J. Functional Programming **11** (4): 359–393, July 2001. Printed in the United Kingdom

Pipelined functional tree accesses and updates: scheduling, synchronization, caching and coherence

ANDREW J. BENNETT*, PAUL H. J. KELLY
Department of Computing, Imperial College,
180 Queen's Gate, London SW7 2BZ, UK
(*e-mail:* p.kelly@doc.ic.ac.uk)

ROSS A. PATERSON
Department of Computer Science, City University, London, UK

Abstract

This paper is an exploration of the parallel graph reduction approach to parallel functional programming, illustrated by a particular example: pipelined, dynamically-scheduled implementation of search, updates and read-modify-write transactions on an in-store binary search tree. We use program transformation, execution-driven simulation and analytical modelling to expose the maximum potential parallelism, the minimum communication and synchronisation overheads, and to control the overall space requirement. We begin with a lazy functional program specifying a series of transactions on a binary tree, each involving several searches and updates, in a side-effect-free fashion. Transformation of the source code produces a formulation of the program with greater locality and larger grain size than can be achieved using naive parallelization methods, and we show that, with care, these tasks can be scheduled effectively. Even with a workload using random keys, significant spatial locality is found, and we evaluate a modified cache coherency protocol which avoids false sharing so that large cache lines can be used to minimise the number of messages required. As expected with a pipeline, the application should reach a steady state as soon as the first transaction is completed. However, if the network latency is too large, the rate of completion lags behind the rate at which work is admitted, and internal queues grow without bound. We determine the conditions under which this occurs, and show how it can be avoided while maximising speedup.

Capsule Review

The functional programming community has a long tradition of using program transformation and analytical models to improve, and explore the characteristics of, sequential programs. This paper is novel in applying these techniques, along with simulation, to improve and characterise, the *parallel behaviour* of a functional program. The program has interesting parallel behaviour, namely a high degree of synchronisation, data-dependent sharing of intermediate results and a high degree of spatial locality. All three techniques are used to improve parallelism by increasing locality and maximal parallelism, and reducing the communication and synchronisation overheads. An analysis of the key pipeline bottleneck

* Currently with Micromuse PLC, London, UK.

in the program allows maximum throughput to be optimised according to the latency and computation speed of the underlying parallel platform.

1 Introduction

This paper is an exploration of a particular approach to parallel functional programming, in which the dynamically-scheduled, shared-memory parallel functional approach is applied to a simple application involving updates to binary search trees.

The application can be considered a simplified, in-store formulation of on-line transaction processing on a database (although there are good alternative techniques available). We selected it for this study because it illuminates more general performance issues in parallel software design, which we expose in this paper:

- Its high rate of synchronization reflects a fundamental requirement of the algorithm, unlike many benchmarks commonly used for parallel graph reduction.
- It displays interesting, data-dependent sharing of intermediate results.
- It displays considerable spatial locality, despite lacking a statically-predictable access pattern.

This case study is a simple example that exposes a number of interesting issues. These issues, we argue, are very likely to arise in more complicated programs, in less tractable form. It follows from the paper's exploratory nature that our conclusions are indicative rather than conclusive.

1.1 Background and related work

A functional formulation of transaction processing was introduced by Trinder *et al.* (Argo et al., 1987; Trinder, 1989), and the results of simulation and execution on the GRIP multiprocessor are reported in Akerholt *et al.* (1993). Our approach to parallel functional programming is based on parallel graph reduction (Peyton Jones, 1989); the most thorough studies of parallel graph reduction performance have arisen from the GRIP project (Peyton Jones *et al.*, 1987), although the most successful implementations have relied on general-purpose shared-memory multiprocessors (e.g. the $\langle v, G\rangle$-machine) (Augustsson and Johnsson, 1989). Closely related work has been done in the context of futures in MultiLisp (Halstead, 1984). Caching and locality of reference in a sequential graph reducer has been studied by several groups (Koopman *et al.*, 1992). The efficient implementation of the shared graph has been studied by the present authors (Bennett and Kelly, 1993), and based on this analysis we have proposed a caching scheme – the Two-Level Ownership (TLO) protocol – specially suited to parallel graph reduction (Bennett and Kelly, 1994; Bennett and Kelly, 1997). We study the effectiveness of both this and conventional cache coherence protocols in supporting functional transaction processing later in this paper.

We are particularly concerned with modern multicomputers, which have relatively

https://doi.org/10.1017/S0956796801003793 Published online by Cambridge University Press

high latency and high bandwidth interconnection networks – these characteristics greatly influence the behaviour and speedup of programs.

There is a considerable literature on algebraic transformation of functional programs to improve their suitability for parallel execution (e.g. (Boyle *et al*., 1987; Kelly, 1989; Harrison, 1992; Darlington *et al*., 1993)). The transformations presented here are unusual in being targeted to parallel graph reduction, and concern reducing control dependence. They are related to the work of Bird and others on using 'circular' programs to eliminate multiple traversals of data (Bird, 1984).

Our work is based on Trinder's technique of using multiple versions of trees, and using the synchronization mechanism of parallel graph reduction for record locking. However the mechanism used to achieve concurrent evaluation in the presence of total transactions is quite different: we have used transformation whereas Trinder proposes various approaches based on implementations of the conditional operator (see section 3). The most successful of these, the 'Friedman and Wise' *fwif* approach tends to produce many small tasks. Recognizing that current and future multiprocessors have high latency and high bandwidth interconnection networks, we apply various transformations to control task granularity and locality. Larger grain tasks raise a number of interesting scheduling issues.

In later work, the Glasgow group conducted extensive detailed applications case studies (Loidl and Trinder, 1998; Loidl *et al*., 1998). To support this activity, they developed simulation tools similar to ours, and visualization techniques (Hammond et al., 1995). In the light of their experience, they have developed techniques for abstract but explicit control over how parallelism is exploited (Trinder *et al*., 1998). Loidl and Hammond (1996) explicitly address the idea of 'bulk fetching', analogous to our use of large cache lines.

1.2 Main contributions

1. We provide an instructive example of systematic algebraic transformation to improve the run-time characteristics of a dynamic, irregular multi-threaded application (sections 3 and 4)
2. We expose and analyse the need to control CPU scheduling to avoid thread migration and improve locality (section 6.2)
3. We evaluate the application's performance using a conventional Distributed Shared Memory (DSM) implementation, showing the presence of spatial locality as predicted by our analytical model, offset by contention for cache line ownership due to false sharing (section 8)
4. We demonstrate that a modified DSM coherency protocol, which exploits the parallel graph reduction memory access discipline, eliminates invalidations so that false sharing does not lead to contention (section 8.3)
5. We identify a serious problem with the application's space behaviour, which occurs under some combinations of workload and system performance characteristics. This could be solved naively by restricting the rate of initiation of tasks. We show (section 8.4) that by balancing the pipeline, much higher performance can be achieved while retaining reasonable space requirements.

1.3 Structure of the remainder of the paper

The remainder of the paper is structured as follows: section 2 introduces the binary search tree access and update operations, and presents the structure of a transaction. section 3 concerns parallelizing the execution of a sequence of transactions. Inter-transaction parallelism can be exploited if transactions are expected to succeed, and we show how an optimistic implementation can be derived formally. A further transformation, combining two similar passes over a data structure into one is shown in section 4. An overview of parallel graph reduction and the simulation scheme used for our experiments is given in section 5. Results of simulations of an ideal shared-memory are used to illustrate the operation of the new form of the program in section 6. Multicache implementations of shared-memory are reviewed in section 7, and simulation results and their corresponding analytical models are presented in section 8. Finally, we present our conclusions and give some pointers to future work.

2 A simple formulation of transaction processing

We begin with a direct functional formulation of transaction processing. Similar material is covered in greater detail in Trinder (1989).

The usual implementation of updates in a conventional database system alters the data structures in place. At first sight it would appear that a functional version must generate a new copy of the entire database for each update. However, it is well known that simple databases may be structured as trees, and the update need only generate new versions of nodes on a path down the tree to the addressed record (this representation is also well suited to write-once media). For ease of illustration, we shall assume a simple database, comprising a collection of records indexed by a single key, and that the whole database resides in memory, or that the language supports persistent data structures.

More specifically, we define a database as a tree of the following kind:

$$\textbf{data}\ DB\ \alpha\ \beta \triangleq Leaf\ \alpha\ \beta \mid Node\ (DB\ \alpha\ \beta)\ \alpha\ (DB\ \alpha\ \beta)$$

A database item is either a leaf, containing a key and associated data, or an index node containing a signpost key and two subtrees. In a properly constructed tree, the key in an index node is greater than or equal to all the keys in the left subtree and less than all those in the right subtree.

The function *lookup* returns the datum associated with a key, if present, or fails:

$$\textbf{data}\ Maybe\ \alpha \triangleq Yes\ \alpha \mid No$$

$$lookup\ :\alpha \to DB\ \alpha\ \beta \to Maybe\ \beta$$
$$lookup\ x\ (Leaf\ k\ v) \triangleq \textbf{if}\ k = x\ \textbf{then}\ Yes\ v\ \textbf{else}\ No$$
$$lookup\ x\ (Node\ l\ k\ r) \triangleq lookup\ x\ (\textbf{if}\ x \leqslant k\ \textbf{then}\ l\ \textbf{else}\ r)$$

The following function assigns the datum associated with a key, if present:

$$assign\ :\alpha \to \beta \to DB\ \alpha\ \beta \to DB\ \alpha\ \beta$$

https://doi.org/10.1017/S0956796801003793 Published online by Cambridge University Press

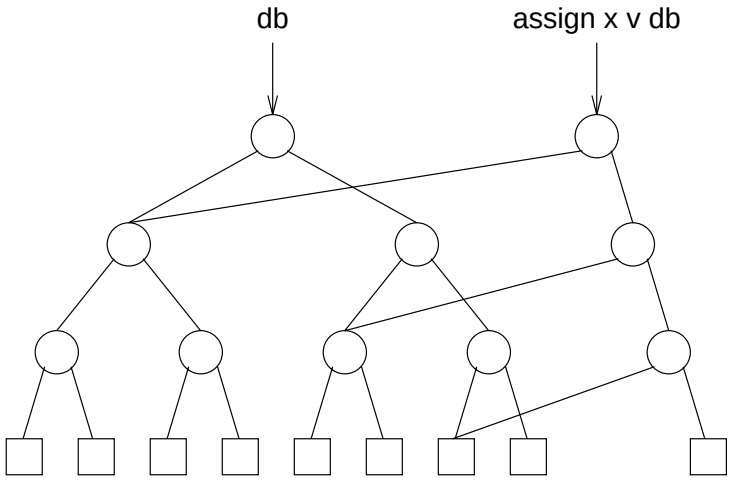

Fig. 1. New nodes created by *assign*.

$$assign\ x\ v'\ (Leaf\ k\ v) \triangleq Leaf\ k\ (\textbf{if}\ k = x\ \textbf{then}\ v'\ \textbf{else}\ v)$$
$$assign\ x\ v'\ (Node\ l\ k\ r) \triangleq$$
$$\quad \textbf{if}\ x \leqslant k$$
$$\quad \textbf{then}\ Node\ (assign\ x\ v'\ l)\ k\ r$$
$$\quad \textbf{else}\ Node\ l\ k\ (assign\ x\ v'\ r)$$

As noted above, only nodes on the path down to the addressed leaf are replaced (see figure 1).

We do not present results in this paper for updates that change the shape of the tree, such as insertions and deletions. As is explained in section 9, the application's behaviour would be complicated somewhat but not substantially changed.

A great deal of parallelism is available in such a database. A *lookup* and an *assign* applied to the same database may proceed independently in parallel, since the *assign* returns a new tree without altering the old one. Another source of parallelism is the non-strict semantics of the functional formulation, which allows a constructed value to be inspected before its components have been evaluated. Here, an instance of *assign* is able to construct each index node without waiting for the recursive call. If the result is the input to another call of *assign*, the new index node may be given to a thread executing the following call while the current thread processes a child node. That is, successive calls to *assign* may run concurrently in a pipeline. Once their paths diverge in the tree, they will be completely independent.

