Parallel Implementation of Functional Languages*

R. Wilhelm, M. Alt, F. Martin, M. Raber **

Fachbereich 14 Informatik
Universität des Saarlandes
Saarbrücken

Lazy evaluation is optimal (up to overhead). J. Niehren, POPL'96

Abstract. Great hopes in the exploitation of the (implicit) parallelism inherent in functional programs have driven a number of projects. General frustration resulted wherever implementations on distributed memory machines were attempted. The grain size of potentially parallel tasks is too small to amortize the enormous costs of the necessary communication. The management of the parallelism is expensive. Decision making in the network operating system suffers from the fact, that most parameters such as the number of potentially parallel tasks and their execution times are dynamic.
This article tries to summarise the state of the art in the parallel implementation of functional languages, to give reasons why attempts have failed to show performance, and argues why this will probably remain so. Similar arguments will hold for other declarative languages and mechanisms with implicit fine grain parallelism such as logic and constraint languages.

1 Introduction

Parallel architectures are a big promise for a further increase in computation power of computer systems. However, the exploitation of the parallelism offered by them is still a difficult task. The step from a sequential to a parallel program incurs the following overheads for managing the parallelism.

- Processes have to be managed; they have to be created, blocked, resumed, and terminated.
- Global scheduling of the processes usually requires global information. Gathering this information needs expensive communication.
- In the case of distributed memory machines, memory management including garbage collection may be quite costly.

* Research supported by the Deutsche Forschungsgemeinschaft, SFB 124 - VLSI-Design Methods and Parallelism

** Author's current address: Metalevel Software, Saarbrücken

– If the data communicated between processors are a graph-like linked structures, each communication may be preceded and followed by an expensive conversion, because the local representation of the graphs may not be invariant under the transport to a different processor.

In an implementation on a distributed memory machine, where communication is expensive, the overhead in communication hurts the most. Among the communication overheads listed above there are *problem inherent* overheads. These consist in the distribution of problem data and/or code over the network. Then there are the *organizational* overheads, e.g. managing distributed memory structures, gathering load information for scheduling. These overheads have to be compensated for by an exploitation of the increased computational power. Successes have only been achieved on restricted classes of problems [FWM94].

Embarrassingly parallel problems: These are problems, which admit solutions with very large grain and little communication, e.g. factorization of large numbers.

Data parallel problems: These are problems, whose solutions consist of regular computations on large sets of homogeneous data.

Large reactive systems: In some applications a large homogeneous set of processes is dynamically created on response to stimuli from the environment.

Distributed data bases: The data are spread over a set of local memories with appropriate replication and migration mechanisms to allow for parallel transactions while preserving data consistency.

Problems in NC: These are problems, which allow efficient solutions on the PRAM, the complexity peoples' favourite machine model. The algorithms typically explicitly allocate work to processors.

What are the expectations of a programmer, who writes a parallel program X?

1. X will run *not much slower* than the best corresponding sequential version.
2. X will run *faster* than the best corresponding sequential version.
3. X will run *faster with a predictable speedup* than the best corresponding sequential version.
4. X will run *faster with a reasonably sized constant fraction speedup* than the best corresponding sequential version.

Here "best sequential implementation" refers to the state-of-the-art and not to proved lower bounds. Thus, the parallel implementor aims at a moving target. Predicting the overhead involved in managing parallelism and thus predicting speedup is difficult and only possible for a small set of algorithms so far.

Usually, X will be some application, such as predicting the weather, simulating the air stream around a car, or solving some other large system of differential equations. We would then interpret the above statements as holding for all inputs to X. Note, that the program may access and use the information about the size of the problem!

In the case of the parallel implementation of a language, the challenge is considerably higher. One expects, that these statements are interpreted as holding for at least a reasonably large, not contrived set of programs and all inputs to these programs.

We should not be content with a language or language class, which doesn't even let us exploit the parallelism admitted by the problem. It would be very surprising if any implementation of such a language would exhibit any speedups given the rather restricted success in the exploitation of parallelism. In other words, if skilled algorithms people and programmers were unable to use parallelism to speed up execution of their programs, how could we expect that implementations of a rather general class of languages could do better? This would be even more true for languages, which don't support constructs responsible for the success in the above listed problem classes, e.g. large grain parallelism, higher order operators on aggregates of homogeneous data, explicit allocation of work to groups of processors.

Hence, we arrive at the following maxim for the designer and the implementor of a language.

The language should allow the programmer to formulate a solution to his problem, such that the maximal parallelism available in the solution is expressible in the program, and the implementation should exploit this parallelism to gain a constant fraction speedup.

