44.2 sec.	5.4 sec.	8.2
mandelbrot	153.5 sec.	13.6 sec.	11.3
raytrace	331.3 sec.	33.0 sec.	10.0

These results have been obtained with an implementation of Concurrent Clean on a network of transputers. If we relate it to other transputer implementations, it certainly outruns interpreters like PAM and π-RED$^+$ and as far as we have been able to determine it is faster than SkelML (raytrace) and the HDG machine (nfib and queens). This has been achieved by combing a state-of-the-art compiler, a specialised parallel runtime system and thresholds to control the grain size of computation. No load balancing technique has been used. Processes have been scattered randomly over the network, which results in a somewhat balanced load if the number of processes is high. Be this as it may, load imbalance is the main reason that processor utilisation is not optimal. Compared to interpretation the problem size had to be increased to achieve similar speed-ups. This comes to no surprise as interpreters are between one and two orders of a magnitude slower.

Let us assume we do not have such a divide and conquer program. Data will be complex, or it will have to travel over a long distance, or both. We will see realistic examples below. The price of copying cannot be ignored now, it has to be kept as small as possible. There are three ways to do this. First of all, one should decrease absolute costs of conversion and communication. Secondly, communication and computation should overlap as much as possible, hiding delays. And finally, copying tasks should be distributed over the network as much as possible. The latter can usually be realised by selecting a suitable algorithm for splitting up the problem. Except for some short contemplations we will not consider this. Instead, we will show how major improvements of the first and second kind can be realised. The next section will be concerned with reduction of the overall conversion costs, the following will focus on overlapping computation and communication.

3. Reducing Conversion Costs

Conversion costs are most significant if data is complex. Large data structures can impose serious overheads when copied to another processor. An example matrix multiplication program will clarify this. Comparing different data structures below, we will see that it is important to choose data structures carefully, because the conversion costs may vary tremendously for different ones. This section will end with a short reflection on distributing copying tasks over the network.

3.1. Matrix Multiplication with Lists

Matrix multiplication is more complicated than the simple divide and conquer programs of the previous section, because the arguments and results are more complex. Increasing the problem size will not automatically result in better performance as the copying costs will grow as well. Depending on the size of the network the complexity of communication may approach the complexity of plain matrix multiplication: $O(n^3)$. This happens for example when every new matrix element is computed by a separate processor: each processor needs $O(n)$ data and $O(n^2)$ processors are needed. It is crucial that copying costs are kept to a minimum here.

A simple way to multiply two square matrices A and B of size n (number of rows) on n processors, is to split A into n rows $a_1 ... a_n$ and compute $a_i \cdot B$ on processor p_i. We will assume that A and B both reside at some processor p_s at the start and that the result should return to p_s. Suppose further, that p_s itself distributes all work. It is easy to see that this will not lead to speed-ups if communication is expensive: copying B (n-1) times is $O(n^3)$. In general, if we use this method on p processors, the costs of communication are $O(p \cdot n^2)$. We can see below what results were obtained with matrices represented by lists of lists of floating point numbers. We should note here that the use of lists does not impose overheads to access matrix elements: the algorithm does not need random access to elements.

Table 2: multiplication of matrices represented by lists of lists of floating point numbers.

	1 T800	16 T800's	Speedup
MatMult 64 x 64	3.9 sec.	5.0 sec.	0.8
MatMult 128 x 128	30.3 sec.	35.0 sec.	0.9

Table 3: time spent by root processor in parallel version on conversion and garbage collection

	Conversions	Collections
MatMult 64 x 64	2.9 sec.	0.2 sec.
MatMult 128 x 128	16.9 sec.	10.0 sec.

These figures need some explanation. First of all, the conversion costs per matrix element are higher than an elementary multiplication step. From the timings for one processor one can deduct that it takes about 15 μs to perform a basic multiplication step (total time is size^3 times basic time). If we assume that it takes p seconds to pack an element and u seconds to unpack it, we can derive the following formula for conversion costs at the root. The root (un)packs data for 15 remote processors and n denotes the size of the square matrices.

conversion at the root:

$$15p\,(n^2 + n^2/16) + 15u(n^2/16) \approx (16p + u)n^2 \qquad (1)$$

In our implementation unpacking is less expensive than packing, as unpacking does not involve copying nodes. It merely has to adjust pointers. We have not yet established the exact difference in speed. Conversion takes between $16pn^2$ and $17pn^2$. This means that packing costs are approximately between 42 and 45 µs for the first test and between 61 and 65 µs for the last. The variation is caused by extra overheads if memory runs low during conversion, which is the case during multiplication of matrices of 128 by 128 reals.

