J. Functional Programming **7** (4): 421–440, July 1997. Printed in the United Kingdom

A competitive algorithm for managing sharing in the distributed execution of functional programs†

GAD AHARONI, AMNON BARAK and AMIR RONEN

Department of Computer Science, The Hebrew University of Jerusalem, 91904 Jerusalem, Israel
(*e-mail:* gadi@cs.huji.ac.il)

Abstract

Execution of functional programs on distributed-memory multiprocessors gives rise to the problem of evaluating expressions that are shared between several Processing Elements (PEs). One of the main difficulties of solving this problem is that, for a given shared expression, it is not known in advance whether realizing the sharing is more cost effective than duplicating its evaluation. Realizing the sharing requires coordination between the sharing PEs to ensure that the shared expression is evaluated only once. This coordination involves relatively high communication costs, and is therefore only worthwhile when the shared expressions require much computation time to evaluate. In contrast, when the shared expression is not computation intensive, it is more cost effective to duplicate the evaluation, and thus avoid the communication overhead costs. This dilemma of deciding whether to duplicate the work or to realize the sharing stems from the unknown computation time that is required to evaluate a shared expression. This computation time is difficult to estimate due to unknown run-time evolution of loops and recursion that may be part of the expression. This paper presents an on-line (run-time) algorithm that decides which of the expressions that are shared between several PEs should be evaluated only once, and which expressions should be evaluated locally by each sharing PE. By applying competitive considerations, the algorithm manages to exploit sharing of computation-intensive expressions, while it duplicates the evaluation of expressions that require little time to compute. The algorithm accomplishes this goal even though it has no *a priori* knowledge of the amount of computation that is required to evaluate the shared expression. We show that this algorithm is competitive with a hypothetical optimal off-line algorithm, which does have such knowledge, and we prove that the algorithm is deadlock free. Furthermore, this algorithm does not require any programmer intervention, it has low overhead, and it is designed to run on a wide variety of distributed systems.

Capsule Review

In a sequential lazy functional programming language, call-by-need combines the advantages of call-by-value and call-by-name (ignoring time overheads and space considerations). An argument to a function is evaluated only if needed and never more than once, no matter how often its value is used. When the argument is first evaluated, the unevaluated expression is overwritten with the computed value, thereby sharing the result and avoiding recomputation. However, in a distributed implementation of a functional programming language, this sharing

† This research was supported in part by the Basic Research Foundation administered by the Israeli Academy of Science and Humanities and in part by the Ministry of Science and the Art.

of a value may introduce significant synchronization and communication costs. In many cases, for example when the argument is a simple numerical expression, the costs of sharing may be far greater than the cost of evaluating the argument.

This paper proposes a simple optimization that reduces the above problem. Each shared value has an executive process. When a shared value with an executive process on another processor is required, a little bit of time is invested in trying to re-evaluate the expression locally before a request for its value is sent to the executive process. The time spent trying to re-evaluate the expression locally is at most a lower bound on the time required to obtain the value from another processor. This ensures that there is at most a constant factor cost for this method. On the other hand, if the value can be quickly computed locally, then a significant amount of time may be saved.

The paper contains some analytical and experimental results, indicating that the proposed method is likely to be of value in practice.

1 Introduction

One of the advantages of functional programs is their inherent implicit parallelism, which facilitates execution on parallel systems. During the execution of such programs many tasks may be created, some of which may be sent for parallel execution to other Processing Elements (PE). This paper addresses a problem that occurs *after* tasks are sent for parallel execution, namely that of managing the evaluation of expressions that are shared between several PEs.

In shared-memory parallel systems, in which all the PEs share the same address space, it is not difficult to avoid duplication of work on shared expressions. In such systems, as soon as an expression is evaluated by one PE, the result is immediately visible to the other PEs. Exploiting the sharing of expressions is thus a desirable and cost-effective feature in shared-memory systems. However, in loosely-coupled distributed-memory systems, exploiting such sharing becomes a non-trivial problem. Ensuring that when an expression is evaluated by one PE, it is not evaluated again by other PEs, requires cooperation between the PEs sharing the expression. This cooperation incurs communication costs that are relatively high in distributed systems. Therefore, in such systems there are cases in which it is more cost effective to duplicate the work by evaluating the shared expression locally on each of the PEs that share it, rather than incurring the cost of negotiating with the other PEs to exploit sharing.

In general, exploiting sharing is effective only when the amount of computation involved in evaluating the shared expression is larger than the cost of coordinating between the PEs. The problem is that, while the communication costs are generally known, there is usually no *a priori* knowledge of the costs of evaluating an expression. This paper addresses the question of which shared expression should actually be shared, without knowing the amount of computation involved in the evaluation of the shared expression.

In this paper we concentrate on graph-reduction-based models that are implemented on loosely-coupled distributed systems. An algorithm for managing the sharing of expressions in such a model should address the following issues:

https://doi.org/10.1017/S095679689700275X Published online by Cambridge University Press

- How to identify a shared expression and how to determine who is sharing it.
- How to decide whether to realize the sharing (effective only when the evaluation of the expression requires much computation time, because of the high communication costs), or whether to duplicate the evaluation of the shared expression (effective only when the expression is computationally small), without having information about the amount of work required to evaluate the shared expression.
- How to realize the sharing efficiently, in terms of minimizing the number of messages passed in the system, and their length.
- How to guarantee a low overhead of the algorithm.

Early implementations of parallel functional models were on shared-memory systems (Darlington, 1981). Such systems enable the synchronizations between the PEs to be mediated via the shared memory. Later implementations were and are based on loosely-coupled distributed-memory systems (Flanagan and Nikhil, 1996; Goldberg, 1988; Kelly, 1989; Kesseler, 1996; Plasmeijer and van Eekelen, 1993; Raber *et al*., 1987; Trinder *et al*., 1996), which facilitate the scalability of the system, i.e. the ability to increase the number of PEs without degrading the performance. However, the cost of cooperation between the PEs in such systems may be rather high due to the potential high communication overhead.

Existing distributed-graph-reduction systems solve the sharing problem by avoiding the sending of shared subgraphs from one PE to another. Instead, *remote pointers* that reference the shared subgraphs in the source PE's address space are sent. (Some systems, for example the GRIP machine (Peyton Jones *et al*. 1990), do send shared subgraphs if they are in Weak Head Normal Form.) This scheme incurs the communication overhead costs for each access of a remote pointer. In other words, this scheme *forces* the system to realize the sharing regardless of the amount of computation involved in evaluating the expression. No attempt has been made to optimize the sharing procedure in a way that avoids the heavy communication costs for shared expressions that require little time to compute.