Functional languages, says the folklore, offer a huge potential for the parallel execution of programs, because these are referentially transparent and therefore inherently parallel. Similar arguments are made in favour of attempts to execute logic languages [AB90], constraint languages [AMT93], and termrewriting systems on parallel architectures [AK96]. For reasons of programming methodology lazy functional languages have been advocated as preferable to strict languages. They offer the greater challenge for the parallel implementation, since the lazy reduction strategy is inherently sequential, and thus parallelism has first to be discovered by extremely expensive and complex analyses [Mau86,Bur90b]. Still many groups, including ours, chose a lazy source language in their attempt to develop efficient parallel implementations [RRH+88] [Bur90a] [NSMC92] [Hil95] [Tre94] [Cha94][THJ+96][NSEP91], although it now looks like a masochist approach. We take the purist's position in this paper. Source language is a purely functional language. Most of the treatment does apply to eager and lazy languages. We don't consider (course grain) functional languages, where programs would have a static number of (rather large) tasks communicating through specified channels.

Besides a few implementations of data parallel languages [FMS+95] none of the attempts have shown impressive results. This article tries to show where progress has been made, where no breakthrough has been achieved, and why this probably will not happen.

We will highlight during our discussion open questions for which no efficient solution is known so far by:

This is an open problem. ▌

2 Target architectures and fair measurements

As target architectures we consider distributed memory machines. Until PRAM-architectures become commercially available [ACC+90,AKP91], these are the favourite target architectures for reasons of scalability and availability in a rather large variety (at least until most makers went bankrupt). Virtual shared memory machines often are not scalable and expose a nonuniform memory access behaviour, which makes it hard to predict runtimes.

Processing power is still increasing faster than the communication speed in the networks. Currently sending one message through the network costs approximately as much as executing 1000 floating point instructions. This makes up the quotient t_{comm}/t_{comp}, mentioned below. Hence, a processor can compute quite a bit, until it pays to send some work it has to another processor. In addition it means, that all the algorithms, which are needed to manage parallelism, i.e. process management, scheduling, and garbage collection heavily increase the organizational overhead, when they need communication.

2.1 Grain size

[FWM94], p. 51, shows a formula for the inter-node communication part of the parallel computation overhead. This part of the overhead is said to be proportional to t_{comm}/t_{comp}, i.e. the relation between the time needed to send a word to a neighbour and the time to execute one instruction. It is claimed to also be proportional to the reciprocal of some root of the *grain size*, the unit of work executed in parallel, and to some factor derived from the difficulty to embed the problem dimension into the topology dimension. Think of which of the different parts can be influenced by an implementation. The problem dimension, i.e. the connectivity of the graph can be influenced by the decision to share or to copy common computations. The main influence on this overhead, however, will come from the grain size. Since communication overhead is the "killer" for many attempts to parallel execution, it pays to increase the grain size. However, this should be done in a honest way, see below.

In our problem domain, a grain usually is the evaluation of one closure or one function application. The size of these grains is statically not known, in general. Non-recursive and *small* expressions [Rab94] are the exceptions. However, given the usual style of functional programming they tend to have rather small evaluation times. Most of the time, it doesn't pay to send them away for a remote evaluation. We are then left with recursive expressions whose dynamic sizes, i.e. evaluation times, are unknown.

2.2 Sources of false happiness

There are several ways an enthusiastic implementor of a functional (or other) language on a DMM can cheat himself.

The first and most commonly found way is to run the parallel implementation on one processor as a comparison with the p-processor version. This usually gives nice speedups mostly due to effects discussed in the next paragraph.

The attribution of certain effects to a parallel implementation are debatable. The fact that a p-processor machine will have roughly p times the memory of a single processor machine often leads to super-linear speedups when carefully chosen problem sizes cause heavy garbage collection on the single processor machine while running without garbage collection on the p-processor machine. Similar effects can be observed due to p-fold increase in processor cache size.

One could argue, and this is often to be found, that it would be sufficient to use comparable implementation techniques in the sequential implementation and the parallel implementation in order to judge the potential for parallelisation, e.g. sequential interpretation vs. parallel interpretation. However, interpretation would increase the grain size by the slowdown factor of the interpreted implementation compared to a compiled version, thus reduce the parallel computation overhead, and therefore give the parallel implementation an unfair advantage. The requirement that the best sequential implementation be the point of comparison prevents this possible way of cheating oneself.