Secondly, the last test suffers from excessive garbage collection times at the root processor. The list representation of each matrix consumes about 400 KByte of memory, while graph conversion claims a similar amount for the resulting message. Only few room remains to perform graph reduction. With a total heap size of 3 MByte per transputer, garbage collection times add up to 10 seconds. The root processor spends 27 seconds on conversion and garbage collection alone, while matrix multiplication takes less than 2 seconds per processor. This is partly due to the use of a two-space copying collector, which traverses all live data and limits the maximum usable space to half the heap size. For this particular problem it might be advisable to reduce the number of garbage collections by using an heuristic that will hold garbage collections as long as the heap contains large - or many - messages. Garbage collection would not free enough memory in this case, as opposed to communication. Apart from fewer and cheaper garbage collections, the transmission of messages benefits from the temporary absence of garbage collection. On the other hand, this introduces alternating phases of communication and computation, which may not always be advantageous. We have not yet tested this.

And finally, idle time is considerable. A large amount of data has to be transported over the transputer communication links. In addition, conversions and garbage collections cause delays as well. Not only do these operations take notable time to complete, they also - unlike normal reduction - make overlapping computation and communication more difficult. One cannot boldly access messages in the heap during these operations because nodes may be in an inconsistent state (perhaps containing forwarding addresses and marks). In the current implementation processors cannot forward messages while they are busy converting graphs and collecting garbage. The matrix multiplication program suffers greatly from this problem.

From the above it becomes clear that copying is excessively expensive here. In short, converting a large graph takes much time and memory. One can observe that relative conversion costs can be reduced by increasing the problem size, but at a much slower rate than for the simple divide-and-conquer programs listed earlier. At the same time physical memory size restrains the use of extremely large matrices. These introduce substantial memory management overheads, which in turn will cause higher communication delays.

3.2. Matrix Multiplication with Arrays

We will show below that the use of strict arrays solves many problems. In this particular case we do not need them for efficient random access to elements, but because they can be copied cheaply. It is important to note that arrays cannot be implemented efficiently in a trivial way for functional languages. Referential transparency has to be guaranteed. Clean achieves this with its uniqueness type system, that derives sharing information for objects. Destructive updates on non-shared objects do not harm referential transparency [1,19]. It allows an array to be represented by a single, efficiently modifiable, block of memory. Strict arrays consist of evaluated elements that are placed one after another in memory. As a result, copying a strict array merely involves moving a block of memory if the elements are flat (e.g. a strict array of integers). No expensive conversion is involved.

A few cases need to be considered while copying a graph that contains strict arrays. Clearly, if the graph is a strict array with flat elements, it can be copied without conversion by means of a simple block move. In many cases however, the graph will consist of a mixture of strict arrays and ordinary nodes. If an array is small, it will not be worthwhile to send a separate message for it. In this case the graph copier will pack it - i.e. copy it - in a message together with the other nodes of the graph. On the other hand, if an array is large, the copier will create a separate message for it. This message consists of the array itself, creation does not involve copying. For our transputer system 16 Kbyte turned out to be a good boundary between small and large arrays. The combination of two-dimensional strict arrays and this copying scheme has lead to the following execution times for matrix multiplication. The algorithm is exactly the same as the one used above.

Table 4: multiplication of matrices represented by 2-dimensional arrays of floating point numbers.

	1 T800	16 T800's	Speedup
MatMult 32 x 32	1.0 sec.	0.4 sec.	2.5
MatMult 64 x 64	7.6 sec.	1.0 sec.	7.6
MatMult 128 x 128	60.3 sec.	5.0 sec.	12.1
MatMult 256 x 256	480.7 sec.	32.5 sec.	14.8

Table 5: time spent by root processor in parallel version on conversion and garbage collection

	Conversions	Collections
MatMult 256 x 256	0.03 sec.	0.08 sec.