On the other hand, if a thread inspects the value of a sub-expression that is being evaluated by another thread, it must block until the value is available. Hence the flow of data within the program provides the only locking mechanism we shall need. In the current case, a thread executing *assign* is blocked from accessing a node until the previous instance has updated the node, and moved on to one of its child nodes. The effect is equivalent to a tree locking scheme, with a tree node being held temporarily and released as soon as one of its children is held.

A more realistic application might group a number of *lookup* and *update* oper-

https://doi.org/10.1017/S0956796801003793 Published online by Cambridge University Press

ations as a *transaction*, so that the updates are taken as a whole – either all take effect, or in the event of some failure the database is unchanged. For example, let t denote a data structure describing a transaction, containing distinct keys $x_1, \ldots, x_5$. Let the corresponding values be $v_1, \ldots, v_5$. If all keys are present and a condition c defined in terms of t and the v_i is satisfied, we wish to replace each v_i with a new value e_i defined in terms of t and $v_1, \ldots, v_5$. Otherwise we wish to return the original database. Thus a simple function to apply such a transaction to a database is:

```
apply : Trans → DB Key Value → DB Key Value
apply t db0 ≙
    let v1 ≙ lookup x1 db0 in
    ...
    let v5 ≙ lookup x5 db0 in
    let all-ok ≙ ok v1 ∧ ... ∧ ok v5 in
    if all-ok ∧ c
    then
        let db1 ≙ assign x1 e1 db0 in
        ...
        let db5 ≙ assign x5 e5 db4 in
        db5
    else db
```

where the function *ok* tests whether a *lookup* has succeeded:

```
ok : Maybe α → Bool
ok (Yes x) ≙ True
ok No ≙ False
```

This sort of global rollback is neatly supported by a functional formulation (Trinder, 1989): the program may return either the old root node or a new one reflecting all the updates.

3 Reducing synchronization

The program of the previous section has much parallelism within a transaction:

- The various *lookup* operations may be executed in parallel.
- Assuming that transactions usually succeed, we may speculatively execute the *assign* operations in parallel with the *lookup*s. Moreover, the *assign* operations exhibit the pipeline parallelism discussed above.

However, in a sequence of transactions, no parallelism between transactions is possible: the top-level **if** forces a synchronization between transactions, as the next transaction cannot begin until the appropriate branch is selected. This is wasteful, since the two possible versions of the output database differ very little, and the differences are all in the leaf records, not the index nodes. Recognizing this, Trinder (1989) and Akerholt *et al.* (1993) proposed that the **if** in the above program

https://doi.org/10.1017/S0956796801003793 Published online by Cambridge University Press

be replaced with a less-strict conditional proposed by Friedman and Wise (1978). The idea is to evaluate the branches concurrently with the condition; if both branches have the same top-level structure, it may be returned even if the condition is as yet unknown, while execution continues on the subtrees. The action of this function might be described by parallel application of the usual conditional rules

$$\mathit{fwif}\ \mathit{True}\ x\ y \stackrel{\triangle}{=} x$$
$$\mathit{fwif}\ \mathit{False}\ x\ y \stackrel{\triangle}{=} y$$

together with an extra rule for each data constructor:

$$\mathit{fwif}\ b\ (C\ x_1 \ldots x_n)\ (C\ y_1 \ldots y_n) = C\ (\mathit{fwif}\ b\ x_1\ y_1) \ldots (\mathit{fwif}\ b\ x_n \ldots y_n)$$

This requires at least two threads, one to evaluate the condition and another to evaluate the two branches to weak head normal form. If the former thread terminates first, then *fwif* behaves like a normal conditional. If the latter terminates first, and the two branches have the same shape, the structure may be returned while execution continues on the subtrees.

The effect is to push the conditional down into its branches, hence reducing synchronization. We prefer to achieve a similar effect using a source-level transformation, generating a new program that exhibits better parallel behaviour, but is equivalent in the specified context. The transformational approach can produce programs in which the grain of parallelism is quite large. The essence of the transformation to remove the synchronization entailed by **if** is a pair of observations

$$\textbf{if}\ b\ \textbf{then}\ e\ \textbf{else}\ e \sqsubseteq e$$

$$\textbf{if}\ b\ \textbf{then}\ C\ e_1 \ldots e_n\ \textbf{else}\ C\ e'_1 \ldots e'_n \sqsubseteq C\ (\textbf{if}\ b\ \textbf{then}\ e_1\ \textbf{else}\ e'_1) \ldots (\textbf{if}\ b\ \textbf{then}\ e_n\ \textbf{else}\ e'_n)$$

The two sides of each relation are equal if b is defined, but differ if b diverges – the right-hand sides can produce results without waiting for the value of b. The difference on partially defined inputs reflects increased laziness in a sequential setting, increased concurrency in a parallel one.

In the current case, we wish to apply such a transformation to a sub-expression of the form

$$\textbf{if}\ b\ \textbf{then}\ \mathit{assign}\ x\ v\ d\ \textbf{else}\ d$$

That is, we require a function

$$\mathit{maybe\text{-}assign} : \alpha \to \mathit{Maybe}\ \beta \to \mathit{DB}\ \alpha\ \beta \to \mathit{DB}\ \alpha\ \beta$$

such that for all fully defined x and d,

$$\textbf{if}\ b\ \textbf{then}\ \mathit{assign}\ x\ v\ d\ \textbf{else}\ d \sqsubseteq \mathit{maybe\text{-}assign}\ x\ (\textbf{if}\ b\ \textbf{then}\ \mathit{Yes}\ v\ \textbf{else}\ \mathit{No})\ d \qquad (*)$$

The function *maybe-assign* always updates the tree, and can therefore proceed down the tree (thereby freeing the upper levels for other processes). It need not wait to know whether the value in the leaf must be changed until it reaches that leaf.

https://doi.org/10.1017/S0956796801003793 Published online by Cambridge University Press

Assuming for the moment the existence of such a function, we can transform our transaction processor to the equivalent

$$
\begin{array}{l}
\mathit{decide} : \mathit{Trans} \to \mathit{Maybe}\ \mathit{Value} \to \ldots \to \mathit{Maybe}\ \mathit{Value} \to \\
\quad (\mathit{Maybe}\ \mathit{Value} \times \ldots \times \mathit{Maybe}\ \mathit{Value}) \\
\mathit{decide}\ t\ u_1 \ldots u_5 = \\
\quad \textbf{let}\ \mathit{all\text{-}ok} \stackrel{\triangle}{=} \mathit{ok}\ u_1 \wedge \ldots \wedge \mathit{ok}\ u_5\ \textbf{in} \\
\qquad \textbf{if}\ \mathit{all\text{-}ok} \wedge c \\
\qquad \textbf{then}\ (\mathit{Yes}\ e_1, \ldots, \mathit{Yes}\ e_5) \\
\qquad \textbf{else}\ (\mathit{No}, \ldots, \mathit{No})
\end{array}
$$

$$
\begin{array}{l}
\mathit{apply}\ t\ db_0 \stackrel{\triangle}{=} \\
\quad \textbf{let}\ u_1 \stackrel{\triangle}{=} \mathit{lookup}\ x_1\ db_0\ \textbf{in} \\
\quad \ldots \\
\quad \textbf{let}\ u_5 \stackrel{\triangle}{=} \mathit{lookup}\ x_5\ db_0\ \textbf{in} \\
\quad \textbf{let}\ (u'_1, \ldots, u'_5) \stackrel{\triangle}{=} \mathit{decide}\ t\ u_1 \ldots u_5\ \textbf{in} \\
\quad \textbf{let}\ db_1 \stackrel{\triangle}{=} \mathit{maybe\text{-}assign}\ x_1\ u'_1\ db_0\ \textbf{in} \\
\quad \ldots \\
\quad \textbf{let}\ db_5 \stackrel{\triangle}{=} \mathit{maybe\text{-}assign}\ x_5\ u'_5\ db_4\ \textbf{in} \\
\quad db_5
\end{array}
$$

The two programs are equivalent on fully defined inputs. However, in the revised version the five instances of *maybe-assign* can be pipelined, and will proceed in parallel once their paths in the tree diverge. Subsequent *apply*s may also run concurrently, blocking only if they refer to one of these keys.

It remains to construct a definition of the function *maybe-assign*. The usual procedure in such cases is to attempt to prove the desired relation (*) by induction over the data structure. In the process, the appropriate definition of the function will often emerge, as happens in this case.

To simplify the notation a little, we introduce an infix function

$$
\begin{array}{l}
\mathsf{or} : \mathit{Maybe}\ \alpha \to \alpha \to \alpha \\
\mathit{Yes}\ x\ \mathsf{or}\ y \stackrel{\triangle}{=} x \\
\mathit{No}\ \mathsf{or}\ y \stackrel{\triangle}{=} y
\end{array}
$$

The *Leaf* case of equation (*) is

$$
\begin{array}{ll}
 & \textbf{if}\ b\ \textbf{then}\ \mathit{assign}\ x\ u\ (\mathit{Leaf}\ k\ v)\ \textbf{else}\ \mathit{Leaf}\ k\ v \\
= & \quad \{\ \text{definition of } \mathit{assign}\ \} \\
 & \textbf{if}\ b\ \textbf{then}\ \mathit{Leaf}\ k\ (\textbf{if}\ k = x\ \textbf{then}\ u\ \textbf{else}\ v)\ \textbf{else}\ \mathit{Leaf}\ k\ v \\
\sqsubseteq & \quad \{\ \text{re-arranging } \textbf{if}\text{s}\ \} \\
 & \mathit{Leaf}\ k\ (\textbf{if}\ b \wedge k = x\ \textbf{then}\ u\ \textbf{else}\ v) \\
= & \quad \{\ \text{re-arranging } \textbf{if}\text{s}\ \} \\
 & \mathit{Leaf}\ k\ (\textbf{if}\ k = x\ \textbf{then}\ (\textbf{if}\ b\ \textbf{then}\ u\ \textbf{else}\ v)\ \textbf{else}\ v) \\
= & \quad \{\ \text{definition of } \mathsf{or}\ \} \\
 & \mathit{Leaf}\ k\ (\textbf{if}\ k = x\ \textbf{then}\ ((\textbf{if}\ b\ \textbf{then}\ \mathit{Yes}\ u\ \textbf{else}\ \mathit{No})\ \mathsf{or}\ v)\ \textbf{else}\ v) \\
= & \quad \{\ \text{synthesized } \mathit{Leaf} \text{ case of } \mathit{maybe\text{-}assign}\ \} \\
 & \mathit{maybe\text{-}assign}\ x\ (\textbf{if}\ b\ \textbf{then}\ \mathit{Yes}\ u\ \textbf{else}\ \mathit{No})\ (\mathit{Leaf}\ k\ v)
\end{array}
$$

https://doi.org/10.1017/S0956796801003793 Published online by Cambridge University Press

To perform the last step, we had to synthesize the *Leaf* case of the definition of *maybe-assign*:

$$\mathit{maybe\text{-}assign}\ x\ u\ (\mathit{Leaf}\ k\ v) \triangleq \mathit{Leaf}\ k\ (\textbf{if}\ k = x\ \textbf{then}\ (u\ \mathsf{or}\ v)\ \textbf{else}\ v)$$

In attempting to prove the *Node* case of relation (*), we may assume induction hypotheses, namely that equation (*) has been demonstrated for l and r.