We solve the dilemma of whether to duplicate the work or to realize sharing by using competitive considerations (Sleator and Tarjan, 1985) that avoid the communication overhead when the shared expressions are computationally small. Let E be some shared expression whose value is needed, and let C be the estimated cost of realizing the sharing. The algorithm first applies (at most) C local evaluation steps to E. Then, if E is not evaluated within these C steps, the sharing is realized. This strategy is highly effective since, in the worst case, it incurs only twice the optimal cost. (This case occurs when the evaluation of E requires $C+1$ steps.) Note that the optimal cost can only be achieved if the cost of evaluating expression E were known *a priori*, which is not the case here because E might include loops and recursion. We call this strategy the Competitive Sharing Management (CSM) algorithm. We are able to show the effectiveness of this algorithm in comparison to the hypothetical optimal algorithm, and its deadlock-free nature.

The CSM algorithm assumes that the PEs support a bounded form of speculative computation (Mattson, 1993; Partridge 1991). This support is required in case the

initial C local evaluation steps do not fully evaluate the expression, and the expression's graph should revert to its original form. The detailed description of the CSM algorithm (Section 4) refers to this case as the 'draw-back' mode of the algorithm, and provides a brief explanation of how to perform this operation efficiently.

Our model of computation captures the main issue in deciding when to realize sharing, i.e. the communication overhead. But it makes some simplifying assumptions that avoid some of the complexities of real-life situations. For example, we assume that the overhead costs of the communication are known, whereas in real systems it is sometimes difficult to predict the total communication costs in advance, and we ignore various costs that arise from supporting speculative computation. Nevertheless, the simplicity of the model enables us to provide formal proofs about the presented algorithm, while taking all the details into account would make the analysis intractable.

This paper is organized as follows. A brief description of distributed graph reduction is presented in section 2. The computational model is defined in section 3. section 4 introduces the CSM algorithm. Section 5 gives three schemes for appointing executive processes, which are needed for efficient realization of sharing. Section 6 discusses a modification of the CSM algorithm. Section 7 proves that the CSM algorithm is deadlock free. Section 8 analyses the algorithm's performance. Section 9 presents experimental performance results of the CSM algorithm. Section 10 addresses the implementation of the CSM algorithm in optimized graph-reduction-based models. Finally, section 11 concludes the paper.

2 Graph reduction

In the graph reduction model, reduction rules are repeatedly applied to the graph representing the program, until the graph is reduced to Weak Head Normal Form (WHNF), at which point it represents the result of the evaluation. The graph contains four types of nodes: Constants, Functions, Formal Variables and Apply nodes.

Let N denote the root of a subgraph (an expression). If N is a constant or a function, then the graph is already in WHNF. If N is an Apply node, that is, N is in the form $Apply(F, E_1, \ldots, E_k)$ then the function F is reduced next. Let F' denote the reduced function F, then after the reduction of F, node N is in the form $Apply(F', E_1, \ldots, E_k)$. If F' is a primitive function then the strict arguments of F' are reduced, followed by applying F', and finally updating node N with the result. If function F' is not a primitive function (it is a lambda expression), then the graph-reduction algorithm makes a copy of the body of the function F', substituting pointers to the formal arguments in the copy with pointers to the actual arguments $E_1 \ldots E_k$. The root of the copy of the body of F' is reduced next, the result of which updates node N. Note that the lazy semantics of the model often increases the number of shared expressions. This brief description of graph reduction is given here mainly for terminology purposes. A detailed description of (iterative) graph reduction and its implementation can be found in Field and Harrison (1988) and Peyton Jones (1987).

2.1 Distributed graph reduction

One of the advantages of graph reduction is that it is inherently a parallel activity, since very often several reduction rules can be applied to the graph at the same time. Since the meaning of a functional expression does not depend on the environment, the evaluation may be performed in parallel and thus accelerate the computation. Graph-reduction-based models are especially suitable for execution on distributed-memory systems, where several processes simultaneously evaluate different portions of the graph. Each such evaluator process keeps the set of all reducible nodes in a local *task pool*.

2.2 Sharing in graph reduction based models

Sharing occurs in the application of a function with a formal parameter that appears more than once in the body of the function, to some arguments. The graph reduction model prevents the duplication of work on shared subexpressions by using pointers. Since all the expressions that share the subexpression point to the same subexpression, once this subexpression is fully evaluated all the sharing expressions are updated with the result. To identify a shared expression, a *reference count* (reference bit) is maintained at each node. The reference count holds the number of pointers pointing at that node. Shared expressions have a reference count that is larger than one.

Such sharing management is difficult on distributed-memory systems. Copies of various shared expressions may reach different PEs, and preventing duplication of work involves relatively expensive communication between the PEs. If the shared expressions require little time to compute, then it may be more worthwhile to duplicate the computation than to incur the communication cost. The CSM algorithms presented in this paper solves this problem.

3 The computational model

We assume a loosely-coupled distributed-memory system with a bounded number of PEs, which communicate via messages. Each PE may evaluate several functional expressions, and may migrate them from one PE to another according to various load and demand considerations. More formally, each PE may run several graph-reducers, which we shall call *evaluators*. Each evaluator is spawned to reduce a functional expression to WHNF. We shall refer to an evaluator as an independent (light weight) processes having its own task pool and message queue. Note that this is a logical view of the system made to simplify the description and the analysis of the system. The actual implementation may take into account other design considerations, and may for example choose to share resources such as queues between evaluators.

During reduction, an evaluator p may decide to spawn another evaluator p' for reducing some subgraph E' in parallel. The spawn includes creating a child process p', and copying subgraph E' into the address space of the child evaluator (see optimizations in section 10). Evaluator p' is responsible for reducing E'. When p'

https://doi.org/10.1017/S095679689700275X Published online by Cambridge University Press

$F = \lambda x.\lambda y.\lambda z.\ +\ (H\ x\ z)\ (G\ y\ z)$
$H = \lambda a.\lambda b.$ IF $(cond1\ a)$ THEN b ELSE 0
$G = \lambda a.\lambda b.$ IF $(cond2\ a)$ THEN b ELSE 1
$W = \lambda x.$ `usually a lot of work`

Fig. 1. Example sharing program.

finishes the evaluation, it returns the result of the reduction to p and terminates. The parent p overwrites the root of the spawned subgraph E' with the result that it has received. This form of spawning can be described by a tree of processes, where each node represents an evaluator. The root of this tree is the evaluator that is spawned to reduce the top-level expression (the program).