These speed-ups are comparable to the ones obtained for languages like C or FORTRAN. The only drawback of using arrays is that sequential matrix multiplication is slower for arrays than for lists. This shows that computing the address of an element in a two-dimensional array, including boundary checks, is more expensive than walking through a list. At this moment special functions are used to perform these computations. An improvement can be expected if the compiler generates code to optimise array addressing. Nevertheless, if we compare these results with the previous ones, the merits of using strict arrays are clear. Larger matrices can be used and for the large ones the speed-up is nearly perfect. If we take a closer look, we can see that all problems listed above are avoided by using this different data representation.

- There are nearly no conversion costs, as matrices do not get packed at all. They already are packed from the start.

- Virtually no time is spent on garbage collections. On the one hand less garbage collections are needed: no memory is allocated for conversions and the matrix representation is more compact. A matrix of 128 by 128 reals only takes about 128 Kbyte as opposed to 400 Kbyte above. On the other hand each garbage collection is less expensive, as the amount of live data is smaller, while having a less complex structure as well.

- Idle time is small. Less data has to be transported, due to the compact matrix representation. The absence of conversions and garbage collections is important as well. Unlike these computations, normal reduction can easily be overlapped by communication, because the transputer communication hardware works independently of the CPU.

One could argue that lists are more elegant, because of their flexibility. On top of the efficient parallel matrix multiplication function for two-dimensional arrays, one can define one for a list representation. Conversions between the two representations are of $O(n^2)$. If one has n processors parallel matrix multiplication is of $O(n^2)$ as well. One can take C^{-1} (Multiply (C A) (C B)), where C is the function that converts the list representation into the array representation. Having n processors one does not convert B n times, but only once. In effect the conversion has been moved in front of all copying operations so that these can share the converted result. Another viable way to achieve this might be the use of a special cache for converted graphs so that these do not need to be recomputed. Some primitive for broadcast communication could be helpful as well, not so much to reduce transportation costs in this particular case, but rather to allow conversion costs to be reduced.

3.3. Matrix Multiplication with Distributed Copying

We have seen that absolute conversion costs can be greatly reduced by choosing appropriate data structures. Alternatively, one could reduce the time needed for conversions by distributing copying operations over all processors. This is mostly worthwhile if copying costs are high, as is the case for the matrix multiplication program above that used lists. It required the root processor to perform most conversions, which turned out to be very expensive. Conversely, one could use an algorithm that avoids this problem. A well-known way to achieve this is to split matrices recursively in half until some basic size has been reached. Matrix multiplication will then be performed on these submatrices. Combination of the partial results yields the final one. This gave the following result for matrices represented by lists.

Table 6: matrix multiplication by recursive split-up

	1 T800	16 T800's	Speedup
MatMult 64 x 64	3.9 sec.	2.6 sec.	1.5
MatMult 128 x 128	30.3 sec.	16.0 sec.	1.9

Table 7: time spent by root processor in parallel version on conversion and garbage collection

	Conversions	Collections
MatMult 64 x 64	0.6 sec.	0.0 sec.
MatMult 128 x 128	4.4 sec.	2.5 sec.

These results are better than the naive matrix multiplication algorithm using lists, but they are far from optimal. Although the root spends 4 to 5 times less time on conversions and garbage collections, the total execution time has only halved. As conversions are not only executed at the root now, but for a considerable part on other processors as well, we need to take into account these costs. Here problems arise because in Clean graphs are copied as a whole and not on a per-node basis. The distribution of data takes several phases that do not overlap. In other words, the copying costs - mostly conversion costs - add up over a path from the root to a leaf. This explains the speed-up of a mere factor of 2. Details can be found in the frame below.

To overcome this problem, one has to avoid the standard copying of lists. It is here that the use of streams comes to mind. One would like to break up the copying of a list in pieces, so that a rather continuous stream of data can be formed in which a pipeline of copying functions works concurrently. This will not reduce the overall copying costs, but ideally it will fold them up so that delays are reduced (Cannon's algorithm for matrix multiplication would be interesting in this respect). The next section will show how such pipelines can be efficiently implemented. For technical reasons we have not (yet) implemented matrix multiplication based on this method. Note also that the conversion costs for the splitting algorithm are still too high to improve the version that uses arrays. We will focus on another example: the sieve of Erathostenes.