$$\begin{array}{ll}
 & \textbf{if}\ b\ \textbf{then}\ \mathit{assign}\ x\ v\ (\mathit{Node}\ l\ k\ r)\ \textbf{else}\ \mathit{Node}\ l\ k\ r \\
= & \quad \{\ \text{definition of } \mathit{assign}\ \} \\
 & \textbf{if}\ b \\
 & \textbf{then}\ (\textbf{if}\ x \leqslant k\ \textbf{then}\ \mathit{Node}\ (\mathit{assign}\ x\ v\ l)\ l\ r \\
 & \textbf{else}\ \mathit{Node}\ l\ k\ (\mathit{assign}\ x\ v\ r))\ \textbf{else}\ \mathit{Node}\ l\ k\ r \\
\sqsubseteq & \quad \{\ \text{re-arranging } \textbf{if}\text{s}\ \} \\
 & \textbf{if}\ x \leqslant k \\
 & \textbf{then}\ \mathit{Node}\ (\textbf{if}\ b\ \textbf{then}\ \mathit{assign}\ x\ v\ l\ \textbf{else}\ l)\ k\ r \\
 & \textbf{else}\ \mathit{Node}\ l\ k\ (\textbf{if}\ b\ \textbf{then}\ \mathit{assign}\ x\ v\ r\ \textbf{else}\ r) \\
= & \quad \{\ \text{induction hypotheses for } l \text{ and } r\ \} \\
 & \textbf{if}\ x \leqslant k \\
 & \textbf{then}\ \mathit{Node}\ (\mathit{maybe\text{-}assign}\ x\ (\textbf{if}\ b\ \textbf{then}\ \mathit{Yes}\ v\ \textbf{else}\ \mathit{No})\ l)\ k\ r \\
 & \textbf{else}\ \mathit{Node}\ l\ k\ (\mathit{maybe\text{-}assign}\ x\ (\textbf{if}\ b\ \textbf{then}\ \mathit{Yes}\ v\ \textbf{else}\ \mathit{No})\ r) \\
= & \quad \{\ \text{synthesized } \mathit{Node} \text{ case of } \mathit{maybe\text{-}assign}\ \} \\
 & \mathit{maybe\text{-}assign}\ x\ (\textbf{if}\ b\ \textbf{then}\ \mathit{Yes}\ u\ \textbf{else}\ \mathit{No})\ (\mathit{Node}\ l\ k\ r)
\end{array}$$

As before, to perform the last step we had to synthesize the *Node* case of the definition of *maybe-assign*:

$$\begin{array}{l}
\mathit{maybe\text{-}assign}\ x\ u\ (\mathit{Node}\ l\ k\ r) \triangleq \\
\qquad \textbf{if}\ x \leqslant k \\
\qquad \textbf{then}\ \mathit{Node}\ (\mathit{maybe\text{-}assign}\ x\ u\ l)\ k\ r \\
\qquad \textbf{else}\ \mathit{Node}\ l\ k\ (\mathit{maybe\text{-}assign}\ x\ u\ r)
\end{array}$$

By pushing the synchronization point downwards, we have constructed a program that always updates, but updates with the original content in the event of failure. This might be considered an optimistic update; in the context of transactions that rarely fail, this optimism is justified.

The improvement results from the fact that the transformed program produces a more defined result when the condition b is undefined – implying that more work can be done before having to wait for input. Although the prospects for automating a transformation like this seem bleak, the semantic basis for comparing the two is intriguing.

4 Increasing grain size

Our revised program is naturally partitioned as a thread for each *lookup* and one for each *maybe-assign*. It is necessary to perform all the assignments concurrently, so that the upper levels of the index are made available to the next transaction as early as possible. However, we can increase the grain size still further.

https://doi.org/10.1017/S0956796801003793 Published online by Cambridge University Press

A well-known transformation (Burstall and Darlington, 1977) combines two similar passes over a data structure into one, saving some effort in searching through the data structure. In the parallel context, this transformation may fuse two similar processes, creating a larger process with greater locality. Here, we shall apply this procedure to *lookup* and *maybe-assign*, which repeat the same process of searching for the key in the tree. Applying the usual technique, we define a function that performs both operations

$$update\ :\alpha \to Maybe\ \beta \to DB\ \alpha\ \beta \to Maybe\ \beta \times DB\ \alpha\ \beta$$
$$update\ k\ v'\ d \triangleq (lookup\ k\ d, maybe\text{-}assign\ k\ v'\ d)$$

A standard application of the fold-unfold technique to the above definition of *update* yields an equivalent recursive definition of *update*:

$$update\ x\ u'\ (Leaf\ k\ v) \triangleq$$
$$\quad (u, Leaf\ k\ v')$$
$$\quad \textbf{where}\ (u, v') \triangleq \textbf{if}\ k = x\ \textbf{then}\ (Yes\ v, u'\ \mathsf{or}\ v)\ \textbf{else}\ (No, v)$$

$$update\ x\ u'\ (Node\ l\ k\ r) \triangleq$$
$$\quad \textbf{if}\ x \leqslant k$$
$$\quad \textbf{then}\ (u, Node\ l'\ k\ r)$$
$$\qquad \textbf{where}\ (u, l') \triangleq update\ x\ u'\ l$$
$$\quad \textbf{else}\ (u, Node\ l\ k\ r')$$
$$\qquad \textbf{where}\ (u, r') \triangleq update\ x\ u'\ r$$

The *update* function uses its database argument immediately, making the top level of its database result available as soon as it moves to the next level. Only when it reaches the leaf does it make its value result available, after which it consults its value argument to determine whether to alter the value in the leaf. Thus several *update*s may be pipelined.

Recall that we assumed that all keys within a transaction were distinct, so that the five updates are independent:

$$lookup\ x_i\ db_{i-1} = lookup\ x_i\ db_0$$

for each $i = 2, \ldots, 5$. We shall use this independence in the reverse direction, serializing the composition of the *update*s. Thus we can restate the transaction processor as follows:

$$apply\ t\ db_0 \triangleq$$
$$\quad \textbf{let}$$
$$\qquad (u_1, db_1) \triangleq update\ x_1\ u'_1\ db_0$$
$$\qquad \ldots$$
$$\qquad (u_5, db_5) \triangleq update\ x_5\ u'_5\ db_4$$
$$\qquad (u'_1, \ldots, u'_5) \triangleq decide\ t\ u_1 \ldots u_5$$
$$\quad \textbf{in}$$
$$\quad db_5$$

https://doi.org/10.1017/S0956796801003793 Published online by Cambridge University Press

The **let** in this program is recursive: the u_i are defined in terms of the u'_j, which in turn are defined in terms of the u_i. However, there is no circularity in the computation, since as noted above *update* provides the value of u_i before it examines the value of u'_i. Moreover, the only part of the database that depends upon u'_i is the data value in one leaf.

5 Experimental design

To evaluate the parallel performance of the new form of the transaction processor, we have studied its behaviour on an implementation of parallel graph reduction. This in turn runs on a simulation of a shared-memory multiprocessor. We have chosen to use simulation rather than an implementation on real hardware since it allows the behaviour of the system to be closely monitored without affecting its behaviour, permitting, for example, counts of important events to be made without changing the schedule of the computation or its simulated execution time.

5.1 Compiler and run-time system

Our complete implementation consists of an optimizing compiler for a functional language that generates an executable when linked with a parallel run-time system, which contains the graph reducer itself. The source language is a lazy, higher-order functional language in the tradition of SASL and Haskell (Hudak *et al*., 1992). The primary objective in building an optimizing compiler for a lazy functional language is to reduce the frequency at which claims and references are made to the heap. It is therefore of great importance that the compiler used in our experiments should perform well. We have adopted the compiler developed for the FAST project (Cox *et al*., 1992), and although comparing compilers is difficult, we are confident that the system is competitive with the state of the art (Hartel and Langendoen, 1993). It also, conveniently, generates C, making generated code very easy to instrument and modify. A comprehensive description of the parallel graph reducer and simulator design can be found in Bennett (1993).

5.2 Parallel graph reduction: closures and 'sparking'

Our parallel implementation is based on a parallel graph reducer that supports the 'call-by-need' parameter passing mechanism required by lazy functional languages by using closures. Function arguments are represented by 'closures'. A closure is a heap-based object containing a method for computing a value; the method is only invoked when the closure is *demanded*, and on completion of the call the closure is overwritten by its result value. Closures cannot be allocated on the stack since their lifetime is unknown: a heap is used instead. The programmer may specify parallel evaluation of closures: an expression is annotated with the keyword **spark**, requesting that the closure representing the expression be added to a pool of available tasks for distribution to other processing elements (Peyton Jones *et al*., 1987). If a closure that is in the process of being evaluated is demanded by another thread, the second

https://doi.org/10.1017/S0956796801003793 Published online by Cambridge University Press

thread blocks until evaluation has been completed. A thread does not need to block when an evaluated closure is demanded – the value can be used directly.

Closures are used in several places in the new formulation of the transaction processor. In the definition of the *apply* function in section 4, each call of *update* is packaged into a closure and then sparked. The call to *decide* is also built as a closure, but is not sparked since its arguments will not be available until the *update*s have reached the leaves of the tree. The u_i' parameters of the *update* closures are components of the value of the *decide* closure. The first call of *update* to reach a leaf node of the tree will demand its u_i' parameter, in turn evaluating the *decide* closure. Any other *update* requiring a u_i' will block until *decide* completes, at which point the new result database can be returned. This is described in detail in section 6.1.

5.3 Simplifications

A number of assumptions and simplifications have been made in our experimental design: it is a compromise between the need to model the important effects, and the need to study these effects in isolation. Our objective is to learn general lessons about a large class of systems, and we are therefore less concerned that the experiments predict the actual performance of some production system, as to do so we would have to introduce many factors that are orthogonal to the issues we intend to study. These simplifications are outlined in the design below.

5.3.1 Scheduling

Parallel graph reduction is inherently dynamically scheduled, and mechanisms are required to distribute sparked tasks around the machine. Logically a single shared task pool is used, but this will inevitably become a source of contention in a large-scale system if implemented as a single queue. The usual approach taken in a large system is to equip each processor with a local task queue to which closures it sparks are added and from which new work can be fetched when necessary. Mechanisms are therefore required to distribute tasks around the machine on demand. Policies relating to this form of scheduling are described and evaluated experimentally in Goldberg (1988).

We have adopted such a work stealing scheme, except that we model ideal behaviour when a processor needs to find a sparked task and none is available locally: the oldest sparked task is used. We do not account for the communications traffic involved in implementing this (i.e. the traffic required to find the oldest sparked task in the machine). If no work is available at all, the process is removed from the set of processes under simulation and added to a queue of idle processors. It is either resumed later when work becomes available, or remains in this state until evaluation of the functional program completes and the simulation system terminates. Note that once execution of a thread has begun, it will not migrate – we return to this issue in section 6.1.

5.3.2 Simulated architectures

Our first simulation results are based on an ideal shared memory in which all accesses have unit cost, allowing optimum speedups to be determined, and scheduling policies to be evaluated without being affected by the performance of the memory system and the interconnection network. We then study the influence of using two multicache shared-memory designs that are suitable for implementation in hardware, one of which has been previously shown to be particularly effective with parallel graph reduction (Bennett and Kelly, 1994).

A simple model of a shared-memory parallel architecture has been adopted: the system consists of a set of processing elements (PEs), each comprising a processor and a large cache. The PEs are interconnected by a network. This is a cache coherent shared-memory multiprocessor such as the SGI Origin 2000 (Laudon and Lenoski, 1997) or the Convex Exemplar (Thekkath, 1997). Some systems also have separate memory units, but for simplicity we assume that the caches are large enough that the memory units need not be simulated. The processor model is based on a simple 32-bit RISC (i.e. load/store) device. It is assumed that stack, private data and code regions of each process are served by separate perfect cache systems; each read or write to these areas has unit latency. Cache associativity conflicts and cache capacity events are ignored, i.e. it is assumed that each cache is infinite in size, and therefore associativity is not an issue.

Between consecutive global events produced by an application process, an amount of time is spent accessing local memory, performing primitive arithmetic operations and in other local operations: the clock associated with each application process must be altered to account for this time in order to maintain a correct ordering of global events. Compiler generated code and the runtime system are annotated with cycle counting code, which assigns a cost of two cycles for each local event instruction.

5.3.3 Garbage collection

Our simulation assumes sufficient memory is present that garbage collection is not needed. This is clearly unrealistic, but garbage collection in large parallel systems lies outside the scope of the current investigation.

In general, garbage collection can have a dramatic effect, both because of the time taken, and because of its impact on data placement. In the simulation, each CPU allocates from its own contiguous region of the shared address space. Thus, spatial locality derives from the sequence with which each CPU allocates space. This approximates what would happen with any copying or compacting collector.

The problem of garbage collection using our TLO protocol is discussed in Bennett and Kelly (1994). We return to this issue in our concluding sections.