Here is an example showing, how one can cheat oneself and people with the same conviction:

```
pfib n == if n < n0
          then fib n
          else spawn f1 == pfib (n-1)
                     f2 == pfib (n-2)
               in f1 + f2

fib 0 == 0
fib 1 == 1
fib n == fib (n-1) + fib (n-2)
```

`pfib` generates an exponentially growing number of tasks (available parallelism!) with no communication between independent tasks. `n0` is used for granularity control. It should be determined taking the number of processors, the computing and the communication power, and even n into account to obtain an impressive speedup. It's the perfect parallel program! Of course, one should not compare it to the linear time sequential program computing Fibonacci numbers.

3 Abstract parallel machine for graph manipulation

To implement a lazy functional language one uses a parallel abstract machine [RRH+88,Joh83] which abstracts from some nasty features of real life machines such as fixed number of processors with bounded memory size. Executing a functional program will then dynamically create an a priori unknown number of incarnations of sequential abstract machines. These must at any time be mapped to the set of physical processors. This is the job of a network operating system.

Because we are here not interested in the details of compiling functional languages into abstract machine code [Pey87,WM95], we introduce another parallel machine, which is reduced to the bare minimum of functionality needed to create and delete sequential incarnations and to manipulate the storage structures they work on.

3.1 Mutator machine

The *parallel mutator machine* [Rab94] which we use forgets about most details of graph reduction. So it is possible to implement the essentially parallel aspects of any kind of parallel functional machine like parallel versions of the G-machine [Joh83], e.g. the νG-machine [Joh91], the CAM [CCM85], the MaMa [WM95] or the TIM [FW87] on this parallel mutator machine. With slight modifications it will also be possible to adapt it to a parallel WAM for a Prolog implementation. The name of the machine is adopted from the garbage collection nomenclature. A similar machine has been described by Kesseler [Kes96].

The machine has a dynamically changing number of (sequentially working) mutators which together evaluate a common global graph that can be accessed (read and written) from all mutators. It only allocates nodes and never frees a single node. A mutator has its stack, some registers and the code as private components. The graph and the data structure representing the set of currently existing mutators are common components.

It is the job of an operating system to simulate an unbounded number of mutators each with unbounded memory on a concrete limited target architecture.

Mutators may turn into *processes* of the operating system. Hence, the two major duties of the operating system are process management and memory management.
There are instructions of the sequential machine to:

manipulate the stack :
push makes a copy of a graph pointer
slide removes graph pointers
manipulate the graph :
mknode creates a node with contents taken from the stack
update overwrites a node
getlabel reads a node if evaluated; may lead to blocking
getsuccs reads the successors of a node referred to from the top of the stack
manipulate mutators :
start creates a new mutator for a reference to some unevaluated expression
stop finishes the mutator or the whole computation, resp.

These instructions reflect the graph manipulations by functional programs (almost) independently of the reduction strategy. So we can abstract from these aspects. All that is left is the functionality, which needs support by the operating system.

3.2 Operating System

The operating system consists of two parts: The *memory manager* distributes the common data (the graph) over the memory of the different processors and reclaims unused space (garbage collection). The *process manager* takes care of the mutators and schedules them to the different processors. In the next sections, we describe what their duties are, and what a minimal functionality they have to offer is. Different implementations of the operating system will implement these functions differently. This is discussed in detail in [Rab94].

Memory Management The abstract machine deals with abstract graph addresses (*references*). These references have to be mapped to real machine addresses. Accesses to graph nodes are then translated into local addresses in case of a local node, and into request messages to the remote owner in case of a remote node. Any choice of a heap representation of graphs is connected to a trade-off between costs for local accesses and costs for conversion of the representation when graph parts are communicated. Any scheme using direct local access needs expensive back and forth conversion before and after sending graph parts. Depending on the computation on the graph, there may be a different representation being both efficient in local access and in communication, cf. Kesseler's Thesis [Kes96].

Call-by-need semantics of the functional language to be implemented suggests that expressions with several "costumers" are shared, that they be evaluated by the first one to ask for the value, and that the others access the value. The meaning of a program will, however, not change, when the expression is copied to and evaluated by several remote costumers. It is a question of design whether to make copies of subgraphs to different processors, or to have only a single, possibly shared copy of each subgraph. Note, that this is the only chance to change the amount of problem inherent communication.

Assume we have n consumers of the result of a computation located at different processors. Then sharing causes one single evaluation, time e, and for each remote access one message to the owner and either one blocking message, time t, and one wake-up message including the result, time d_{result}, or one copy message with the requested value, time d_{result}. Duplication causes n evaluations and one access message to the owner, time t, and $n-1$ copy messages with the unevaluated subgraph, time d_{work}.