4. Overlapping Communication and Computation

To some extend communication in Clean is lazy. By default results are not transmitted until needed. This has some advantages. It avoids needless communications, suppresses small (expensive) messages, and allows 'remote' results to be referred without copying. The main disadvantage is that it increases delays in obtaining results: if a result is

Delays in Matrix Multiplication

For the naive matrix multiplication solution the total amount of delays was largely determined the copying costs at the root. If we ignore the effects of garbage collection and define p and u to comprise the conversion and communication costs per element, we can use the following formula for delays.

delay: $15p(n^2 + n^2/16) + u(n^2 + n^2/16) + pn^2/16 + 15un^2/16 = (16p + 2u)n^2$ (2)

The copying costs at the root are given by $15p(n^2 + n^2/16) + 15un^2/16 \approx (16p + u)n^2$ (see also (1)), which is almost the same. For the matrix splitting solution this situation differs. The conversion costs at the root are as follows:

conversion at the root: $2p(n^2 + n^2/2) + un^2 = (3p + u)n^2$ (3)

It indicates that packing costs per element are now roughly between 37 and 49 μs and between 67 and 90 μs respectively. This is in accordance with the figures found earlier if we consider that memory management overheads are worst for the largest matrices. The formula for delays is very different however. If we recursively split the matrices four times, we obtain 16 submatrices. The delays for splitting and recombining can then be approximated by the following formulae.

packing delays during splitting: $(3n^2 + 3n^2/2 + 3n^2/4 + 3n^2/8)p$ (4)

unpacking delays during splitting: $(3n^2/2 + 3n^2/4 + 3n^2/8 + 3n^2/16)u$ (5)

packing delays during recombination: $(n^2/2 + n^2/4 + n^2/8 + n^2/16)p$ (6)

unpacking delays during recombination: $(n^2 + n^2/2 + n^2/4 + n^2/8)u$ (7)

(4)+(5)+(6)+(7): $(6.5625p + 2.8125u)n^2$ (8)

Depending on the relative costs of unpacking these delays differ with a factor 1.9 to 2.4. This largely explains the speed-ups that were found.

needed, one has to wait until it has been returned. First a request message will be transmitted. After this has arrived, the result will be packed as soon as it has been computed, and finally the obtained message can be returned and unpacked. Only then the requesting process can proceed. This may have great effect on performance for certain programs.

In addition to the programs above, which suffered to a large extent from the complexity of conversion alone, there is a class of programs that is seriously affected by these delays. In general this is a problem if data is not necessarily complex, but has to travel over a large distance. Many (small) delays will then add up to a considerable amount. In functional languages this travelling distance is closely related to the depth of parallel function composition, whereas the costs of each communication step are related to the structural complexity of arguments and results. Although reduction of conversion costs can be important for reducing delays as well, we will not consider this here. The previous section already has focused on this. In contrast, this section will concern itself with reducing delays by overlapping computation and (more eager) communication.

Streams consist of lengthy compositions of functions. These pipelines can be used to solve real-world problems, but they are very sensitive to delays. A classical example - although not very useful - is the sieve of Erathostenes. It generates a list of prime numbers by filtering all non-primes from the list of natural numbers. The filter consists of a pipeline of small filters. Each assumes the first number in the stream to be a prime number. It will first deliver this prime and next remove all numbers from the stream that are divisible by it. The resulting stream is passed to the next filter.