6 Parallel behaviour

To run the transaction processor on a parallel machine, we need to indicate the parallel grains by annotating the program with **spark** directives. Each *update* must

https://doi.org/10.1017/S0956796801003793 Published online by Cambridge University Press

be run independently, so that the following transaction will not be blocked in any part of the tree. However, the program processes the root node in a purely sequential manner, so locality will be increased with no loss of concurrency if a single thread is responsible for all root updates. Hence, we unfold the definition of *update*, sparking threads to update nodes below the root:

```
update k u' (Leaf k v) ≙
    (spark u, Leaf k v')
    where (u, v') ≙ if k = x then (Yes v, spark(u' or v)) else (No, v)

update x u' (Node l k r) ≙
    if x ⩽ k
    then (spark u, Node l' k r)
         where (u, l') ≙ update' x u' l
    else (spark u, Node l k r')
         where (u, r') ≙ update' x u' r

update' k u' (Leaf k v) ≙
    (u, Leaf k v')
    where (u, v') ≙ if k = x then (Yes v, spark(u' or v)) else (No, v)

update' x u' (Node l k r) ≙
    if x ⩽ k
    then (u, Node l' k r)
         where (u, l') ≙ update' x u' l
    else (u, Node l k r')
         where (u, r') ≙ update' x u' r
```

Evaluation of the first component of the result of the function *update* requires computation all the way down to the leaf, so by sparking it we ensure that a single thread performs this scan.

In general, a sparked closure is evaluated only to weak head normal form, exposing the topmost constructor. A further **spark** is required to schedule work on arguments of that constructor. This occurs, for example, in the *Leaf* case of *update*′. In the implementation we are using, a small optimization is performed in such a case: if a thread is about to terminate returning a constructed object, and one of the arguments of the constructor is a **spark**, the thread starts work on that argument. This increases the grain size of parallelism, saving the overhead of thread creation, and increases locality within threads. In this particular example, the combined effect is to spawn a single thread for each *update*, after the root node is reconstructed by the master thread.

For our experiments, we used the following parameters:

- Size of the database: 10000 consecutively numbered records, yielding a binary tree of approximately 13 levels. This size was adopted in order to produce a reasonable execution time for each simulation run.

https://doi.org/10.1017/S0956796801003793 Published online by Cambridge University Press

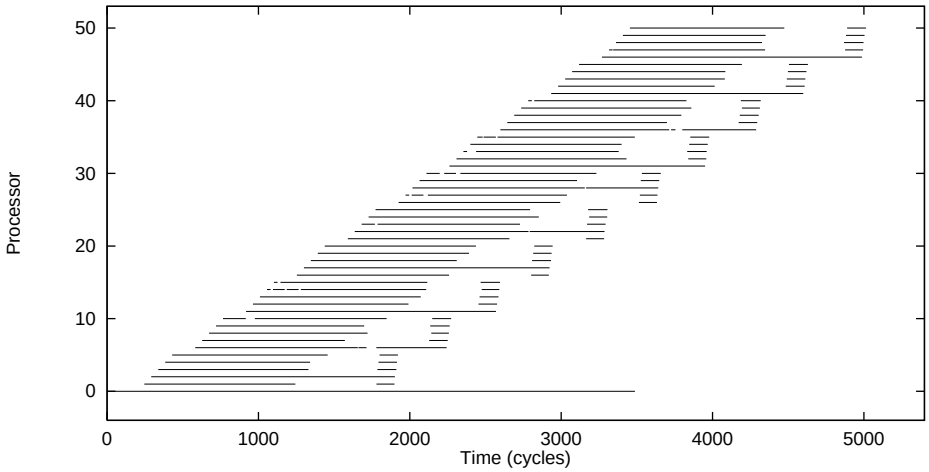

Fig. 2. New nodes created by *assign*.

- Initial distribution of the database: the entire database is divided into subsets of consecutively numbered keys, which are allocated to individual PEs.
- Number and size of transactions: 10 transactions each operating on five randomly selected records. Some runs of 100 such transactions were also performed.
- No disks are simulated, i.e. the database is entirely RAM-resident.

6.1 Results under an ideal memory model

To study the concurrency of the program independently of memory cost issues, our first simulations were of a system in which all memory accesses had unit cost. The parallelism present is clearly illustrated by figure 2, Activity on a 64-processor ideal shared-memory machine (see section 5.3.2 for explanation of simulation model). showing the activity of each processor when the program was run using a large number of processors to give a simple picture. As can be seen from the figure, the master thread spawns five threads for each transaction and then terminates. Each of these threads represents an *update*, scanning from the root of the tree to a particular leaf. The path of nodes to that leaf are updated by the thread, so any following thread requesting the same node will block. As the figure shows, such collisions usually occur in the upper levels of the tree, and are transient. They are most noticeable where several keys are close together in the tree, and thus have long paths in common, as in the seventh transaction (processor number 33). For each transaction, the first *update* thread to reach its leaf goes on to apply *decide*, while the others block. When the new values are known, all the *update* threads commit together.

Since the keys in the transactions are spread across the tree, collisions are rare. However, in our example, one of the keys to be updated by the seventh transaction has been updated by the fifth. Thus the evaluation of *decide* blocks until the value

https://doi.org/10.1017/S0956796801003793 Published online by Cambridge University Press

is updated. If our implementation supported a more sophisticated $\wedge$ connective, evaluating the conditions concurrently, slightly more parallelism would be possible.

6.2 Achieving a reasonable schedule

From figure 2 we can form some expectations about the behaviour on fewer processors. When a thread blocks after reaching a leaf, the processor on which it is running may usefully take up another *update* thread. The modification of the leaf data can be done later; only another *update* requiring the same leaf will be blocked. When transient blocks occur high in the tree (usually near the root), the processor takes up a new thread. This thread also quickly blocks, as it requires the node held by the former thread (or a similarly blocked thread on another processor). In this way, the processor accumulates many blocked threads. Moreover, other threads requiring the output of these also block. Since in our simulation threads do not migrate, most of the processors will be idle, waiting for a few heavily loaded processors. Once such a load imbalance occurs, the situation deteriorates rapidly.

One attack on this problem is to break up the grain size of the computation by timesharing between runnable threads, allowing queued threads to progress far enough down the tree to release other blocked threads. Another is to allow runnable threads to migrate to idle processors, but this is typically rather expensive. However, it is apparent from the program that

- the *update*s are pipelined, and
- the nodes and leaves of the tree are built quickly.

Thus we have introduced a new annotation, **quick**. An expression **quick** *e* is semantically equivalent to *e*, with a hint to the implementation that the value of *e* can be computed quickly, so that if a thread blocks waiting for the value of the annotated expression, the processor should not schedule extra work. *Quick* simply sets a bit on the closure, which is inspected by the evaluator before blocking.

Automatic detection of short computations is useful for other purposes (for example to avoid the overhead of thread switching, or to avoid creating short-lived threads) and some partial solutions are known (Goldberg, 1988).

We use **quick** for the recursive calls to *update* (see section 6). With this modification there are some momentary delays high in the tree, but otherwise the processors are almost completely utilized. For example, figure 3 shows activity and state transitions on a 10-node machine. Note that there are three different typical lengths of activity, in decreasing order:

- The scan from the root of the tree to the leaf.
- The work of *decide*.
- The actual update of the leaf.

As can be seen, the work of deciding whether to commit, and of updating the leaves may be done one or more scans later than the *update* of which they were part.

Figure 4 shows the relative speedup achieved for various sizes of machine. We have used a larger number of transactions (100) to reduce the significance of the

https://doi.org/10.1017/S0956796801003793 Published online by Cambridge University Press

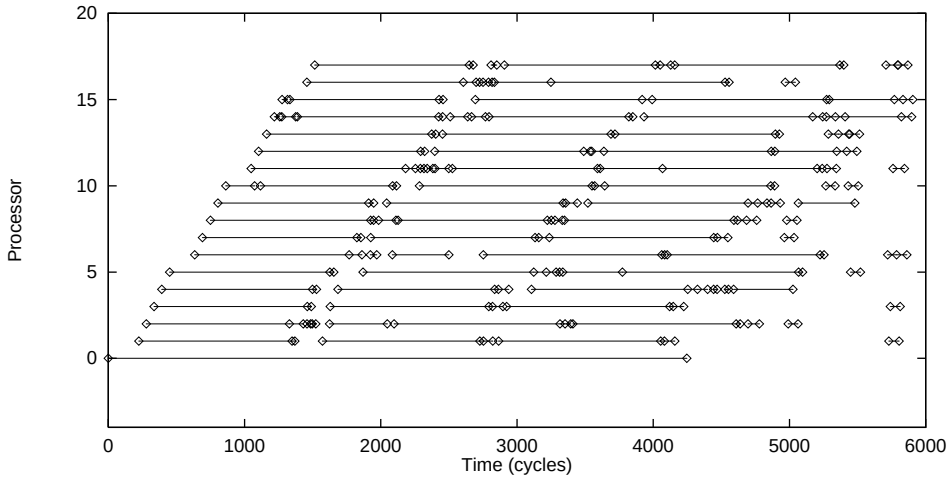

Fig. 3. Activity and state transitions on a 10-processor machine (see section 5.3.2 for explanation of simulation model).

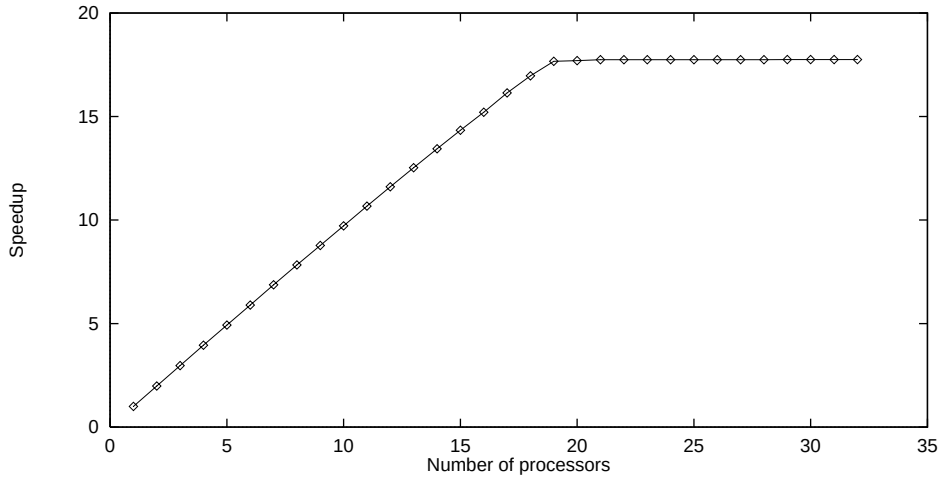

Fig. 4. Relative speedup on an ideal shared-memory.

uneven finish times seen in figure 3. As one might expect from figure 2, the graph shows near-linear speedup up to the ratio between the cost of processing the root node (by processor 0) and the cost of processing the entire path down the tree and performing the update, which is proportional to the depth of the tree (Trinder, 1989). For our sample tree of 10,000 nodes, the asymptotic speedup is 18.

7 Multicache shared-memory

Simulating an ideal shared-memory has enabled us to isolate the functional transaction processor from details of the implementation of the shared-memory and the performance of the interconnection network. This has allowed us to measure the parallelism inherent in the application, thereby setting a performance standard

https://doi.org/10.1017/S0956796801003793 Published online by Cambridge University Press

against which to compare simulated execution results from more realistic architectural models. In addition, we have identified a scheduling problem, which was resolved by introducing a simple annotation. However, network latency is relatively large in modern multiprocessors and the influence of the network must be addressed.

Specifically we address four main issues:

- How can the single shared heap be efficiently maintained?
- Should the scheduling policies already adopted be changed to accommodate the increased latency?
- Is significant locality of reference available in the program, and can it be exploited?
- Finally, how is speedup affected by network latency and cache line size?

These questions are addressed by means of simulation experiments in section 8. In the following three subsections we discuss how the shared heap can be supported on shared-memory multiprocessors, and describe further details of the experimental design.

7.1 Caching, coherency and line size

Since the class of architectures we are considering has relatively high latency interconnection networks, memory references that can be satisfied by the local cache can take place quickly; otherwise the operations incur the cost of arbitrating for the network, a remote cache operation, and a message transaction on the network itself. Minimizing the number of shared-memory references requiring use of the network will therefore improve execution times.

Our previous results demonstrated that cache coherent shared-memory multiprocessors can effectively support the shared heap required by parallel graph reduction (Bennett and Kelly, 1993). In such a system the shared-memory region is divided into cache lines of some constant size. Each line has a single owner; ownership changes dynamically according to coherency transactions. When a processor attempts to read a line that is not present in its cache, a read request is sent to the owner of the line, which responds by sending a copy of the line. The requesting processor adds the line to its cache (expelling another line if the cache is full) and proceeds to use it. In this way, multiple copies of lines come into existence in the system. A write to a line that is present in the local cache but not in any remote caches can take place without using the network.