Realizing the sharing is achieved in our model by appointing one of the sharing processes to evaluate the shared expression and then having the other sharing processes turn to this process when they need the result of the evaluation. More specifically, for each shared expression E, the algorithm nominates an *executive process* (evaluator) P_E such that E belongs to the graph of P_E, and every evaluator that would need to evaluate E would be a descendant of P_E.

Each evaluator process has a single message queue from which it receives messages. When an evaluator needs the result of a spawned expression, it waits (sleeps) on its message queue until a message arrives.

The model assumes that the evaluator processes can exchange messages, where the cost of sending and receiving each message is D. We assume a fixed cost D in order to simply the analysis. See section 10 for further discussion about possible optimizations.

Two messages are required for the realization of a shared expression E: a request for the value of E, which is sent to the executive process of E, and the reply, which contains the result of the evaluation of E. Therefore, the overall cost of communication incurred by realizing sharing is $C = 4D$ (see section 8). Note that it is assumed that both messages are of equal cost.

The computational model that is assumed here normally has more evaluator processes than PEs. This way, when an evaluator is suspended (e.g. waiting for a message), the PE holding this evaluator can run other processes. This paper does not address questions of *when* a process should be spawned and *where* to place the process, but rather presents a solution to a problem that occurs *after* tasks have been sent.

3.1 An example

Consider the program in figure 1. Let p be a process evaluating the expression $E = (F\ 7\ 8\ (W\ 9))$. Note that p does not know in advance whether it is going to evaluate the expression $(W\ 9)$. The first step in the evaluation is to apply β reduction, which gives: $(+\ (H\ 7\ (W\ 9))\ (G\ 8\ (W\ 9)))$. Note first that the expressions

https://doi.org/10.1017/S095679689700275X Published online by Cambridge University Press

$E_1 = (H\ 7\ (W\ 9))$ and $E_2 = (G\ 8\ (W\ 9))$ may be evaluated in parallel. Also note that the expression $E_3 = (W\ 9)$ has become shared.

Assume that process p decides to evaluate E_1 itself and send E_2 for parallel evaluation. A process p' is created and given that task of evaluating expression E_2. When p finishes its evaluation of E_1, it needs the result of E_2 in order to complete the whole evaluation. If p' finished its evaluation, the result already exists on the message queue of p; otherwise, p would need to wait till p' finishes and in the meantime the PE running p can execute other processes.

Now consider the shared expression E_3. Before the sending of E_2, both pointers to E_3 pointed at the same location in the graph. If E_3 were evaluated, both E_1 and E_2 would have the result and duplicated evaluation of E_3 would have been prevented. However, once expression E_3 was sent unevaluated, taking advantage of the sharing requires expensive communication between the processes. Hence, exploiting the sharing of E_3 is worthwhile only if it turns out that $E_3 = (W\ 9)$ is a 'heavy' expression. The problem is that the weight of the expression is not known in advance.

4 The CSM algorithm

Consider a subgraph E that is shared between two different PEs. Let $R_p(E)$ be the reduction cost of subgraph E by an evaluator process p, and let C be the cost of realizing the sharing of E. If $R_p(E) < C$, then it is more cost effective to duplicate the reduction of the shared subgraph on the PEs that share it, than to incur the cost of communicating the result of the reduction to the sharing PEs. However, while C is usually known, or can be estimated fairly accurately, the value of $R_p(E)$ cannot, in general, be determined because it relies on the run-time behaviour of the program for which there is no *a priori* information. Therefore, it is not generally possible to decide in advance whether it is more cost effective to duplicate the work or to negotiate for the shared value.

The CSM algorithm suggests a solution to this dilemma of when to duplicate work, and when to request the value of a shared subgraph. The algorithm performs up to C local reduction steps on each shared expression; then, if the shared expression is in WHNF, no communication is required. If the shared expression is not reduced within C reduction steps, then the value of the shared expression is requested from the executive process. The outcome of such a strategy is that computationally small expressions are reduced locally without incurring communication overheads, while the reduction of computation-intensive expressions is not duplicated.

We now describe the details of the CSM algorithm by using the state transition diagram depicted in figure 2. An evaluator process in the system may be in either of four *states* (modes): *Normal, Sharing, Request* or *Draw-back*. The following describes the actions taken in each state of the algorithm, and the transitions between the modes:

- Normal: this is the initial state of the algorithm. In this state the evaluator process performs ordinary graph reduction, during which tasks are accumulated in the task pool. Some of these tasks may be spawned for parallel execution.

https://doi.org/10.1017/S095679689700275X Published online by Cambridge University Press

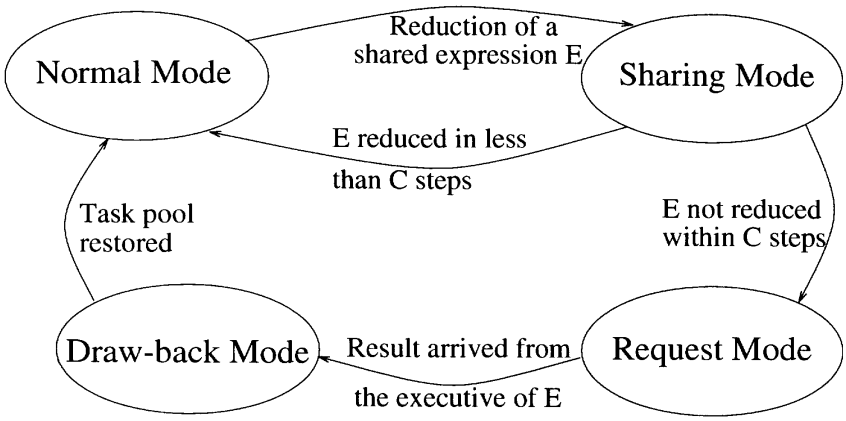

Fig. 2. A state-transition diagram describing the CSM algorithm.