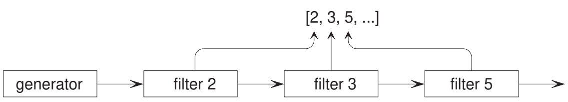

Figure 1: snapshot of the process structure of the sieve of Erathostenes

The definitions for the most important functions are given below. We will start with a short explanation of the Concurrent Clean syntax. For clarity, notations are in bold face. The annotations {I}, {P} and {P AT proc} are all process annotations. They start up a new process on the annotated function, which will reduce it to root normal form. The {I} annotation starts up a new process on the current processor, whereas the {P} annotation starts up a new process on some other processor. The {P AT proc} annotation creates a new process on the given processor. If a processor has more than one process, these processes run interleaved. A list with head h and tail t is denoted by [h | t], whereas [] is the empty list. Pattern matching is used in textual order on each alternative (an alternative starts with the function name). In addition, guarded expressions are possible within each alternative. They are separated by arrows and Boolean expressions (starting with IF). The first Boolean expression that evaluates to TRUE indicates the expression chosen.

```
Sieve n [p | s]
   -> [p | {P} Sieve ns {I} f], IF (= n 0)
   -> [p | {I} Sieve (-- n) f],
      f:   Filter s p,
      ns:  SquareRoot p;
Sieve n [] -> [];

Filter [x | xs] p
   -> Filter xs p,          IF (= (% x p) 0)
   -> [x | {I} Filter xs p];
Filter [] p -> [];
```

New processes are either started at a remote processor with a {P} annotation, or on the current processor with a {I} annotation. A heuristic has been used to balance the load. The higher the prime number of a filter, the less numbers it has to filter. An increasing number of filter processes will be grouped on the same processor. Processes reduce functions to root normal form. Extra internal processes are needed to drive the filtering (in a speculative way). In addition one needs to annotate a filter function with an {I} annotation to keep it on the current processor when a remote sieve is started. The results of this program have been listed in the table below.

Table 8: The sieve of Erathostenes.

	1 T800	32 T800's	Speedup
Sieve 10000	18.4 sec.	8.9 sec.	2.0
Sieve 20000	61.7 sec.	23.7 sec.	2.6
Sieve 40000	213.3 sec.	64.7 sec.	3.3
Sieve 80000	752.2 sec.	?	?

The speed-ups are not very impressive, mainly due to the effects of delay. In addition, the use of the {I} annotation in the filter function gives rise to unbridled eager computation of results. This may flood the heap, which introduces substantial memory management overheads and prevents the use of large - or infinite - lists. The exact cause of delays varies widely, because there is no mechanism to control the length of the list that will be copied. If lists are long, conversion time introduces considerable delay. If lists are short, many small messages are needed and communication overheads are relatively high.

To overcome these problems an efficient buffering mechanism between processors is useful. Firstly, it is able to limit the number of elements that can be computed in advance so that memory problems are avoided. Secondly it can minimise delays by controlling the size of messages and the use of early requests, meaning that request messages are sent before the requested object is actually needed. This will cause conversions and communication to take place at less critical moments. Reducers do not have to stop (that long) performing useful computations, so that communication and computation will overlap. By controlling the message size one avoids messages that are too large or too small to be efficient.

Strict arrays are very useful for implementing a buffering skeleton. As has already been shown above, strict arrays can be communicated very efficiently, so buffers implemented this way can be as well. In addition, the type system can ensure that buffers are not shared, so that they can be destructively updated. This is very important for efficiently filling and emptying a buffer. We have implemented a low level buffering function for lists of integers that has been based on this method. The definitions below are a little more complicated in reality. For clarity, we have left out boundary conditions and the definitions of low level functions.

```
:: Buffer PROCID INT [INT] -> [INT];
   Buffer proc size list
     -> Read ({P AT proc} Write size list);

:: Write INT [INT]   -> BufArray;
   Write size list
     -> Fill (CreateBuffer size) size list;

   Fill buf s list -> Pack result next,
    next: Fill (CreateBuffer s) s rest,
      (result, rest): FillBuffer buf s list;

:: Read BufArray -> [INT];
     Read buffer    -> [ n | Get new next],
    (n, new): GetElement result,
    (result, next): Unpack buffer;

   Get curr next
     -> [n| Get new next], IF NotEmpty curr
    -> Read next,
        (n, new): GetElement curr;
```

Each Buffer function starts a Write function at another processor and passes the result to a local Read function. The Write function will fill a newly created buffer with the values it obtains by evaluating 'size' elements of its parameter list. It finally packs the resulting buffer and a reference to the next Fill function for the rest of the list. Evaluation of the Read function will force this packet to return. The Unpack function extracts the returned buffer and the reference to the next one. Both packing and unpacking are very efficient due to the use of arrays. The unpacking function will additionally send a request for the next buffer without suspending itself. This forces speculative evaluation of the next 'size' elements. The GetElement Function will take the first element out of the buffer. Hereafter the situation will be as follows.