The problem comes when a processor attempts to write to a cache line that exists in more than one cache: in a conventional protocol the other copies of the line must either be invalidated or updated when the write takes place (Archibald and Baer, 1986); either will require use of the network and will have a large latency. We have previously proposed a coherency protocol, called Two-Level Ownership (TLO), which has been shown to be more effective at supporting the sharing requirements of parallel graph reduction than either invalidation or update (Bennett and Kelly, 1994). This new scheme is described in section 7.2 and used to produce the results in section 8.

The design issue of cache line size is particularly important. Since heap cells are allocated incrementally, the new path from the root of the database to a leaf constructed by a call of *update* is built in a contiguous region of the shared-memory. Since the tree is binary, there is therefore a 50% probability that the node at the next level down the tree encountered when scanning down to a leaf from the root will be adjacent to its parent, which could be on the same cache line. Large cache lines are therefore beneficial since they allow this locality to be exploited.

Modern shared-memory multiprocessors favour the use of such large cache lines. The latency of sending a message (or cache line) is dominated by two costs: a constant startup time and a time related to the the message length. Modern multiprocessors are characterized by having high latency and high bandwidth interconnection networks (Markatos and LeBlanc, 1992), and therefore once the high latency setup time has been incurred, data can be rapidly pipelined through the network. The latency of large messages is not significantly worse than that of small ones. For example in the Meiko CS-2, a unit sized message can be sent in 10μs, whereas 400 bytes takes only twice as long (Barton *et al.*, 1994), and if sufficient locality can be exploited by such large messages to more than halve the number of messages required during a program run, an improved execution time may result.

7.2 The TLO protocol

The protocol is based on the three-stage lifetime of closures:

INACTIVE**:** The closure has not been evaluated, and no thread has yet gained the right to evaluate it.

ACTIVE**:** The closure has not been evaluated, but a thread has gained the right to evaluate it, and the closure will be updated by its result by that thread at some time in the future.

EVALUATED**:** The closure has been evaluated and will not be updated again.

When a PE needs the value of a given closure, it checks to see whether it has a local, EVALUATED copy – if so, this is used. If not, it sends a request to the closure's 'owner' – the PE which allocated it (easily detected by inspecting its address):

- If the owner's copy is already EVALUATED, it replies immediately. The reply contains the entire cache line on which the closure falls, and this replaces the requesting PE's version of that data (since it is guaranteed to be at least as up-to-date).
- If the owner's copy is INACTIVE, the owner atomically marks the closure as being evaluated by the requesting PE, then sends the cache line as before – the requesting PE must then evaluate the closure and send an update message to the owner when it finishes.
- If the owner's copy is ACTIVE, the owner adds the requesting thread to a list of blocked threads associated with the closure. Some PE is currently evaluating the closure, and on completion it sends an update message to the owner. The owner then marks the closure as EVALUATED and replies to each PE on the list with the updated cache line.

https://doi.org/10.1017/S0956796801003793 Published online by Cambridge University Press

When a PE sends a request message, the requesting thread blocks and the processor switches to a runnable thread, or if none exists, starts a newly-sparked closure from its work pool.

Copies of remotely created cache lines are made whenever a network transaction is made. Incoherent copies of lines come into existence when a line that resides in more than one cache is updated, but this can be done safely since any access to a non-EVALUATED closure on such a line will result in a network transaction with the owner of the line whose copy is always fully coherent.

7.3 The memory timing models

This model determines the latency of each heap access. Two models are used: the low latency model represents a first-generation shared-memory machine such as the Sequent Balance (Thakkar *et al.*, 1988) in which a shared-memory access requiring use of the network has a latency of about an order of magnitude greater than one that can be satisfied by the local cache. Specifically, access latencies are either 1 cycle or 10 cycles.

The high latency model places a greater cost on memory references requiring use of the network, and more closely represents the cost of communication in modern scalable multiprocessors. An access that can be satisfied by the local cache has a latency of 1 cycle, whereas others have a latency of 100 cycles.

Note that it is the *relative* cost of communication that has changed. The speed of processors has improved considerably more than the speed of communication; modern shared-memory multiprocessors have much shorter cycle times than older machines.

We do not simulate details of the interconnection network. We assume messages are handled with very little work by the processors, and they are delivered after the given latency. We do not model pipelined message transfer, nor do we model blocking due to contention. Note in addition that the transaction latencies are kept constant despite varying the cache line size. All subsequent results are based on 100 transactions, each operating on five randomly selected records.

8 Program behaviour with a multicache shared-memory

In this section we present some experimental results showing the behaviour of our program on a more realistic shared-memory. First, we consider the communication requirements of the program in comparison with experimental results, and show how the TLO protocol allows the locality of the program to be exploited. We then address some scheduling issues raised by high latency networks, before presenting an empirical comparison between the invalidation and TLO protocols.

8.1 A model of communication requirements

How much communication is required by the transaction processor? We shall distinguish between communications that deal directly with the database, and those that do not.

https://doi.org/10.1017/S0956796801003793 Published online by Cambridge University Press

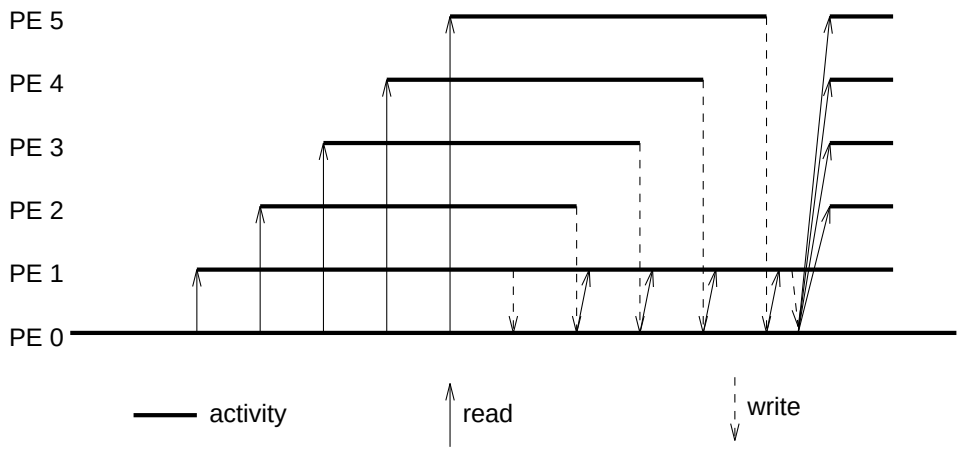

Fig. 5. Communication within a transaction.

Consider first the data cells representing the database itself. Creation of new nodes involves no communication, since these nodes are created on cache lines owned by the creator. Moreover, each *update* operation creates a path of new nodes from the root to a leaf, and these new nodes will be adjacent in memory. For each node accessed by some subsequent operation, the probability that the node was created by the same process as its parent will be 50%. Hence as cache line size increases, approximately half of node accesses will be satisfied by the local cache, while half will require network communication.

The other communications are those required to start new processes, and the internal communications within a transaction. The essential communications for a single transaction are summarized by figure 5. The master thread (on processor 0) starts five *update* threads on processors 1 to 5. Each of these must read its arguments from the cache of processor 0. These threads then run without further communication (except involving the nodes of the tree, as discussed above) until they reach their respective leaf nodes, when each returns the current value in its leaf to some central point. Then the first thread to reach its leaf begins to evaluate the *decide* condition, while the other threads block, waiting for the result. With the version of the TLO protocol used in our experiments, returning the current value of each leaf involves updating a redex created (and therefore owned) by processor 0. The thread evaluating the condition must then retrieve the values from the cache of processor 0. It need not retrieve the value it (needlessly) wrote itself, as it is already present. Similarly, since the *decide* redex was also created by processor 0, its result must also be passed through the cache of processor 0, and then each *update* thread is able to complete.

Clearly, the simple ownership scheme used here is limited: any communication between two threads must use a redex created by a common ancestor of the threads. Thus an extra processor will often be involved. Moreover, in a situation like the transaction processor, where all the worker threads are children of a single master,

https://doi.org/10.1017/S0956796801003793 Published online by Cambridge University Press

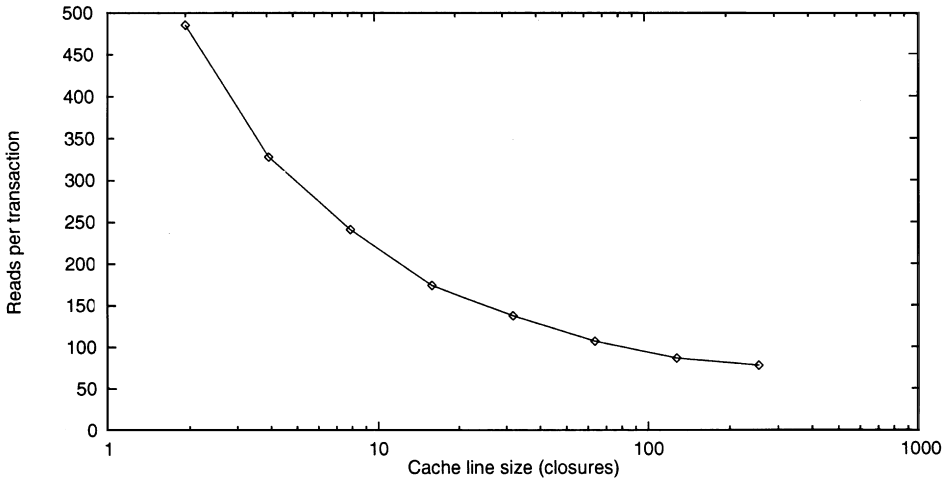

Fig. 6. Measured network reads per transaction, for the TLO protocol on a 64-processor machine.

the servicing of cache requests on the central processor may interfere with the work of the master thread. It would be better if the creator of a redex were able to transfer its ownership to another thread. In the transaction processor, this would remove the contention for the cache of processor 0, and might reduce the number of writes, but the same number of reads (13) would still be required. Also, note that it is not possible to determine in advance which of the *update* threads will evaluate the *decide* condition.

We have described the ideal situation. A functional implementation typically creates many extra cells, for intermediate tuples, indirections, and so on. However these are adjacent to the required values, so that while they will generate many extra network reads with small cache lines, they will have little effect with large enough cache lines. Figure 6 shows the observed number of reads requiring use of the network per transaction using the TLO protocol for various cache line sizes on a 64-processor machine. The simulation reflects a cold start, with the database distributed across the processor caches, so that each node access will require a cache read, except for a few nodes near the top of the tree. Hence the asymptotic number of reads per transaction is $13 + 5d$, so with a database of 10,000 nodes ($d = 13$) we expect the asymptotic number of reads per transaction to be 78. Figure 6 shows that the TLO protocol achieves this. After the transaction processor has been running for a long time, the locality of database nodes will increase, as described above, and the number of reads required by database node accesses will approach $13 + 5\frac{d+1}{2}$.

8.2 Unnecessary updates

While the unnecessary reads caused by the extra data structures representing intermediate results are eliminated by large cache lines, the corresponding writes are a more serious problem. With the TLO protocol, which lacks migration of ownership,

https://doi.org/10.1017/S0956796801003793 Published online by Cambridge University Press

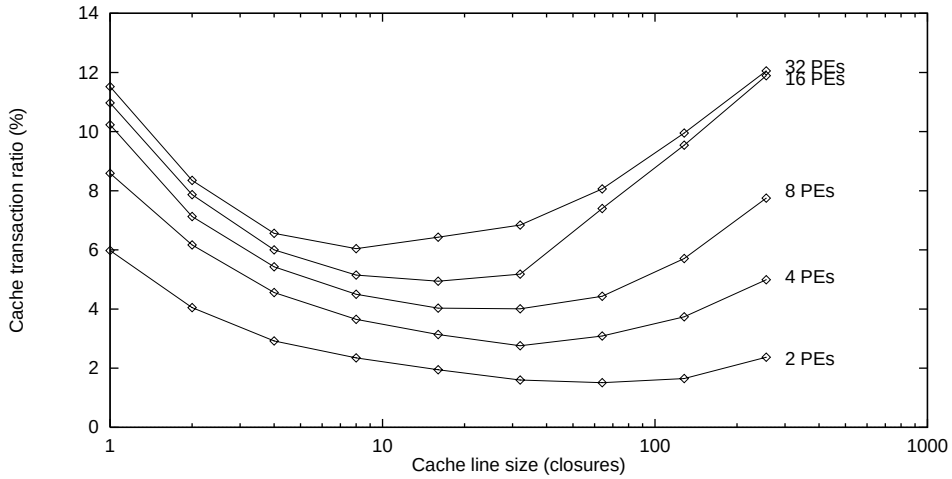

Fig. 7. Cache transaction ratio for the invalidation protocol using 2, 4, 8, 16 and 32 processors.

communication due to updates cannot be alleviated by increasing cache size. The number of updates required will vary greatly with the quality of the implementation, as lazy evaluation causes the fetching of small packets of work from their owners and writing back of their results. With a sophisticated implementation, eliminating unnecessary intermediate data structures, we would expect the number of updates to conform to our model.