Summarizing we have:

	evaluations (e)	tag-messages (t)	data-messages (d_X)
Sharing	1	$2 \times (n-1)$	$n-1$
Duplication	n	$n-1$	$n-1$

which leads to the inequality

$$\underbrace{e + (n-1) \times (2 \times t + d_{result})}_{sharing} > \underbrace{n \times e + (n-1) \times (t + d_{work})}_{duplication}$$

This can be simplified (assuming $n \neq 1$) slightly to

$$t - e \times \frac{n+1}{n-1} + d_{result} > d_{work}$$

Satisfaction of the inequality favours duplication, otherwise sharing. Of course, the existence of local costumers may favour sharing as it avoids duplication of work and does not need communication.

> A decision about whether to copy or to share would need information about the relative sizes of the e, d_{work}, d_{result} which are unknown. The value of n is unknown, too.

Independently of the particular method used, the memory manager needs information about

– the structure of the graph nodes (which information is a reference to another node, and the size of objects)
– the pointers from the stack and registers of the mutators to the graph. These pointers are needed to find unused nodes and to rearrange the heap.

Additionally, one has to decide which garbage collection method to use, cf. section 5.

The minimal functionality of the memory manager needed for the implementation of mutator machines is:

alloc	creates a node of a given type and returns a new reference to it
copyref	makes a copy of a reference
delref	informs the memory subsystem that a reference is no longer in use
write	updates the information in a graph node
read	extracts information from a graph node

alloc creates references for real addresses; the other operations receive references as parameters, decode them to real addresses and take effect there.

Different implementations of the memory manager will implement these functions differently.

Process Management The decision, which computations should form a process is taken in several stages:

1. The language implementor selects the language constructs, which may potentially lead to parallel activities; either there are some explicitly parallel constructs in the language, or he selects those whose parallel execution is possible and supposedly profitable;
2. the compiler based on information about the program may decide whether a computation is worth to be executed in parallel and should become a mutator and correspondingly, whether to generate a *start* for it;

3. the operating system knowing the actual load can decide which mutators to convert into processes.

The process manager administrates the (dynamically changing number of) sequential mutators. In particular, it schedules them to the different processors. The main goal is to minimize the maximal running time of any processor; it may help to evenly distribute the work among the processors.

The process manager needs to offer at least the following functions:

fork	creates a new mutator for a subgraph; *may* result in a new process
dispatch	attempts to find a waiting process
suspend	suspends a process not able to continue evaluation
resume	resumes a process; reason of blocking disappeared
finish	terminates a process

The following table shows the relation between the mutators and the operating system. Mutator instructions on the left side are implemented using the operating system commands on the right side.

mknode	alloc, write
push	copyref
slide	delref
update	write, resume
getlabel	read, suspend
start	fork
stop	finish; dispatch

mknode calls alloc to obtain a new heap area and write to store a label and to pop a list of stack addresses into this area.

push copies a graph pointer into the stack. May call copyref depending on the kind of garbage collection used.

slide deletes a list of heap graph pointers from the stack; may have to call delref depending on the type of garbage collection used.

update overwrites a graph node using write. Calls resume if a blocked process waited for the value just written.

getlabel issues a read–command. The process trying to access a node is blocked by a suspend–command if the node isn't yet evaluated. Another process is scheduled by executing the dispatch–command.

start creates a new mutator. This mutator may be turned into a process by calling fork.

stop terminates the mutator. It calls finish to terminate the corresponding process and then *dispatch* to schedule another ready process.

4 Load balancing, scheduling

The goal of load balancing is:

Distribute work such that the maximal execution time on any processor is minimized

Global scheduling of processes to processors is in charge of achieving a good load balance. The above stated problem can in general not be solved statically, since the work is evolving dynamically. This means, that one has to deal with an online problem. It could be assumed, that an even distribution of the work among the processors guarantees satisfaction of the above goal. However, this need not be true due to the relative high costs of communication over computation. Any scheduling algorithm requiring too much communication, e.g. for gathering load information, may achieve a balanced work distribution, but will fail to minimize the overall execution time.