Any consumer will be able to get the next element from the buffer quickly by evaluating the Get function. At the same time the Fill function will be filling the next buffer. On average 'size' elements will be buffered. If processes proceed at equal speed the next buffer will be filled - and possibly returned - by the time the consumer needs it. Note that evaluation of the list will stop if the consumer does not evaluate the Read function for some buffer: the next buffer will not be requested then.

If we use this new buffer function for the implementation of the sieve program, we get the following code. It does not need extra process annotations to drive the computation, but it does need a strictness annotation (!), to locate the filter function correctly: each remotely started parallel sieve will start a filter function at the current processor. Note that the CurrentP function is not really referentially transparent: it has a different result if evaluated on different processors. On the other hand it will not have any influence on the final outcome of the program, except for the placement of processes and as a result of this, on efficiency. We are working on a stream implementation that does not need the CurrentP function.

```
Sieve n [p | s]
 -> [p | {P} Sieve ns stream], IF (= n 0)
 -> [p | {I} Sieve (-- n) f],
     stream:   Buffer !CurrentP 40 f,
     f:      Filter s p,
     ns:     SquareRoot p;
Sieve n [] -> [];

Filter [x | xs] p
 -> Filter xs p,     IF (= (% x p) 0)
 -> [x | Filter xs p];
Filter [] p -> [];
```

The execution times for this program have been listed below. Clearly the speed-ups for the sieve are significant if enough numbers are filtered. We should note that sieves are not necessarily running on neighbouring processors and that the buffering function is not yet optimal. In addition, idle time is approximately 50% on each processor. This means that even better performance can be expected in the future, although these results already are a notable improvement compared to results presented earlier for functional programs.

Table 8: The sieve of Erathostenes with buffering

	Sequential	16 Transputers	
	Seconds	Seconds	Speedup
Sieve 10000	18.4	3.9	4.7
Sieve 20000	61.7	9.8	6.3
Sieve 40000	213.3	29.8	7.2
Sieve 80000	752.2	90.9	8.3
Sieve 100000	1136.9	131.4	8.7
Sieve 200000	4225.8	476.0	8.9

	Sequential	32 Transputers	
	Seconds	Seconds	Speedup
Sieve 10000	18.4	3.5	5.3
Sieve 20000	61.7	8.3	7.4
Sieve 40000	213.3	21.8	9.8
Sieve 80000	752.2	56.4	13.3
Sieve 100000	1136.9	81.1	14.0
Sieve 200000	4225.8	269.6	15.7

5. Conclusions

We have concentrated on the costs of graph copying. These costs turned out to form a bottleneck for a class of useful parallel programs, such as matrix multiplication and programs that use pipelines. Observing that copying costs vary considerably for different data structures we have been able present a solution that relies on the use of arrays. These can be transferred very efficiently, and they form the basis of efficient skeletons for communication. This has lead to significant speed-ups for the example programs.

6. References

[1] Plasmeijer M.J., Eekelen M.C.J.D. van (1993). *Functional Programming and Parallel Graph Rewriting.* Addison Wesley, 1993.

[2] Nöcker E.G.J.M.H., Smetsers J.E.W., Eekelen M.C.J.D. van, Plasmeijer M.J., (1991). 'Concurrent Clean', In *proceedings of Parallel Architectures and Languages Europe (PARLE'91)*. Springer LNCS 505, Vol. II, page 202-219.

[3] Eekelen M.C.J.D. van, Plasmeijer M.J., Smetsers J.E.W., (1990). 'Parallel Graph Rewriting on Loosely Coupled Machine Architectures' In *proceedings of the workshop on CTRS'90*. Montreal Canada.

[4] Smetsers J.E.W., Nöcker E.G.J.M.H., Van Groningen J.C., Plasmeijer M.J. (1991). Generating Efficient Code for Lazy Functional Languages. In *proceedings of the Fifth International Conference on Functional Programming Languages and Computer Architecture (FPCA'91),* Cambridge, U.S.A., Springer LNCS 523, page 592-617.

[5] L. Augustsson, T. Johnsson (1989). Parallel Graph Reduction with the <*υ,G*> - machine. In *proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA '89),* page 202-213, London, U.K. ACM 1989.