- Sharing: this state is entered when an evaluator process begins to evaluate a shared expression E, whose executive process resides on another PE. In this state, (at most) C reduction steps are applied to E, where C is the cost of realizing the sharing of E. If E is fully reduced within C steps, then the evaluator returns to operate in Normal Mode. Note that in this case the evaluator process performs the optimal scheme, which is duplicating the evaluation. Otherwise, if E is still not fully reduced after C steps, then the evaluator process enters the Request Mode. It is important to emphasize that in this state *at most* C reduction steps are applied to the shared subgraph, that is, if during these C reduction steps another shared node needs to be reduced, then the Sharing state is *not* entered again.
- Request: in this state the evaluator process requests the value of E from the executive process of E. Note that while the evaluator is waiting for the result to arrive, the PE on which the evaluator process resides can run other evaluators. When the executive process of E returns the result of the shared expression, the evaluator process enters the Draw-back Mode.
- Draw-back: in this mode node E is overwritten with the result received from the executive evaluator, the task pool and the stack are restored to their previous state (before the Sharing Mode). The Depth-First Search (DFS) nature of graph reduction can be used to perform this action in constant ($O(1)$) time.

While some evaluator process p waits on its message queue, a request to evaluate some expression E might arrive, where p is executive process of E (that is, $P_E = p$). In such a case the evaluator first checks in its local-graph space if E is already reduced. If so, then p can return the result immediately. Otherwise, if E is currently under evaluation, then the request is registered and the result is sent once E is fully reduced. If E is not evaluated at all, then p reduces E and returns the result when E is fully reduced. To implement recursion in this case the algorithm uses a stack of states. The current state S is pushed into the state stack, the algorithm enters

https://doi.org/10.1017/S095679689700275X Published online by Cambridge University Press

Normal Mode, evaluates E, returns the result to the requesting process, and returns to state S, which is at the top of stack.

Now let us return to the example presented in figure 1. Recall that p' is an evaluator process evaluating an expression E_2, and that $E_3 = (W\ 9)$ is a shared expression (subgraph) of E_2. For the purpose of this example assume that p' does indeed need the value of E_3, that E_3 is the only shared expression, and that the executive process of E_3 is located on another PE. When p' begins to evaluate E_3 it moves from Normal Mode to Sharing Mode, initializes the counter that counts the number of steps to zero, and stores the position of the evaluation stack and the task pool. There are two possibilities; if p' finished evaluating E_3 in less than C steps, it returns to Normal Mode. In such a case, p' performs the optimal action (duplicating work). If, on the other hand, the counter has reached $C+1$ steps and the evaluation has not ended, p' sends the executive process of E_3 a request to evaluate it. Process p' now begins to wait on its message queue for the result, during which p' may be suspended and other processes can run in its place. Once the result arrives, p' updates the result in the root E', restores the head of the evaluation stack and the task pool, and returns to Normal Mode. Note that in the worst case p' finishes the reduction in $C+1$ steps, in which case the cost incurred by p' is only a factor of two of the optimal hypothetical cost.

5 Appointing executing processes

To allow efficient realization of sharing between processes, the algorithm appoints each shared expression E an executive process P_E such that:

1. E belongs to the subgraph of P_E.
2. Each process p wishing to evaluate E is a descendant of P_E.

When a process needs the result of a shared expression E, it turns to the executive process of E. This scheme avoids duplication of work since all the requests for the value of E pass through the executive process P_E. The scheme has several important advantages:

- Natural use of identifiers: since E belongs to the subgraph of P_E, P_E can give E an identifier. This identifier is inherited to all the descendants of P_E. Any process p that needs E is a descendant of P_E, hence p can send a request message for evaluating E that includes only the identifier of E. In fact, the use of identifiers is mandatory since E could have gone through several transformations both in p and in P_E.
- Immediate access to P_E: since p is a descendant of P_E, p does not need to search the system for the executive process of E.

These are important advantages, as they considerably reduce the size and number of messages sent in the system. An additional advantage is a convenient synchronization between the life spans of the evaluator processes. Once a process ends its work, it is known that no more requests for nodes will arrive at that process and it can therefore safely terminate.

https://doi.org/10.1017/S095679689700275X Published online by Cambridge University Press

The remainder of this section suggests three schemes for appointing an executive process. The correctness of these schemes (that is, the fact that duplication of work is avoided) is proven in Ronen (1993). In general, there exists a trade-off between the simplicity and efficiency of a scheme, and the scheme's ability to select an executive process that is deeper (closer to the leaves) in the tree of processes. The higher the executive process is, the more danger there is of it becoming a serializing bottle neck; the deeper the executive process is, the more room there is for exploiting parallelism.

5.1 Creator-process scheme

This scheme is the simplest of the three schemes proposed. In this scheme a process that creates a new node (expression) marks itself as the executive process of that node. A process p that finishes its task, traverses the result that it is about to return and marks the parent process as the executive process of all the nodes that are marked with p as their executive process. This scheme does not add any significant overhead, as the result needs to be traversed anyway in order to return it. A process receiving a result of a shared expression from the executive process, does not change the marks. The important advantage of this scheme is in its low overhead. This scheme is especially suitable for systems that perform Unix-like fork spawning. That is, systems in which the child process receives parts of the graph when it needs them.

5.2 Sharing-process scheme

In this scheme a node is marked (the executive process of the node is determined) when it becomes shared. This scheme is more complex than the previous creator-process scheme, but it sometimes chooses an executive process that is deeper in the tree of processes.

Consider a process performing a substitution operation (β reduction) in the form $Apply((\lambda x.M), E_1, \ldots E_k, \ldots E_n) \xrightarrow{\beta} M'$, where argument E_k becomes shared. In this scheme, the process traverses E_k's subgraph and marks itself as the executive process of all non-marked nodes. As in the previous scheme, a process p that finishes its task, marks its parent process as the executive process of all the nodes marked with p as their executive process, and a process receiving a result does not change the markings.

Marking a shared argument is performed by a Depth-First Search (DFS) of the argument's graph. An interesting question that arises is whether there is a need to mark the whole subgraph. Consider the two expressions in figure 3. Informally, the only type of operation that can be performed on $e1$ is the evaluation of the whole expression. Therefore, it is sufficient to avoid the duplication of work on $e1$. In other words, only $e1$ should be marked. However, evaluating $e2$ does not prevent the possibility of the evaluation of one of its subexpressions. Preventing duplication in this case requires the marking of $e3$ and $e4$ as well.