8.3 Exploiting locality

In this section the model presented above is used to interpret counts of heap accesses reported by simulation runs. The model predicts the number of reads made by a transaction, whereas the simulation results include writes and accesses to closures representing intermediate results created by the functional language implementation. The simulation results are used to quantify the advantage gained by adopting large cache lines in the implementation of the shared heap.

All references to the heap are classified according to whether they require use of the interconnection network. Counts of each heap reference type were maintained during simulation, and used to produce graphs in which the percentage of all heap references made that require use of the network (referred to as the cache transaction ratio) is plotted against cache line size for a variety of processor configurations. Results from a conventional invalidation protocol are shown in figure 7, and from the TLO protocol in figure 8.

A reduction in cache transaction ratio can be seen when moving from the minimum to a line size of two closures in each case. This represents a reduction in the load on the network, and ultimately in execution time. However, in the case of the conventional protocol, a minimum is reached (at a point dependent on the number of processors), after which the trend reverses. The increase is due to *false*

https://doi.org/10.1017/S0956796801003793 Published online by Cambridge University Press

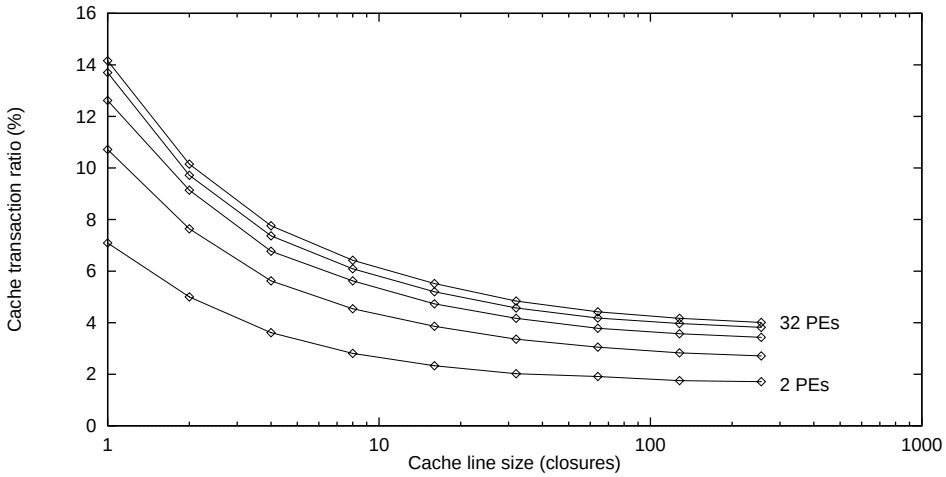

Fig. 8. Cache transaction ratio for the TLO protocol using 2, 4, 8, 16 and 32 processors.

sharing (Dubois, 1992), a contention effect commonly observed when conventional coherency protocols are used with large cache lines.

The graph for the TLO protocol shows that, regardless of the number of processors used, the cache transaction ratio falls as the cache line size is increased from 1 to 256 closures. An increase never occurs as the line size is increased, showing that the false sharing problem has been eliminated. False sharing is described in more detail, with experimental results from other functional programs in Bennett and Kelly (1994). The remainder of the results in this paper are based on the TLO protocol.

Although the reduction in cache transaction ratio is often small, the effect it has on simulated execution time is determined by the latencies of accesses requiring use of the network, which are many times greater than the latency of a cache-hit for the class of architecture envisaged. Therefore even relatively small variations produced by changing line size can have a significant effect on performance.

The reduction in cache transaction ratio as line size is increased is due to locality. Initially the reduction is very significant, but the cache transaction ratio begins to level off at the largest line sizes. This behaviour is largely due to the reduction in reads resulting from increasing line size, which can be explained by the analysis of section 8.1.

8.4 Scheduling and latency

In this section we consider some scheduling issues raised by network delays. All the experiments here used the TLO protocol, with a very large cache line size (256 cells).

The behaviour of our program on a relatively low latency network (remote accesses 10 times more expensive than local ones) is shown in figure 9. As expected, everything takes longer than with an ideal memory (figure 2), but the overall shape of the computation is much as before. In particular, the asymptotic speedup is unchanged, but a larger number of processors will be required to achieve it. Note, however, that the transient delays that occur near the top of the tree are longer.

https://doi.org/10.1017/S0956796801003793 Published online by Cambridge University Press

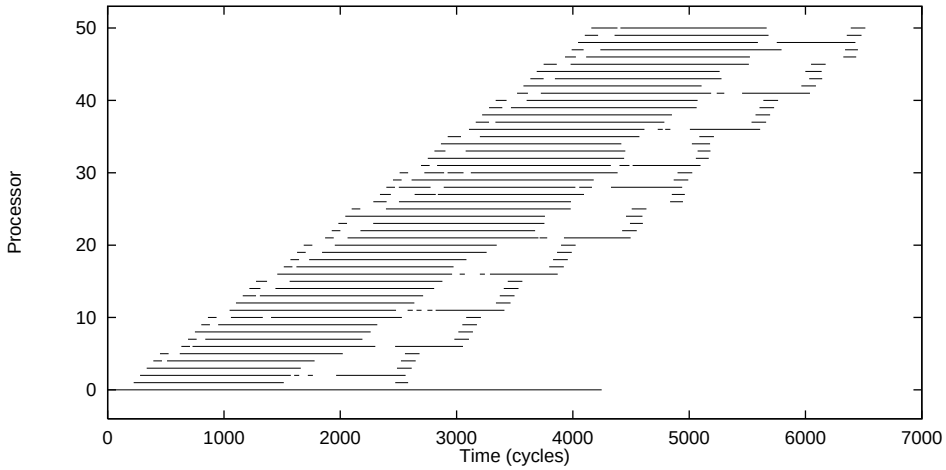

Fig. 9. Activity on a low latency network.

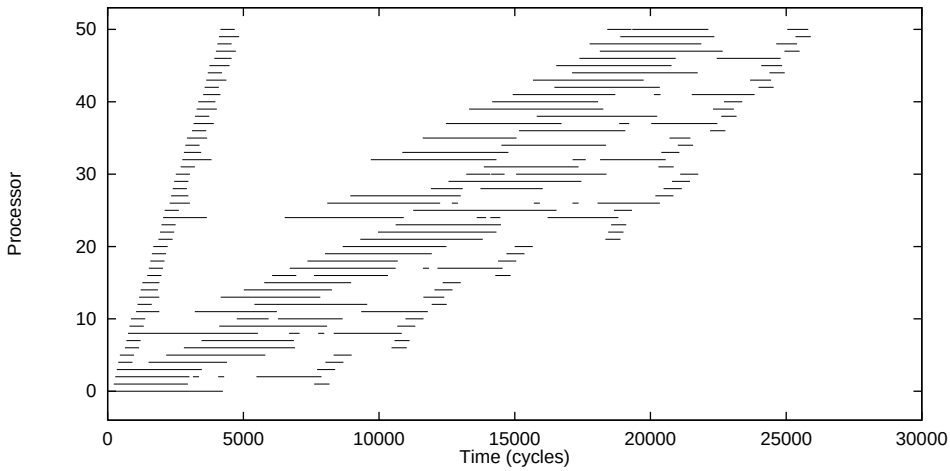

Fig. 10. Activity on a high latency network.

If the network latency is greatly increased relative to processor speed (see section 5.3.2), we have the situation of figure 10. Here remote accesses are 100 times more expensive than local ones. The slope of the leading edge, the thread creations, is identical to that in figure 9. As discussed in section 6.1, this is determined by the time taken by the master thread running on processor 0 to reconstruct the root node in each update. While latency was low, this was the limiting factor on the incremental time required per transaction. Now the limiting factor is the rate at which the two nodes at the second level of the tree can be passed between the update threads. Indeed, closer examination of figure 10 reveals that the thread resumptions form two arcs, reflecting queues for the left and right child nodes of the root.

What can be done? Throttling process creation will prevent the system from being overloaded with useless processes, but will not increase throughput, which is

https://doi.org/10.1017/S0956796801003793 Published online by Cambridge University Press

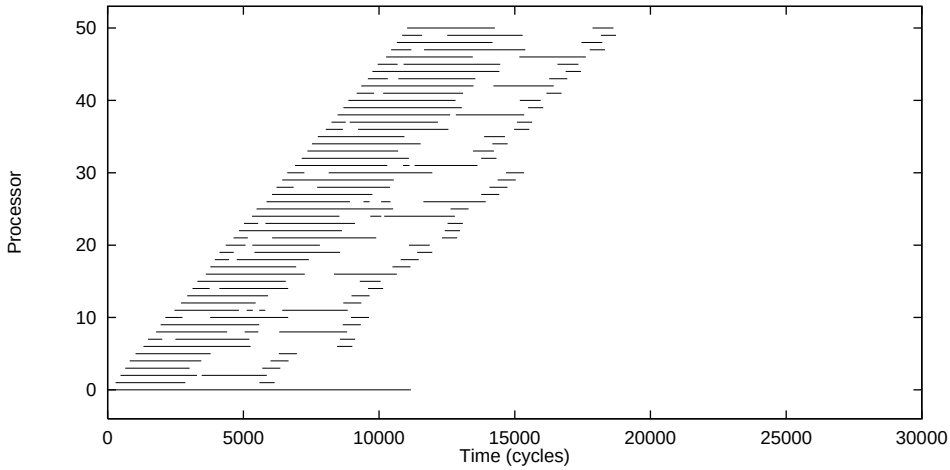

Fig. 11. Activity on a high latency network with delayed **spark**s ($k = 3$).

bounded by these queues. It is necessary to increase the number of queues, which can be done by moving the **spark**s further down the tree, effectively broadening the root node. This has two effects:

- The work done by processor 0 on each *update* increases, a throttling effect decreasing the slope of the leading edge in the figure.
- The *update*s are divided between a larger number of queues, increasing the slope of the thread-resumption part of the figure and reducing overall time.

If the **spark**s occur after k levels, processor 0 must process k nodes, and the threads are divided between 2^k queues. Assuming a uniform distribution of keys, if t_n is the time to reconstruct a node and l the latency:

- If latency is low, the incremental time per update is bounded by the time taken by processor 0 to process those k nodes, i.e. kt_n.
- If latency is high, the incremental time is bounded by the rate at which each node at level k is passed from thread to thread, divided by the number of such threads (since the nodes are updated concurrently), i.e. $\frac{l+t_n}{2^k}$. The node must be fetched across the network (l) and updated (t_n).

Hence a crude estimate of the incremental time per update is

$$\max\left(kt_n, \frac{l+t_n}{2^k}\right)$$

Figure 12 shows the total time taken to process 100 transactions (which is roughly proportional to the incremental time) for various latencies and various values of k. The number of processors used is very large. As expected, for each value of k, the incremental time is constant (and proportional to k) until the point where the latency becomes dominant, and the time grows linearly.