4.1 Scheduling

There is a whole zoo of scheduling problems, each characterised by a set of parameters [BESW94,GLLK79]. [BESW94] use three sets of together 11 parameters. They firstly concern the characteristics of the processor sets, i.e. single processor or multiple identical/uniform/dedicated and the actual number of processors. Secondly, they describe task and resource characteristics. These are the possibility of preemption, the existence of other resources, information about task processing times, i.e. uniform time, bounded times, arbitrary, but known, and dynamic, and the type of dependences between tasks. These range from no dependence over chains and forests to arbitrary dependences. The last parameter indicates the optimization criterion. There is one parameter missing, i.e. the one saying whether the scheduler works in a centralised or in a decentralised fashion. The former is assumed as default.

Most scheduling problems are NP-complete. Turning one parameter to a worse value usually makes the situation in some way more undesirable. A problem solvable in polynomial time may turn into an NP-complete problem. An NP-complete problem, for which a standard heuristics is guaranteed to nearly reach the optimum, may turn into one, for which no such good bound can be proved. Finally, an offline problem may turn into an online problem.

Our setting belongs almost to the worst in the zoo. We have a dynamically changing set of tasks, whose processing times in general are unknown. Rather general dependences evolve during the execution. In addition, we have to do with decentralised scheduling, and both the scheduling algorithm and the reallocation of work cause communication with nonnegligible costs. To the best of our knowledge,

> there is no algorithm known to solve our scheduling problem
> with any guaranteed performance.

There are (as always) strategical and technical questions about the scheduling method. The strategical questions are:

when should process migration be initiated?

which processes should migrate? One could decide to only migrate processes, whose results are needed. It has, however, been tried to increase the amount of parallelism by also migrating processes, whose results are not known to be needed. This is called *speculative evaluation*. Kesseler[Kes96] argues, that no proposed method should be used in managing speculative parallelism, since they are all too expensive in communication.

where, from where should a process migrate? There are two extreme answers to this this question. The first is the attempt to preserve locality by migrating processes only to neighbours. This risks a slow distribution of work. The other is to randomly distribute work [SW89]. The latter achieves provably good load balancing under realistic assumptions, but will probably be too expensive in communication.

The technical questions are:

how should process migration be executed? By *fork, resume*, or *dispatch*? The main characteristic is, whether an overloaded processor *actively* distributes work, i.e. tasks from its task queue, or whether idle (*passive*) processors ask other processors for excess work. A second design issue is, whether only newly created processes are migrated or also older processes together with their context. Hence, the following design choices are available: Actively distribute only new processes applying *fork* on the same processor, actively distribute all processes applying *dispatch* to the own task queue, or passively acquire work using *dispatch* applied to another processor's task queue.

The main disadvantages of active strategies are that idle processors may wait a long time, since they cannot influence the distribution, and that busy processors are additionally burdened with collecting load information and distributing processes.

Passive strategies have as main advantages, that only idle processors ask for work. If all processors are busy, no communication takes place.

how much should migrate with the process? This question has also to be solved under the condition, that communication should be minimized. Copying a graph from one processor to another one can take two forms. The basic techniques are *eager* copying and *full lazy* copying. The former means that the graph is copied as a whole, the latter transmits only nodes when they are needed. Advantages of the first approach are that only a single message is needed to transmit the graph which can decrease communication overheads. The latter may introduce a lot of small messages which can be costly, if the whole graph is needed, but is good if only small parts are needed. *Full lazy* copying also increases the (possible) sharing of results. Kesseler [Kes96] developed a combined technique, called *lazy* copying. It tries to avoid the copying of work, while retaining the advantage of copying multiple nodes together. It copies nodes unless it hits a redex which has a *defer attribute*. These nodes represent work, which should not be copied. Once a deferred node has been evaluated, it looses its defer attribute and may be copied. Deadlock avoidance requires an evaluation of each deferred node.

> Which node should have a *defer* attribute initially?

Some schedulers base their decision on actual load information. The whole system has to be halted or an elaborate and expensive protocol to collect a snapshot has to be executed to gather exact load information. Thus, only approximate information can realistically be collected. Higher demands on the precision and the actuality of this information increase the communication costs of the computation. Kuchen and Wagener [KW90] have experimented with several frequencies of collection. Collecting load information periodically and after each change in the load turned out to be too expensive. Most promising was the strategy to collect each time a processor's state turned from idle to busy and vice versa.

Another aspect, which could be relevant for scheduling decisions is memory consumption on the different processors. This has so far not been regarded.