[6] L. George (1989). An Abstract Machine for Parallel Graph Reduction. In *proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA '89),* page 214-229, London, U.K. ACM 1989.

[7] L. Maranget (1991). GAML: a Parallel Implementation of Lazy ML. In *proceedings of the Fifth International Conference on Functional Programming Languages and Computer Architecture (FPCA'91),* Cambridge, U.S.A., Springer LNCS 523, page 102-123.

[8] S. L. Peyton Jones, C. Clack, J. Salkild, M. Hardie (1987). GRIP - a High Performance Architecture for Parallel Graph Reduction. In *Proceedings of the Third International Conference on Functional Programming Languages and Computer Architecture (FPCA '87).* Portland, U.S.A., Springer LNCS 274, page 98-112.

[9] S. L. Peyton Jones, C. Clack, J. Salkild (1989). High Performance Parallel Graph Reduction. In *Proceedings of Parallel Architecture and Languages Europe (PARLE '89),* Eindhoven, the Netherlands, Springer LNCS 365/366, page 193-206.

[10] P. Watson and I. Watson (1986). Graph Reduction in a Parallel Virtual Memory Environment. In *Graph Reduction,* Santa Fé, New Mexico. Springer LNCS 279, page 265-214.

[11] P. Watson and I. Watson (1987). Evaluation of Functional Programs on the Flagship Machine. In *proceedings of the third International Conference on Functional Programming Languages and Computer Architecture (FPCA '87).* Portland, U.S.A., Springer LNCS 274, page 80-97.

[12] I. Watson, V. Woods, P. Watson, R. Banach, M. Greenberg, J. Sargeant (1988). Flagship: A Parallel Architecture for Declarative Programming, In *15th IEEE/ACM synp. Computer Architecture.* Honolulu, Hawaii. SIGARCH newsletter, 16(2), 1988, pages 124-130.

[13] H.P. Barendregt, M. Beemster, P. H. Hartel, L. O. Hertzberger, R. F. H. Hofman, K.G. Langendoen, L. L. Li, R. Milikowski, J. C. Mulder, W.G. Vree (1992). *Programming Clustered Reduction Machines.* Technical report CS-92-05, Dept. of. Comp. Sys, Univ. of Amsterdam, 1992.

[14] R. Loogen, H. Kuchen, K. Indermark, W. Damm (1989). Distributed Implementation of Programmed Graph Reduction. In *proceedings Parallel Architectures and Languages Europe (PARLE '89),* Eindhoven, the Netherlands, Springer LNCS 365/366, pages 136-157.

[15] T. Bülck, A. Held, W. Kluge, S. Pantke, C. Rathsack, S. Scholz, R Schröder (1993). Preliminary Experience with a π-RED$^+$ Implementation on an nCUBE/2 System. In *proceedings of the fifth International Workshop on Implementation of Functional Languages,* Nijmegen, the Netherlands. Technical report 93-21, Faculty of Mathematics and Computer Science, Univ. of Nijmegen, 1993.

[16] H. Kingdon, D. R. Lester, G.L. Burn (1991). The HDG-machine: a Highly Distributed Graph-Reducer for a Transputer Network. *The Computer Journal,* 34(4): 290-301, 1991.

[17] T. A. Bratvold (1993). A Skeleton-Based Parallelising Compiler for ML. In *proceedings of the fifth International Workshop on Implementation of Functional Languages,* Nijmegen, the Netherlands. Technical report 93-21, Faculty of Mathematics and Computer Science, Univ. of Nijmegen, 1993.

[18] H. Kuchen, R. Plasmeijer, H. Stoltze (1993). Efficient Distributed Memory Implementation of Functional Languages. In *proceedings of the fifth International Workshop on Implementation of Functional Languages,* Nijmegen, the Netherlands. Technical report 93-21, Faculty of Mathematics and Computer Science, Univ. of Nijmegen, 1993.

[19] S. Smetsers, E. Barendsen, M. van Eekelen, R. Plasmeijer (1993). Guaranteeing Safe Destructive Updates through a Type System with Uniqueness Information for Graphs. In *Graph Transformations in Computer Science.* Dagstuhl Castle, Germany, Springer LNCS 776, pages 358-379.