An alternative view is to treat the situation as a queuing system, with the average inter-arrival time in each queue is $k2^k t_n$, while the service time is $l + t_n$. To achieve

https://doi.org/10.1017/S0956796801003793 Published online by Cambridge University Press

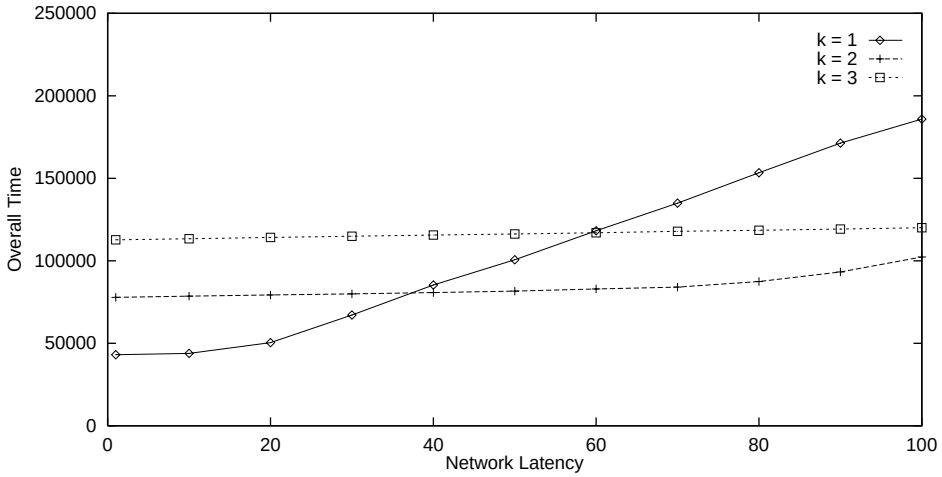

Fig. 12. Total time *vs* latency for different values of k.

a stable system (the flat part of the graphs in figure 12), we choose k such that

$$k2^k t_n > l + t_n$$

For the network parameters reflected in figure 10, this value is 3, leading to the behaviour seen in figure 11. The processors are used for shorter periods, and the overall time has also improved. However, in comparison with the ideal memory model (figure 2) the asymptotic speedup is reduced by this factor k.

According to figure 8.4, a value of $k = 2$ gives a faster overall time on infinitely many processors, but the unstable queues will grow indefinitely, eventually exhausting the available memory.

8.5 Relative performance

The success of exploiting locality has been clearly demonstrated using event counts rather than simulated execution times in order to reduce the influence of the particular latencies assigned to various events in the simulation. However, the sensitivity of relative performance to network latency and cache line size is ultimately of greatest importance. Graphs of relative performance for the TLO protocol are shown in figure 13 for the low latency network, and figure 14 for the high latency network. In each case, spark annotations were placed at a level in the tree appropriate for the network latency: at the top-level for the low latency network (i.e. $k = 1$), and the third level for the low latency network (i.e. $k = 3$).

Recall that an asymptotic speedup of 18 was achieved with 19 processors with an ideal shared-memory (figure 4). The asymptotic speedup of 17.25 for the low latency network compares well with the ideal case. The advantage of using large line sizes can be clearly seen: large line sizes result in the asymptotic speedup being achieved at a significantly lower number of processors.

The graph for the high latency network is essentially similar in nature, but differs in a number of key respects. The asymptotic speedup is considerably lower, at

https://doi.org/10.1017/S0956796801003793 Published online by Cambridge University Press

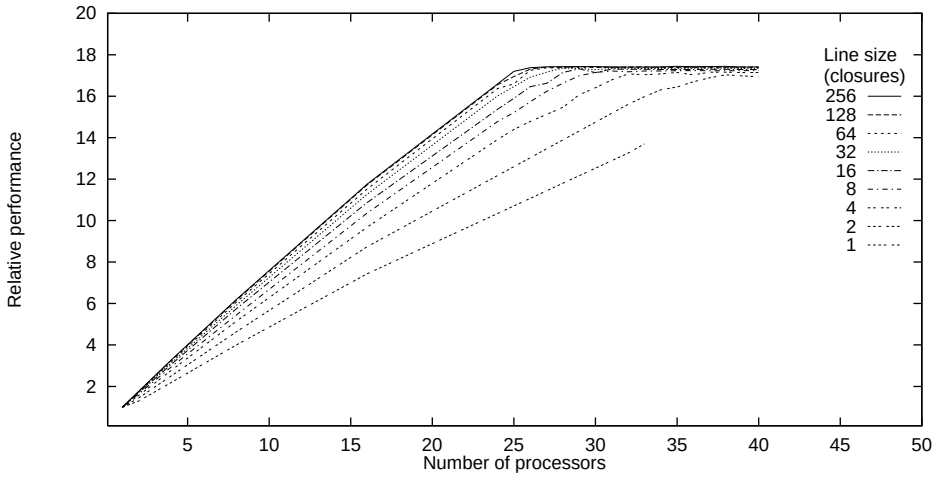

Fig. 13. Relative performance with the TLO protocol and a low latency network.

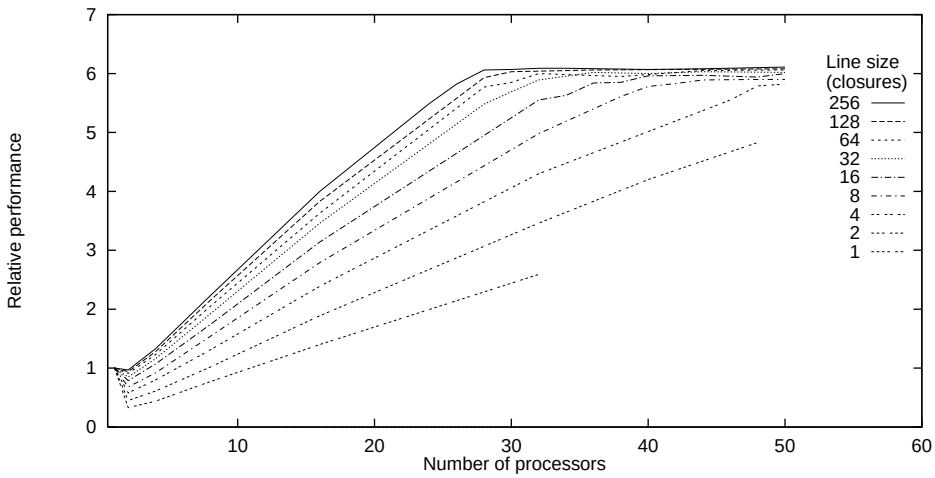

Fig. 14. Relative performance with the TLO protocol and a high latency network.

about 6 – as explained in section 8, the asymptotic speedup is reduced by a factor of $k = 3$. Most obvious from this graph is the reduction in relative performance shown when moving from 1 to 2 PEs. This demonstrates that the cost of communication outweighs the advantage of parallel evaluation resulting from the addition of only a single extra processor.

These results appear to indicate that the largest possible line size is the most effective. This is because we have assumed that the time taken to send a line is independent of its size. This assumption makes results considerably easier to interpret and independent of the characteristics of particular networks. However, the cost of transmitting larger line results in the optimum size being less: the exact value is determined by such factors as the startup time, bandwidth and blocking.

https://doi.org/10.1017/S0956796801003793 Published online by Cambridge University Press

9 Conclusions

A number of conclusions can be drawn from our experience with this case study, some of which, at least, have broader significance:

1. **Program transformation** (sections 3 and 4)
   Two transformations of the initial problem specifications were vital in achieving the desired behaviour:
   - *Speculative execution*: the speculative/optimistic execution needed in this application does not require run-time support, but can instead be implemented using a systematic formal program transformation.
     This manual transformation step improves performance by exposing more, probably-useful, work before construction of the updated tree has to wait for confirmation of the transaction's success. It relies on the programmer's knowledge that most transactions succeed. It also relies on knowing that useful pipeline parallelism is available, and why it is blocked. Profiling tools could help in this respect.
   - *Grain size*: repeated traversals of the tree to `lookup` then `assign` a leaf value were fused into a single `update` function.

   Both of these optimisations exploit the 'transaction' structure we imposed on sequences of tree operations.
   The power of these transformations raises the exciting prospect of applying similar techniques to optimisation of multithreaded software written in conventional languages. Although these transformations are, in some restrictive sense, well-known optimisations for conventional languages – code motion based on branch prediction, and loop fusion – their scope is dramatically increased thanks to the control over side-effects present in a functional language.
2. **Scheduling** (section 6.2)
   In our implementation, a **spark**ed closure can be assigned to any idle processor, but once it has started it cannot be migrated to another processor, even if it blocks. Meanwhile, when a thread blocks and no other local thread is ready to run, the processor selects another **spark**ed closure from the task pool.
   These are often good policies: the first avoids transferring the active thread's stack; the second avoids wasted processor time and avoids deadlock. They lead to very poor behaviour when many tasks in the task pool tend to block before doing useful work.
   In this case, a processor will initiate many threads, all of which will block. This is inefficient, because of the overheads of blocking and unblocking the threads. Much worse, it can lead to a severe load imbalance: when they become unblocked, all the threads must complete on the processor on which they started.
   One solution is to allow migration of blocked threads. This ought to avoid the load imbalance, but at the expense of large overheads due to blocking, restarting and migration. In this case study, we instead identified the cause of the blocking, and observed that blocking will always be momentary, as the node is updated very quickly after being locked. Furthermore, there is no

https://doi.org/10.1017/S0956796801003793 Published online by Cambridge University Press

deadlock hazard because no other nodes need to be evaluated before this can happen.

Our solution, then, is for the processor to 'spin' on the node, rather than seek another task from the pool. The time wasted is bounded and very small.

The choice between spin- vs queue-locks is well-known in the operating systems context. Choosing between the two statically is very hard, in general – the choice may depend on run-time conditions. One approach might be to use some kind of adaptive or competitive scheme (Karlin *et al.*, 1991).

3. **Caching and cache coherency** (section 8.3)

   This application displays substantial spatial locality, which is evident despite the use of random keys in our experiments. By transferring a large block of adjacent nodes in response to each remote memory reference, the total number of remote memory references can be reduced dramatically.

   However, conventional approaches to implementing shared memory fail to exploit this advantage fully because of invalidations resulting from false sharing. We demonstrate that a modified, multicast-free cache coherency protocol avoids this problem.

   Other applications may, of course, have less locality and different update patterns. The actual performance effect will depend both on the application and on the network latency and bandwidth characteristics. Our results show that with sufficiently high bandwidth, 'cache lines' can be several kilobytes (e.g. 256 closures, where our implementation's minimum closure size is 24 bytes). This is similar to the operating system's page size, and suggests a software implementation based on distributed virtual memory might be interesting.

4. **The problems caused by unnecessary updates** (section 8.2)

   We analyse in detail the minimum communication requirements of the algorithm, and compare our model with the performance reported by our implementation. We find that the number of remote reads is close to minimal, but the number of remote updates is higher than expected.

   These unnecessary updates are caused by unnecessary lazy evaluation. The large cache lines transferred on remote reads contain evaluated closures and sparked but unevaluated closures, as expected. Unfortunately, they also contain unevaluated closures which have not been sparked, but are instead demanded later in the computation. When these closures are later demanded, further communication is needed to avoid recomputation.

   The closures are introduced in formulating the application in a lazy functional language. Our optimising compiler uses strictness analysis to avoid the use of closures, employing in-line evaluation or call-by-value wherever possible. Unfortunately, it does not always succeed.

   The most straightforward solution is to use an implementation language which creates no unwanted lazy evaluation. We have built a prototype implementation as a library for parallel C programmers which results in no spurious updates.

5. **Space** (section 8.4)

   The problem of space management is, arguably, the Achilles' heel of parallel

https://doi.org/10.1017/S0956796801003793 Published online by Cambridge University Press

functional and dynamic, multithreaded software. The space required for parallel functional programs can easily increase in proportion to the execution time, even for applications with reasonable behaviour when executed sequentially. This turns out conditionally to be the case for our application when running on a high-latency interconnection network.

One approach is to restrict the structure of the application so that space can be controlled automatically. Data-parallel structures obviously have reasonable space behaviour; more interestingly, the work of Blumofe *et al.* (1995) shows that there is a large class of 'fully strict' dynamic, multithreaded applications which can run in reasonable space ($< S_1P$ per processor on P processors, where S_1 is the space required on one processor).

The application studied in this paper is not 'fully strict', and cannot be made so while retaining its pipelined, parallel structure. For applications like this, experience is needed to identify and solve space problems; our work contributes as a case study. The key issue is not just to control the space utilisation, but to do so without reducing the parallelism.