5 Distributed garbage collection

Naive conversions of sequential garbage collection algorithms to parallel ones often have high communication costs due to the necessary synchronization overhead. In a distributed multiprocessor system each processor is responsible for allocating and reclaiming structures residing in its local memory. Traditionally, the following three techniques are distinguished; each could be used in a parallel context.

mark/scan
: Cells are not reclaimed immediately when they become garbage. Collection works in two phases. The first (mark) phase starts from all references to heap cells in stacks and registers and marks the reachable cells. The second (scan) phase reclaims the unmarked cells (normally by a move/fix-up run). It takes several runs over the heap. During garbage collection the computation is halted. In a parallel implementation, there is substantial communication during the scan phase. To detect termination of the mark phase one needs an expensive global snapshot computation.

reference counting
: Every node x has an additional attribute, its *reference count* ($RC(x)$). It stores the number of pointers to the node. The operation alloc initialises the count, delref and copyref update the reference counts. Garbage is reclaimed when a count becomes zero. The additional space storing the reference counts and the inability to reclaim cycles are the main drawbacks.
: In distributed memory management, reference counting needs communication in the case of non-local references. A *non-overtaking* message system is required because garbage is reclaimed when a count becomes zero thereby possibly ignoring pending increment messages.

copying
This technique divides the heap equally into two *semi-spaces*. During computation, one semi-space contains the current data and the other obsolete data. Garbage collection is done by copying live cells to the obsolete semi-space and then swapping the roles of the two parts. In contrast to the *mark/scan* it needs only one run over the heap but twice the space.

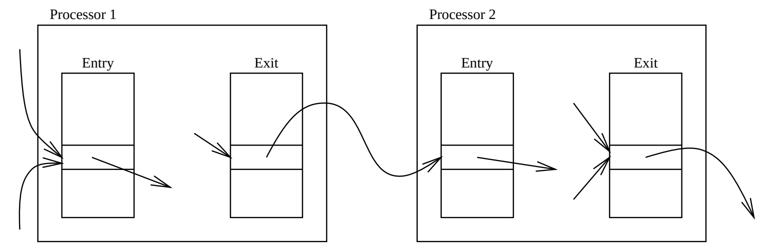

Fig. 1. Entry and exit tables

Any method used in the distributed case should avoid global synchronisation and should minimize communication. To decouple the activities of the different processors, garbage collection is split into two parts: *local* and *global* garbage collection. Pointers are divided into local and global ones. Non-local references are managed in entry and exit tables at each processor, cf. Figure 1. Global references into a heap go indirectly through the entry table of the processor, global references from a heap go indirectly through the exit table of that processor. Local garbage collection is normally done via a form of the *mark/scan* technique. A technique with a priori low communication costs is chosen for global garbage collection, namely reference counting. One has to extend this technique to be able to reclaim non-local cycles. The required non-overtaking message passing system can be realized spending additional space as follows. *Weighted reference counting* [Bev89] reduces the amount of communication needed for reference counting. It associates a weight $W(X, Y)$ with each reference from X to Y. The sum of all reference weights is equal to the reference count of the node:

$$RC(Y) = \sum_{X \to Y} W(X, Y)$$

Initially, a node created by alloc is given a sufficiently large power of two as reference count. When a reference X to Y is asked to supply a copy of the reference i.e. copyref, it splits its weight into half, keeping one half for itself and passing on the other half with the new copy. Indirection cells are needed when the weight of a reference to be copied has reached one. Thus, copying does not need communication.

Deleting a reference by delref requires a message to the owner of the node containing the weight of the reference. The owner subtracts the weight from its reference count. When the reference count falls to zero, the node is identified as garbage. Because only deletion message are needed, it can run on an *over-taking* message system. Figure 2 shows the three needed operations.