Definition 5.1

A function F is said to be an *NF function* if F always returns an expression in

https://doi.org/10.1017/S095679689700275X Published online by Cambridge University Press

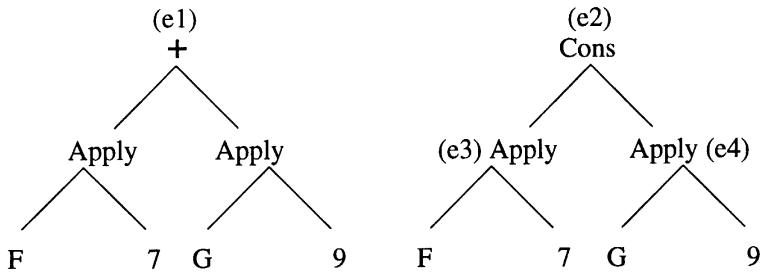

Fig. 3. Examples of an NF expression (e1) and a non NF expression (e2).

```
MarkArg( E ) =
    if E is not marked then
        Mark myself as the executive of E
        if E is not an NF-expression then
            ∀ subexpression e of E
                MarkArg( e )
```

Fig. 4. An outline of the marking algorithm of the sharing-process scheme.

Normal Form (NF). An expression E is said to be an *NF expression* if E applies an NF function (on some arguments).

If E is an NF expression there is no need to mark its subexpressions. Determining whether some expression is an NF expression should be performed at compile time. Much of this analysis can be handled via type checking. In particular, a function that returns a simple type is an NF function. The details of such an optimizing analysis is a topic that is left for future research.

An outline of the marking function of a shared argument in this scheme is given in figure 4.

5.3 Sending-process scheme

Most existing systems realize sharing by sending remote pointers to the shared expressions as part of a spawned task. The third scheme presented here adopts a similar approach.

When a process p spawns a task (expression) T it traverses T's graph: if it contains an unmarked shared expression E that is shared outside of T, then the process marks itself as the executive process of E. As in the sharing-process scheme, the marking procedure can stop once an NF expression is reached. As in the previous schemes, a process p that is about to return a result marks its parent process as the executive process of the nodes that have p marked as the executive process, and upon receiving a result the process does not change the markings. This scheme is most suitable for systems that spawn tasks by packing the whole task and sending it.

https://doi.org/10.1017/S095679689700275X Published online by Cambridge University Press

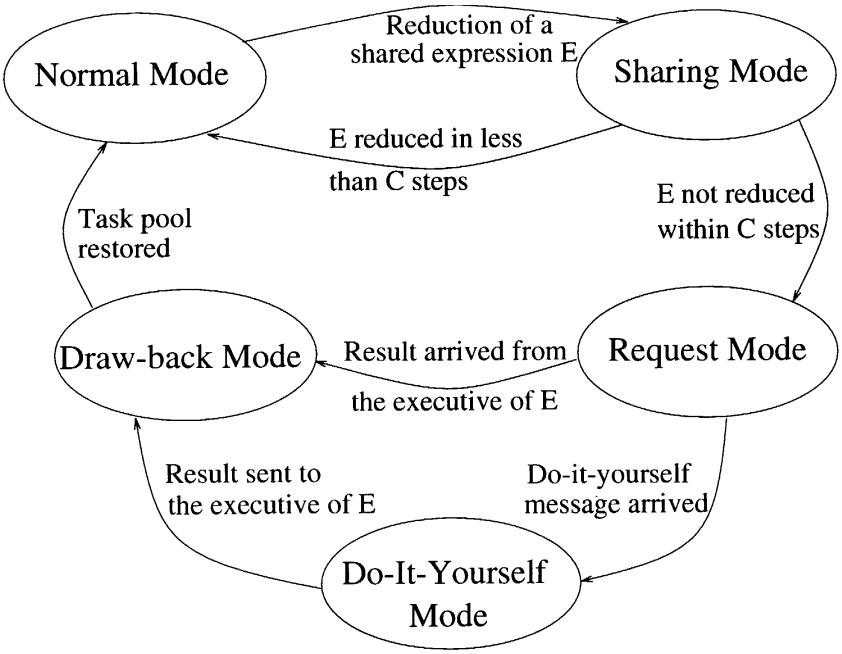

Fig. 5. A state-transition diagram of the CSM algorithm that handles do-it-yourself messages.

6 Do-it-yourself messages

The CSM algorithm, as described so far, avoids the duplication of work on computation-intensive shared expressions. Nevertheless, it may cause contention to occur in the executive processes. Such a situation may arise when one evaluator is the executive process of many shared expressions, and many evaluators request values of different expressions at the same time. If these shared expressions are not already reduced, then their executive process becomes a serializing-bottleneck, which may inhibit parallelism.

To alleviate such potential contention, the CSM algorithm may be modified to allow some of the requesting evaluators to reduce the shared expressions on their own. That is, when executive process P_E receives a request from evaluator p for the value of E, it checks the status of E. If E is evaluated, then the result is returned immediately. If E is under evaluation, then the result is returned when the evaluation ends. Otherwise, P_E replies with a *do-it-yourself* message and marks E as *blocked* ('under evaluation'). Then, p evaluates E and returns the result of this evaluation to P_E. A state-transition diagram describing the modified algorithm is depicted in figure 5.

This modification may involve a rather high overhead to ensure that work is not inadvertently duplicated. The following scenario explains how such duplication can occur: an evaluator process p encounters a shared expression E, performs C reduction steps on E, and then sends a request for the value of E to the executive process of E. According to the modification above, the executive process sends back

https://doi.org/10.1017/S095679689700275X Published online by Cambridge University Press

a do-it-yourself message to ask p to reduce E. Two reasonable actions may now be taken by p:

1. Continue the reduction from the place that p had stopped, that is, after C reduction steps.
2. Start the reduction of E from the root node E.

Both possibilities may lead to duplication of work if p had encountered shared nodes during the first C reduction steps of E. Let s be a shared node that was visited by p during the first C reduction steps. The visit to s did not activate the sharing procedure again (as noted in the description of the Sharing Mode above). If p chooses the first option and starts from where it had finished, then the information that s was a shared node would now be lost, causing p to reduce s by itself and thus duplicating the work on it. If p chooses the second option and starts the reduction from node E, then the work previously performed on reducing s is duplicated once node s is encountered again.