Further work

This work has raised many issues deserving further investigation:

- The transactions considered above comprise only lookups and assignments, operations that do not alter the structure of the database tree. Insertions and deletions will require rebalancing of the tree, but if this is done using one of several well-known top-down methods, our methods will still be applicable. (The more common bottom-up rebalancing schemes are unsuitable, as they would lock the whole tree until the end of the operation.) For example, insertions may be supported by 2–4-trees using the top-down rebalancing scheme: before entering a node, the algorithm performs such rotations as required to ensure that it is not a 4-node. After this, the parent node may be returned, to be used by any following operation. In this way, imbalance percolates up the tree, one level at a time, until it can be removed. Now suppose the insertion is part of a transaction, with a commitment condition c dependent on other time-consuming operations. As with *assign*, we wish to transform a sub-expression

  if c **then** *insert* x v d **else** d

  into an equivalent expression

  maybe-insert x (**if** c **then** *Yes* v **else** *No*) d

  using a function *maybe-insert* that always proceeds to the leaf, performing such rebalancings as required on the way. The decision of whether to actually perform the insertion is delayed until the leaf is reached. The equivalence here is more lax: all we require is that the two trees represent the same mapping. Of course, such laxer notions of equivalence cannot be used with the *fwif* approach.

https://doi.org/10.1017/S0956796801003793 Published online by Cambridge University Press

- We have not implemented a garbage collector for our simulated parallel graph reduction machine using the TLO protocol. The design of a collector which minimises invalidation traffic is an interesting exercise, both with our TLO protocol and with conventional protocols. Note that, although the TLO protocol avoids invalidations during computation, invalidation is still needed when addresses are reclaimed for re-allocation.
- In the TLO protocol, accesses and updates to a cell are always directed to the processor which allocated the cell: unlike the invalidation protocol, there is no migration of cache line ownership. As is discussed in section 8.1, this may lead to some contention. The significance of this effect needs to be evaluated, in case a more sophisticated scheme (such as 'proxies' (Talbot and Kelly, 1998)) is needed.
- A cache line consists of many cells, and we have assumed that the whole line would be transferred. In fact, there is no need to include cells which have not yet been evaluated. If an unevaluated cell is referred to remotely, a communication is anyway needed to ensure exclusive access. This optimisation is obviously worthwhile only on architectures where the cost of packing a message buffer is small compared with the other costs of message passing.
  An extension to this idea is to pack each message with a logical subgraph rather than a physically-contiguous block of cells. This is the approach taken in GUM (Trinder *et al*., 1996) and evaluated in Loidl and Hammond (1996). Note that this relies on maintaining a separate mapping from a node's global identifier to the local address of the cell within each processor's memory. By marshalling the graph in breadth-first order, GUM should achieve a higher cache hit rate for a given transfer block size.
- This paper concerns simple read-modify-write transactions on a single binary search tree. Transactions which use the result from one lookup as a key in a subsequent operation have different pipeline characteristics, with much greater potential for blocking. Maximising parallelism while minimising computational work would involve combining the approach presented in this paper with conventional relational query optimization.
- Although we have no ambitions to use this case study as a basis for database transaction processing, we are applying ideas from the work in adaptive, tree-structured parallel applications such as particle simulation and adaptive mesh problems, running on large distributed-memory systems. For example, we report a similar application-specific caching mechanism (Wu Qian *et al*., 1997).

Acknowledgements

This work was funded by the UK Engineering and Physical Sciences Research Council under grants GR/J 14448 (*Compaqt*: Combined program and query optimisation for parallel database processing), and GR/J 99117 (Combining randomisation and mixed-policy caching for bounded-contention shared memory). Particular thanks

are due our anonymous referees, and to Simon Peyton Jones, Phil Trinder, Kevin Hammond and Tony Field for their helpful comments on our work.

References

Akerholt, G., Hammond, K., Peyton Jones, S. and Trinder, P. (1993) Processing transactions on GRIP, a parallel graph reducer. In: Bode, A., Reeve, M. and Wolf, G., editors, *PARLE 93 Parallel Architectures and Languages Europe*, Munich, Germany. *Lecture Notes in Computer Science 694*, pp. 634–647. Springer-Verlag.

Archibald, J. and Baer, J.-L. (1986) Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Syst.*, **4**(4), 273–298.

Argo, G., Fairbairn, J., Hughes, R., Launchbury, E. and Trinder, P. (1987) Implementing functional databases. *DBPL'95: Lecture Notes in Computer Science 470*. Springer-Verlag.

Augustsson, L. and Johnsson, T. (1989) Parallel graph reduction with the $\langle v, G\rangle$-machine. *4th International Conference on Functional Programming Languages and Computer Architecture*, London, pp. 202–213.

Barton, E., Cownie, J. and McLaren, M. (1994) Message passing on the Meiko CS-2. *Parallel Computing*, **20**(4):497–507.

Bennett, A. J. (1993) *Parallel graph reduction for shared-memory architectures.* PhD thesis, Department of Computing, Imperial College, London.

Bennett, A. J. and Kelly, P. H. J. (1993) Locality and false sharing in coherent-cache parallel graph reduction. In: Bode, A., Reeve, M. and Wolf, G., editors, *PARLE 93 Parallel Architectures and Languages Europe: Lecture Notes in Computer Science 694*, pp. 329–340. Springer-Verlag.

Bennett, A. J. and Kelly, P. H. J. (1994) Eliminating invalidation in coherent-cache parallel graph reduction. In: Halatsis, C., Maritsas, D., Philokyprou, G. and Theodoridis, S., editors, *PARLE 94 Parallel Architectures and Languages Europe: Lecture Notes in Computer Science 817*, pp. 375–386. Springer-Verlag.

Bennett, A. J. and Kelly, P. H. J. (1997) Efficient shared-memory support for parallel graph reduction. *Future Generation Computer Systems.* **12**(6), 481–503.

Bird, R. S. (1984) The promotion and accumulation strategies in transformational programming. *ACM Trans. Programming Languages and Systems*, **6**(4):487–504.

Blumofe, R. D., Joerg, C. F. ., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H. and Zhou, Y. (1995) Cilk: An efficient multithreaded runtime system. *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

Boyle, J. M., Dritz, K. W., Muralidharan, M. N. and Taylor, R. J. (1987) Deriving sequential and parallel programs from pure LISP specifications by program transformation. In: Meertens, L. G. L. T., editor, *Program Specification and Transformation.* IFIP.

Burstall, R. M. and Darlington, J. (1977) A transformation system for developing recursive programs. *J. ACM*, **24**(1):44–67.

Cox, S., Huang, S.-Y., Kelly, P., Liu, J. and Taylor, F. (1992) An implementation of static process networks. In: Etiemble, D. and Syre, J.-C., editors, *PARLE 92 Parallel Architectures and Languages Europe: Lecture Notes in Computer Science 605*, pp. 497–512. Springer-Verlag.

Darlington, J., Field, A. J., Harrison, P. G., Kelly, P. H. J., Sharp, D. W. N., Wu, Q. and While, R. L. (1993) Parallel programming using skeleton functions. In: Bode, A., Reeve, M. and Wolf, G., editors, *PARLE 93 Parallel Architectures and Languages Europe: Lecture Notes in Computer Science 694*, pp. 146–160. Springer-Verlag.

Dubois, M. (1992) Delayed consistency. In: Dubois, M. and Thakkar, S. S., editors, *Workshop on Scalable Shared Memory Multiprocessors*, Seattle, WA, pp. 207–218. Kluwer Academic.

Friedman, D. P. and Wise, D. S. (1978) A note on conditional expressions. *Commu. ACM*, **21**(11).

Goldberg, B. F. (1988) *Multiprocessor Execution of Functional Programs.* PhD thesis, Yale University, New Haven, CT.

Halstead, R. H. (1984) Implementation of Multilisp: Lisp on a multiprocessor. *ACM Symposium on Lisp and Functional Programming*, Austin, TX, pp. 9–17.

Hammond, K., Loidl, H.-W. and Partridge, A. (1995) Visualising granularity in parallel programs: A graphical winnowing system for haskell. *HPFC'95 – Conference on High Performance Functional Computing*, Denver CO, pp. 208–221.

Harrison, P. G. (1992) A higher-order approach to parallel algorithms. *Computer J.*, **35**(6):555–566.

Hartel, P. H. and Langendoen, K. G. (1993) Benchmarking implementations of lazy functional languages. *Proc. Conference on Functional Programming Langauges and Computer Architecture*, Copenhagen, Denmark.

Hudak, P., Peyton Jones, S. L. and Wadler, P. (1992) Report on the programming language Haskell – a non-strict purely functional language, version 1.2. *SIGPLAN Notices*, **27**(5):1–162.

Karlin, A. R., Li, K., Manasse, M. S. and Owicki, S. (1991) Empirical studies of competitive spinning for a shared-memory multiprocessor. *Proc. 13th ACM Symposium on Operating Systems Principle*, pp. 41–55. Pacific Grove, CA. (Published in *ACM Operating Systems Review*, **25**(5), 1991. Also as Tech report CS-TR-319-91, Princeton University. Department of Computer Science.)

Kelly, P. (1989) *Functional Programming for Loosely-Coupled Multiprocessors.* Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge, MA.

Koopman, P. J., Lee, P. and Siewiorek, D. P. (1992) Cache performance of combinator graph reduction. *ACM Trans. Programming Languages and Systems*, **14**(2):265–297.

Laudon, J. and Lenoski, D. (1997) The SGI Origin: a ccNUMA highly scalable server. *24th Annual International Symposium on Computer Architecture*, Denver, CO. In: *Computer Architecture News*, **25**(2):241–251.

Loidl, H.-W. and Hammond, K. (1996) Making a packet: Cost-effective communication for a parallel graph reducer. *IFL'96 – International Workshop on the Implementation of Functional Languages: Lecture Notes in Computer Science 1268*, pp. 184–199. Springer-Verlag.

Loidl, H.-W., Morgan, R., Trinder, P. and Poria, S. (1998) Parallelising a large functional program or: Keeping LOLITA busy. *IFL'97 – International Workshop on the Implementation of Functional Languages: Lecture Notes in Computer Science 1467*. Springer-Verlag.

Loidl, H.-W. and Trinder, P. (1998) Engineering large parallel functional programs. *IFL'97 – International Workshop on the Implementation of Functional Languages: Lecture Notes in Computer Science 1467*. Springer-Verlag.

Markatos, E. P. and LeBlanc, T. J. (1992) Shared-memory multiprocessor trends and the implications for parallel program performance. Technical Report 420, Computer Science Department, University of Rochester, Rochester, NY.

Peyton Jones, S. L. (1989) Parallel implementations of functional programming languages. *Computer J.*, **32**(2):175–186.

Peyton Jones, S. L., Clack, C., Salkild, J. and Hardie, M. (1987) GRIP – a high performance architecture for parallel graph reduction. In: Kahn, G., editor, *Functional Programming Languages and Computer Architecture: Lecture Notes in Computer Science 274*, pp. 98–112. Springer-Verlag.

Talbot, S. A. M. and Kelly, P. H. J. (1998) Reactive proxies: a flexible protocol extension to reduce ccNUMA node controller contention. *EuroPar'98: Lecture Notes in Computer Science* 1470. Springer-Verlag.

Thakkar, S., Gifford, P. and Fielland, G. (1988) The Balance multiprocessor system. *IEEE Micro*, **8**(1):57–69.

Thekkath, R. (1997) An evaluation of a commercial ccNUMA architecture — the Convex Exemplar SPP1200. *Proc. 11th International Parallel Processing Symposium*. IEEE Press.

Trinder, P. (1989) *A Functional Database*. PhD thesis, Computing Laboratory, Oxford University, Oxford, UK.

Trinder, P. W., Hammond, K., Loidl, H.-W. and Jones, S. L. P. (1998) Algorithm + strategy = parallelism. *J. Functional Programming*, **8**(1):23–60.

Trinder, P. W., Hammond, K., Mattson, J. S., Partridge, A. S. and Jones, S. L. P. (1996) Gum: a portable parallel implementation of haskell. *Proc. ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pp. 79–88.

Wu Qian, Field, A. J. and Kelly, P. H. J. (1997) M-tree: A parallel abstract data type for block-irregular adaptive applications. *EuroPar'97: Lecture Notes in Computer Science 1300*.