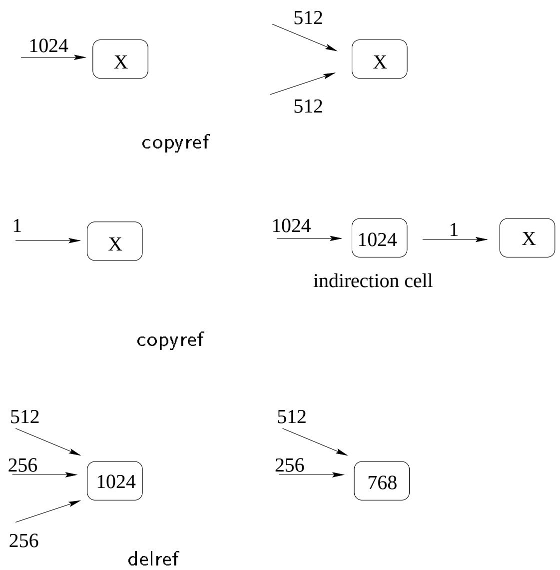

Fig. 2. Weighted reference count technique

The technique of Lins and Jones [LJ91], *cyclic weighted reference counting*, solves the problem of reclaiming cycles. They associate a second reference count (SRC, only temporary for mark phase) with each cell. A color is assigned to each cell during garbage collection: *red* for potential garbage (cyclic), *green* for live cells, and *blue* for garbage. It keeps a control queue for potential entries to garbage cycles. A node is put into the queue at a delete reference operation, if the reference count is not zero ([LV91] give a discussion of managing the control queue). At garbage collection time, a node is selected from the control queue. Then a *mark red* phases colors all reachable cells, updating additionally the SRC. In an additional run, cells are divided into green (SRC>0) and blue cells. A further run reclaims the blue cells and removes related references from the control queue. This technique causes much synchronization communication to detect termination of the separated runs and also global synchronisations (*barrier synchronisation*).

It is still not clear how to reclaim efficiently and concurrently (non-local) garbage.

6 Conclusion

Of course, no negative results of the type "parallelism in functional program cannot be exploited" have been proved. Techniques developed in other areas may come to help. Increasing the grain size by lazy threads [GSC95], prefetching needed remote data by sampling [BAW] may lead to overlapping computation and communication and thus to reduced wait times. However, the problem doesn't seem to admit an easy solution.

7 Acknowledgements

The authors thank Susanne Albers, Yosi Ben Asher, Thomas Rauber, and Gudula Rünger for valuable discussions.

We would like to end this article in the spirit of our starting quotation.

Parallelism in functional programs is maximal (up to exploitation).

References

[AB90] V. Bafna A. Gupta, A. Banerjea, V. Jha and PCP Bhatt. Parallel implementation of logic languages. In *CONPAR 90 - VAPP IV*, pages 154–165, Berlin - Heidelberg - New York, 1990. Springer.

[ACC+90] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. *1990 International Conf. on Supercomputing*, June 11-15 1990. Published as Computer Architecture News 18:3.

[AK96] I. Alouini and C. Kirchner. Toward the Concurrent Implementation of Computational Systems. In *Algebraic and Logic Programming*, volume 1139 of *LNCS*, pages 1–31. Springer-Verlag, September 1996.

[AKP91] F. Abolhassan, J. Keller, and W.J. Paul. On the Cost-Effectiveness of PRAMs. In *SPDP*, pages 2–9. IEEE Computer Society, 1991.

[AMT93] C. Atay, K. McAloon, and C. Tretkoff. 21p: A highly parallel constraint logic programming language. In Linda R. Petzold Richard F. Sincovec, David E. Keyes, Michael R. Leuze and Daniel A. Reed, editors, *Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing*, pages 822–828, Norfolk, VI, March 1993. SIAM Press.

[BAW] Y. Ben-Asher and R. Wilhelm. Compiler support for fair execution of shared memory parallel programs on networks of workstations. in preparation.

[BESW94] J. Błazewicz, K. H. Ecker, G. Schmidt, and J. Węglarz. *Scheduling in Computer and Manufacturing Systems*. Springer-Verlag, 1994. Second, Revised Edition.

[Bev89] D.I. Bevan. An efficient reference counting solution to the distributed garbage collection problem. *Parallel Computing*, 9(2):179–192, 1989.

[Bur90a] G. L. Burn. Implementing Lazy Functional Languages on Parallel Architectures. In P. C. Treleaven, editor, *Parallel Computers — Object-Oriented, Functional, Logic*, Series in Parallel Computing, chapter 5, pages 101–140. John Wiley & Sons, Chichester, 1990.

[Bur90b] G. L. Burn. Using Projection Analysis in Compiling Lazy Functional Programs. In *LISP'90, Nice, France*, pages 227–241. ACM Press, 1990.

[CCM85] G. Cousineau, P. L. Curien, and M. Mauny. The categorical abstract machine. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 50–64. Springer-Verlag, Berlin, DE, 1985. Lecture Notes in Computer Science 201Proceedings of. Conference at Nancy.

[Cha94] M. M. T. Chakravarty. A self-scheduling, non-blocking, parallel abstract machine for lazy functional languages. Proceedings of the 6th International Workshop on the Implementation of Functional Languages, 1994.

[FMS+95] J. Feo, P. Miller, S. Skedzielewski, S. Denton, and C. Solomon. Sisal 90. In A. P. Wim Bohm and John T. Feo, editors, *High Performance Functional Computing*, pages 35–47, April 1995.