The duplication problem caused by this modification can be overcome, but the solutions seem to hinder the simplicity of the do-it-yourself mechanism and may lead to high overheads.

7 The CSM algorithm avoids deadlock

This section proves that the CSM algorithm is deadlock free. Assume by contradiction that the system does reach a deadlock state. Note that there are two cases in which a process in the system waits for another process:

1. When a process waits for a result of expression E that it spawned.
2. When a process waits for the result of a shared expression E, which is evaluated by the expression's executive process P_E.

Definition 7.1
Let $E_p \rightarrow E'_{p'}$ denote a situation where a process p, which is evaluating an expression E, is waiting for the evaluation of E' by p'.

Since $E_p \rightarrow E'_{p'}$ implies that E is computationally dependent on E', the following claim is an immediate outcome:

Claim 7.1
If there exists a cycle of the form $E1_{p1} \rightarrow E2_{p2} \rightarrow \cdots \rightarrow En_{pn} \rightarrow E1_q$, then the evaluation of $E1$ does not terminate.

Such a situation is not difficult to detect, and we therefore assume that such a cycle does not exist.

Definition 7.2
A dependency $E_p \rightarrow E'_{p'}$ is said to be a *last dependency* if there does not exist a dependency of the form $E'_{p'} \rightarrow E''_{p''}$ in the system.

Let $d = E_p \rightarrow E'_{p'}$ be a last dependency (because there are no cycles, there exists at least one such last dependency).

https://doi.org/10.1017/S095679689700275X Published online by Cambridge University Press

Claim 7.2
E' is not a task that was spawned by p.

Proof
If p did spawn E' then p' would have been created to evaluate E'. Since the evaluation E' does not depend on anything, it continues uninterrupted, which stands in contradiction to the deadlock state that the system is assumed to be in.

Hence, E' is a shared expression that is evaluated by its executive process p', and p is waiting for the result of this evaluation. Process p' is a ancestor of p in the tree of processes, and therefore cannot terminate before p terminates. Specifically, p' cannot terminate before the request to evaluate E' arrives at its message queue. Since each process has a single message queue on which it receives messages, and since a process can be deadlocked only when it waits on its queue, p' must have received the request and began to evaluate E'. Since d is a last dependency, the evaluation of E' continues uninterrupted, in contradiction to the system being in a deadlock state. □

8 Performance analysis of the CSM algorithm

This section analyses the performance of the CSM algorithm and compares it with the performance of a hypothetical optimal off-line algorithm. We note that an off-line algorithm has complete knowledge of the shape of the graph and its future evolution, while in practice this information is not available, and on-line algorithms (such as the CSM algorithm) have to be developed to operate without knowledge of the future. The ratio between these two algorithms varies according to the ability of the system to take advantage of the time in which processes wait for the evaluation of a shared expression. We shall analyse the two extreme cases: the case in which the system is unable to utilize this time at all (i.e. the PE remains idle while the process waits for the result) and the case in which the system fully utilizes this time (i.e. the PE always has a runnable process that can execute while the process is waiting for the result).

Let S be a set of evaluator processes that are sharing some expression E. Let $R(E)$ be the number of reduction steps required to reduce E to WHNF (without loss of generality, assume that this reduction cost is equal in all the evaluator processes of the system). Let $\#S$ be the number of evaluators in the set S, and let p be the number of PEs in the system. Recall that D is the cost of sending one message, and that $C = 4D$ is the amount of work duplicated at each evaluator that is not the executive process for each shared expression. Let $T_{opt}(E)$ be the overall time spent by the optimal algorithm for evaluating expression E, and let $T_{csm}(E)$ be the time spent by the CSM algorithm. All costs are in units of reduction steps.

First consider the case in which the system does not utilize the time the processes wait for the evaluation of the shared expression. That is, when a process waits for the evaluation of a shared expression, there is no other process that can take its place and run on the PE. The worst possible ratio between the performance of the CSM algorithm and the optimal algorithm occurs when $R(E) > C$ and $\#S \leq p$.

https://doi.org/10.1017/S095679689700275X Published online by Cambridge University Press

In this case, each evaluator in S that is not the executive process duplicates the evaluation of the shared expression E for C time units, and then requests the value of E from the executive evaluator. These $\#S-1$ requests for the value of the shared expression are handled serially by the executive evaluator, where the *parallel time* required by each request-reply operation is $2D$. Therefore, in this case the ratio between the CSM algorithm and the optimal algorithm is given by:

$$\frac{T_{csm}(E)}{T_{opt}(E)} \leq \frac{C + R(E) + (\#S - 1)2D}{R(E)} \leq \frac{p+3}{2}$$

We note that the CSM algorithm can be enhanced with an optimization that would overcome the difficulty presented in this case. That is, the algorithm could be changed so that when a value of a shared expression is required, the process would continue to duplicate work as long as there is no other runnable process on its PE. However, in practice, it may not be worthwhile to incur the overhead of this optimization scheme; for example this optimization keeps the PE artificially busy, which may prevent more useful processes migrating to it.

Now consider the case where there are enough runnable processes to fully utilize the system. That is, if a process waits for the result of a shared expression there is always another evaluator that takes its place and runs on the PE. In such a case, the system performs useful work while the process waits for the result, and this waiting time is not 'wasted'. Hence, the computation time is (approximately) equal to the total work performed by all the PEs divided by the number of PEs (p). Note that this approximation does not refer to scheduling anomalies, such as described in Burton and Rayward-Smith (1994). Let $Time(E_1, E_2, \ldots, E_n)$ be the time to compute all expressions $E_1 \ldots E_n$, and let $Work(E)$ denote the total work performed on expression E. $Work(E)$ is an additive function and therefore:

$$Time(E_1, E_2, \ldots, E_n) \approx \frac{Work(E_1 \ldots E_n)}{p} = \frac{\sum Work(E_i)}{p}$$

Recall that requesting the value of a shared expression from the executive process involves the sending and receiving of two messages, where the cost of each message is D. The evaluator sending the request 'wastes' D computation power while sending the request message, and similarly the executive process receiving the request. Hence, $2D$ of work are 'lost' during sending of the request message. Similarly for the reply message, where the executive process and the evaluator 'waste' another $2D$ of work. Therefore, the overall work for requesting a shared expression is $4D$. Note, however, that the *time* spent on handling these two messages is only $2D$, because the executive process and the evaluator process wait D time in *parallel* for the request message, and another D in parallel for the reply message.

Since at least one process needs to perform the evaluation, the optimal cost is bounded by $R(E) + (\#S - 1)\min(R(E), C)$. If $R(E) \leq C$, then both the optimal algorithm and the CSM algorithm have each of the evaluators in S reduce E independently, and do not incur any communication overheads. Hence, when $R(E) \leq C$, $Work_{opt} = Work_{csm} = \#SR(E)$. If $R(E) > C$, then the CSM algorithm duplicates the evaluation of E at each evaluator in S that is not the executive process for a duration of C reduction steps, and only then requests the result of the reduction

https://doi.org/10.1017/S095679689700275X Published online by Cambridge University Press

from the executive process. The ratio between the times of the CSM algorithm and the optimal algorithm in this case is therefore given by:

$$\frac{Time_{csm}(E)}{Time_{opt}(E)} \approx \frac{Work_{csm}(E)}{Work_{opt}(E)} \leq \frac{R(E) + (\#S - 1)2C}{R(E) + (\#S - 1)C} < 2.$$

Hence, the ratio between the overall time of the CSM algorithm and the optimal algorithm is bounded by:

$$\frac{\text{overall time of CSM}}{\text{overall time of OPT}} \approx \frac{Work_{csm}(E_1 \ldots E_n)}{Work_{opt}(E_1 \ldots E_n)} = \frac{\sum Work_{csm}(E_i)}{\sum Work_{opt}(E_i)} < 2.$$

9 Experimental measurements

This section presents performance results of the CSM algorithm. The execution platform is the MOSIX system (Barak *et al.*, 1993), a distributed operating system with a built-in dynamic process migration mechanism. The MOSIX system integrates a cluster of loosely-coupled, independent processors to a virtual, single machine UNIX environment. The specific configuration used includes eight NS32532 microprocessor based computers, each with its own local memory and communication devices. These computers are arranged in two identical enclosures, each with four processors that communicate via a shared VME bus. The two enclosures are connected by a ProNET-80, an 80 Mbits/second token-ring LAN.

Our experiments were performed on an implementation of a λ-calculus evaluator, which is based on compiled graph reduction techniques (Peyton Jones, 1987), but without many of the optimizations. The evaluator accepts a functional program written in the usual λ-calculus notation (enhanced with named functions), and produces target code in C. The evaluator processes are realized as Unix processes, that is, spawning a task to reduce some subexpression is implemented by forking a Unix process. This implementation takes advantage of the automatic load-balancing of the MOSIX system.

The CG algorithm (Aharoni, Feitelson and Barak, 1992) is used to decide *when* to spawn new tasks for parallel execution. This algorithm balances the amount of local computation with the cost of distributing the tasks. That is, the algorithm ensures that for every parallel task spawned, an amount of work that equals the cost of the spawn is performed locally. Three different versions of the CG algorithm were used, where each version applies a different sharing-management scheme. The first is a CG algorithm (Dup) that always duplicates work, that is, sharing is never exploited and thus a shared expression may be evaluated several times. The second is a CG algorithm (Exp) that always exploits sharing, that is, work is never duplicated. The third is a CG algorithm (CSM) that uses the CSM algorithm as its sharing-management scheme.

Table 1 compares the performance results of the above distributed implementations (Dup, Exp, and CSM), and a serial algorithm (Ser), which performs ordinary graph reduction on a single PE. The four implementations are applied to the functions Bin(), Massive() and Useless(). The table gives the execution time (in seconds) of these implementations, as well as their speedup in comparison with CSM (the

https://doi.org/10.1017/S095679689700275X Published online by Cambridge University Press

Table 1. *Performance of the CSM algorithm*

	Execution times (sec)				Speedup ratios		
Function	Ser	Dup	Exp	CSM	Ser/CSM	Dup/CSM	Exp/CSM
Bin	1310	246	246	246	5.32	1	1
Massive	1457	452	362	323	4.51	1.4	1.12
Useless	2346	410	510	415	5.65	0.99	1.23

CG algorithm that uses the CSM sharing-management scheme) when running on eight PEs.

The function Bin() performs a purely heavy parallel computation by expanding a full balanced binary tree. This function is meant to test the performance of the CSM algorithm when there is absolutely no sharing. The function Massive() performs a parallel computation with many computation-intensive shared expressions (Bin(14)). This function tests the performance of the system when there is much useful sharing. The function Useless() performs a parallel computation with many computationally small shared expressions. This function tests the performance of the system when all the shared expressions are computationally small and their sharing would not be exploited by an optimal algorithm that knows their size.

The experimental results reinforce the contribution of the CSM algorithm to distributed graph-reduction system. The algorithm exploits the sharing of computation-intensive expressions while avoiding the communication overhead of realizing the sharing of computationally small expressions. This is shown in the experiments, where the CSM algorithm outperforms both a parallel algorithm that always exploits sharing and a parallel algorithm that always duplicates the evaluation of shared expressions. The low overhead (up to 1%) maintain the system's performance in almost every case.

10 Applying further optimizations to the CSM algorithm

The CSM algorithm is designed to work on a wide range of graph-reduction-based models that support speculation. This section discusses the possibility of incorporating the CSM algorithm in graph-reduction models that apply various optimization techniques to the classic graph-reduction algorithm.

The substitution operation (β-reduction) is a frequent and relatively expensive operation, and is therefore a target for many optimizations. The CSM algorithm does not use this operation (unless the sharing-process scheme described in section 5.2 is used) and therefore, these optimizations should not interfere in its work.

Many graph-reduction models optimize the structures that represent the graph. The CSM algorithm does not traverse the graph (unless the sharing-process or the sending-process schemes described in sections 5.2 and 5.3 are used) and therefore

https://doi.org/10.1017/S095679689700275X Published online by Cambridge University Press

these optimizations should not interfere in its operation. Nevertheless, the algorithm does depend on the existence of the notion of a node. That is, the ability to identify and locate an expression (and its transformation) via an id. Without this assumption it is rather difficult to exploit sharing.