[FW87] J. Fairbairn and S. Wray. TIM: A simple, lazy abstract machine to execute supercombinators. In *Functional Programming Languages and Computer Architecture*, pages 34–45. Springer Verlag, September 1987. LNCS 274.

[FWM94] G. C. Fox, R. D. Williams, and P. C. Messina. *Parallel Computing Works!* Morgan Kaufmann, 1994.

[GLLK79] R. Graham, E. Lawler, J. Lenstra, and A. Rinnoy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. In *Annals of Discrete Mathematics*, volume 5, pages 287–326. North–Holland, Amsterdam, 1979.

[GSC95] S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy threads, stacklets, and synchronizers: Enabling primitives for compiling parallel languages. In *Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, 1995.

[Hil95] J. M. D. Hill. Data-parallel lazy functional programming (Phd Thesis). Technical Report QMW-DCS-1995-701, Queen Mary and Westfield College, Department of Computer Science, Mile End Road, London, E1 4NS, UK, May 1995.

[Joh83] T. Johnsson. The G-machine: An abstract machine for graph reduction. In *Proceedings of the Joint SERC/Chalmers Workshop on Declarative Programming, University College*, London, UK, 1983.

[Joh91] T. Johnsson. Parallel evaluation of functional programs: The $<\nu$,G$>$ approach (summary). In *Parallel Architectures and Languages Europe PARLE*, pages 406–422. Springer-Verlag, 1991. LNCS 505.

[Kes96] M. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distributed Memory.* PhD thesis, "Katholieke Universiteit Nijmegen, The Netherlands", April 1996.

[KW90] H. Kuchen and A. Wagener. Comparison of dynamic load balancing strategies. Technical Report 90-05, Technical University of Aachen (RWTH Aachen), 1990.

[LJ91] R. D. Lins and R. E. Jones. Cyclic weighted reference counting. Technical Report 95, University of Kent, Canterbury, United Kingdom, December 1991.

[LV91] R. D. Lins and M. A. Vasques. A comparative study of algorithms for cyclic reference counting. Technical Report No. 92, Computing Laboratory, University of Kent, Canterbury, UK, 1991. Submitted to Software - Practice and Experience.

[Mau86] D. Maurer. Strictness Computation Using Special Lambda-Expressions. In Harald Ganzinger and Neil D. Jones, editors, *Programs as Data Objects*, volume 217 of *LNCS*, pages 136–155. Springer-Verlag, October 1986.

[NSEP91] E. G. J. M. H. Nöcker, J. E. W. Smetsers, M. C. J. D. Eekelen, and M. J. Plasmeijer. Concurrent Clean. In E. H. L. Aarts, J. van Leeuwen, and M. Rem, editors, *PARLE '91, Parallel Architectures and Languages Europe, Volume I: Parallel Architectures and Algorithms*, volume 505 of *Lecture Notes in Computer Science*, pages 202–219, Eindhoven, The Netherlands, June 10–13, 1991. Springer, Berlin.

[NSMC92] J. H. Nang, D. W. Shin, S. R. Maeng, and J. W. Cho. A parallel execution of functional logic language with lazy evaluation. In *Fifth Generation Computer Systems*. IOS Press, Amsterdam, NL, 1992.

[Pey87] S. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[Rab94] M. Raber. *Parallele Implementierung funktionaler Programmiersprachen*. PhD thesis, Universität des Saarlandes, 1994.

[RRH+88] M. Raber, Th. Remmel, E. Hoffmann, D. Maurer, F. Müller, H.-G. Oberhauser, and R. Wilhelm. Compiled graph reduction on a processor network. In *'Architektur und Betrieb von Rechensystemen', 10. GI/ITG-Fachtagung*, pages 198–212, Paderborn, FRG, March 1988. *Informatik Fachberichte 168*, Springer-Verlag.

[SW89] H. Seidl and R. Wilhelm. Probabilistic load balancing for parallel graph reduction. In *Proceedings IEEE Region 10 Conference*, pages 879–884, New York, 1989. IEEE.

[THJ+96] P W Trinder, K Hammond, J S Mattson Jr, A S Partridge, and S L Peyton Jones. GUM: a portable parallel implementation of Haskell. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 79–88, Philadelphia, Pennsylvania, 21–May 1996.

[Tre94] G. Tremblay. *Parallel implementation of lazy functional languages using abstract demand propagation*. PhD thesis, McGill University, November 1994.

[WM95] R. Wilhelm and D. Maurer. *Compiler Design*. International Computer Science Series. Addison–Wesley, 1995.