The use of compilation techniques has been a great contribution to the efficient execution of graph reduction. The CSM algorithm can be incorporated in the compiled code without too much difficulty.

Finally, we note that the CSM algorithm takes advantage of the DFS nature of the evaluation in developing efficient and simple mechanisms. The algorithm might not work as efficiently in models that do not preserve this property.

11 Conclusions and future research

We have described an efficient run-time algorithm that is able to manage the sharing of subexpressions across a loosely-coupled distributed system. The CSM algorithm that was presented takes advantage of ideas from the field of competitive algorithms to optimize the handling of shared expressions. The algorithm exploits the sharing of computation-intensive expressions, but it duplicates the evaluation of computationally small expressions. This desired goal is achieved without having any *a priori* information about the amount of computation that is required for evaluating the shared expression. We have shown that the CSM algorithm competes well with the hypothetical optimal algorithm, which does have all the information that is needed about the future evolution of the program. The algorithm does not require any programmer intervention, is easy to implement, and has low overhead.

One problem that has been left open is the question of *how* to spawn tasks, that is, how to send subexpressions (subgraph) for evaluation from one PE to another. One alternative is to copy the whole spawned subgraph into the address space of the destination PE. Another alternative is to send a remote pointer pointing at the graph in the address space of the source PE. Although this paper assumes the former scheme, in fact both schemes can be incorporated into the CSM algorithm, while still maintaining its deadlock-free nature. Note that it is not clear which scheme is more cost effective. On the one hand, the former scheme might waste work on copying parts of the spawned graph that are not needed in its evaluation, while the latter remote-pointer scheme suffers from a considerable overhead in the amount of communication between the PEs. The solution probably lies in a scheme that somehow balances between these two schemes. The solution to this problem might also be based on competitive principles. In the mean time, until this problem is properly addressed, one of the referees of this paper suggested a scheme whereby only physically small graphs would be copied, while a remote pointer would be sent for large graphs.

We assume in this paper that the result of any spawned expression can be returned in a single, fixed-sized message. This assumption does not hold in cases where the result includes a large graph. The handling of large results is in fact a problem in any distributed-memory systems. Future work will include investigating the possibility

of alleviating this problem by applying type checking schemes that only allow the spawning of subgraphs that are guaranteed to return small subgraphs as results.

The CSM algorithm assumes that if the speculative computation of the initial C evaluation steps on some shared expression do not fully evaluate this expression, then the expression's graph should revert to its original form. Section 4 suggests that the DFS nature of the evaluation can be used to perform this operation efficiently. Nevertheless, future work should include a closer inversigation of the details of the speculative-computation support that is required by the algorithm.

Further future work will also include enhancing the CSM algorithm with information obtained by various compile-time analysis methods such as sharing analysis, granularity analysis, and strictness analysis. And in addition, investigating the statistical distribution of the amount of work involved in the evaluation of functional expressions. Finding this distribution will enable to change the value of C and get an improved (statistical) ratio between CSM and the hypothetical optimal algorithm.

Acknowledgements

The authors are grateful to Inbal Achiman, Moshe Ben Ezra, and Paula Ross for many helpful comments. Thanks are due to the referees for their efforts in reviewing the paper.

References

Aharoni, G., Feitelson, D. G. and Barak, A. (1992) A run-time algorithm for managing the granularity of parallel functional programs. *J. Functional Programming*, **2**(4): 387–405.

Barak, A., Guday, S. and Wheeler, R. (1993) *The MOSIX Distributed Operating System, Load Balancing for UNIX. Lecture Notes in Computer Science 672.* Springer-Verlag.

Burton, F. W, and Rayward-Smith, V. J. (1994) Worst case scheduling for parallel functional programs. *J. Functional Programming*, **4**(1): 65–75.

Darlington, J. and Reeve, M. J. (1981) ALICE – a multiprocessor reduction machine for the parallel evaluation of applicative languages. *Proc. Conf. on Functional Programming Languages and Computer Architecture*, Portsmouth, NH, pp. 66–67.

Field, A. J. and Harrison, P. G. (1988) *Functional Programming.* Addison-Wesley.

Flanagan, C. and Nikhil, R. S. (1996) pHluid: the design of a parallel functional language implementation. *Proc. Int. Conf. on Functional Programming*, Philadelphia, PA, pp. 169–179.

Goldberg, B. (1988) Multiprocessor execution of functional programming. *Int. J. Parallel Programming*, **17**(5): 425–472.

Kelly, P. J. (1989) *Functional Programming for Loosely-coupled Multiprocessors.* Pitman.

Kesseler, M. (1996) The implementation of functional languages on parallel machines with distributed memory. *PhD thesis*, Computing Science Institute, Faculteit Wiskunde en Informatica, Katholieke Universitaet Nijmegen, The Netherlands.

Mattson, J. S. (1993) An effective speculative evaluation technique for parallel supercombinator graph reduction. *PhD thesis*, Department of Computer Science and Engineering, University of California, San Diego.

Partridge, A. S. (1991) Speculative evaluation in parallel implementations of lazy functional languages. *PhD thesis*, University of Tasmania.

Peyton Jones, S. L. (1987) *The Implementation of Functional Programming Languages*. Prentice-Hall.

Peyton Jones, S. L., Clack, C., Salkild, J. and Hardie, M. (1990) GRIP – a high-performance architecture for parallel graph reduction. In Fountain, T. J. and Shute, M. J. (eds.), *Multiprocessor Computer Architectures*, North-Holland, The Netherlands, pp. 101–119.

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

Raber, M., Remmel, T., Maurer, D., Müller, F., Oberhauser, H.-G. and Wilhelm, R. (1987) A Concept for a Parallel G-Machine. *Technical Report 06/1987*, Universität des Saarlandes, Saarbrücken, Germany.

Ronen, A. (1993) A competitive approach for managing sharing in the distributed execution of functional programs. *Master's thesis*, Computer Science Department, The Hebrew University of Jerusalem, Israel. (In Hebrew.)

Sleator, D. and Tarjan, R. (1985) Amortized efficiency of list update and paging rules. *Commun. ACM*, **28**(2): 202–208.

Trinder, P., Hammond, K., Mattson, J. S., Partridge, A. S. and Peyton Jones, S. L. (1996) GUM: a portable parallel implementation of Haskell. *Proc. Conf. on Programming Languages Design and Implementation*, Philadelphia, PA.