Derivation and Performance of a Pipelined Transaction Processor

A. J. Bennett P. H. J. Kelly R. A. Paterson

Department of Computing
Imperial College of Science, Technology and Medicine
London SW7 2BZ

Abstract

Transaction processing can be formulated as a simple functional program operating on a stream of transaction requests and a tree-structured database. In this paper we use algebraic transformation of the initial program to yield an optimistic implementation in which unnecessary synchronization is eliminated, thereby allowing concurrent processing of transactions. A detailed simulation is used to study the program's behaviour and to assess scheduling policies based on the characteristics of the target architecture. Our results show that good speedups can be achieved, and that transformation can be used to derive a highly concurrent program with better locality and grain size.

1 Introduction

Transaction processing is an important application area where the need for high performance has motivated sophisticated parallel implementation techniques. In this paper we present a simple formulation of the problem in a functional style, and use this as a basis for a reconstruction of a fairly sophisticated parallel algorithm. We then give a detailed analysis of its performance using parallel graph reduction (PGR), a dynamically-scheduled implementation scheme. Our use of simulation makes it possible to isolate a number of effects which were obscured in earlier experiments using real hardware [1].

The transaction processing example which forms the subject of this paper has inherently unpredictable sharing behaviour. It also relies heavily on the PGR synchronization mechanism. It is therefore a good example of a PGR application. Indeed we hope our results will be of interest to a wide audience since the techniques are essentially language-independent.

The problems associated with allowing concurrent accesses to a database in a parallel transaction processing system have received much attention. The key problem is to ensure the consistency of data so that the concurrent execution is equivalent to a schedule in which the transactions are executed sequentially [5], and several styles of concurrency control algorithms have been proposed [3]. Using a functional language leads naturally to an approach in which concurrent transactions operate on multiple versions of the database [12]. We begin by reviewing this style of transaction processing, before addressing the issues involved in its parallelization.

2 Functional transaction processing

The problem we consider involves updates to a simple database, comprising a collection of records indexed by a single key. We shall assume that the whole database resides in memory, or that the language supports persistent data structures. The usual implementation of such updates in a conventional database system updates the data structure 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 such a database can be structured as a tree, and the update need only generate new versions of nodes on a path down the tree to the addressed record.

More specifically, using a Haskell style of presentation [9], we define a database as the following tree structure

$$\begin{aligned}\textbf{data}\ DB \triangleq\ & Leaf\ Key\ Value\ \oplus \\ & Node\ DB\ Key\ DB\end{aligned}$$

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:

```
**data** $Maybe\ \alpha \triangleq Yes\ \alpha \oplus No$

$lookup : Key \to DB \to Maybe\ Value$
$lookup\ x\ (Leaf\ k\ v) \triangleq$
    $\textbf{if}\ k = x\ \textbf{then}\ Yes\ v\ \textbf{else}\ No$
$lookup\ x\ (Node\ l\ k\ r) \triangleq$
    $lookup\ x\ (\textbf{if}\ x \leq k\ \textbf{then}\ l\ \textbf{else}\ r)$
```

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

```
$assign : Key \to Value \to DB \to DB$
$assign\ x\ v'\ (Leaf\ k\ v) \triangleq$
    $Leaf\ k\ (\textbf{if}\ k = x\ \textbf{then}\ v'\ \textbf{else}\ v)$
$assign\ x\ v'\ (Node\ l\ k\ r) \triangleq$
    $\textbf{if}\ x \leq k$
    $\textbf{then}\ Node\ (assign\ x\ v'\ l)\ k\ r$
    $\textbf{else}\ Node\ l\ k\ (assign\ x\ v'\ r)$
```

As noted above, only nodes on the path down to the addressed leaf are updated.

We shall not consider updates that change the shape of the tree, like insertions and deletions. However, there are standard schemes for such operations which perform top-down rebalancing of the tree, and we expect that they could be added without greatly complicating our analysis.

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.
- 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.

A more realistic application might group a number of *lookup* and *update* operations 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:

```
$apply : Trans \to DB \to DB$
$apply\ t\ db_0 \triangleq$
    $\textbf{let}\ v_1 \triangleq lookup\ x_1\ db_0\ \textbf{in}$
    ...
    $\textbf{let}\ v_5 \triangleq lookup\ x_5\ db_0\ \textbf{in}$
    $\textbf{let}\ all\text{-}ok \triangleq ok\ v_1 \wedge \ldots \wedge ok\ v_5\ \textbf{in}$
    $\textbf{if}\ all\text{-}ok \wedge c$
    $\textbf{then}$
        $\textbf{let}\ db_1 \triangleq assign\ x_1\ e_1\ db_0\ \textbf{in}$
        ...
        $\textbf{let}\ db_5 \triangleq assign\ x_5\ e_5\ db_4\ \textbf{in}$
        $db_5$
    $\textbf{else}\ db_0$
```

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

```
$ok : Maybe\ \alpha \to Bool$
$ok\ (Yes\ x) \triangleq True$
$ok\ No \triangleq False$
```

This sort of global rollback is neatly supported by a functional formulation [12]: 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 [12, 1] proposed that the **if** in the above program be replaced with a conditional proposed by Friedman and Wise [6]. 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. This approach produces a large number of short-lived threads, each responsible for evaluating a single node of the tree. Scheduling is thus considerably simplified, at the cost of greater overheads. In the context of the above program, many of the threads do no useful work, since all the altered nodes are concentrated on a path down the tree. Hence the implementation of this conditional requires *ad hoc* treatment of a number of special cases, some of them specific to this program, to achieve reasonable behaviour [1]. Even so, the overheads of thread creation and the costs of the resulting loss of locality may be excessive.

An alternative to introducing fine-grain parallelism and relying on general-purpose scheduling strategies, is to transform an initial program into one 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.

Removing synchronization amounts to pushing a conditional down into its branches. In the current case, we wish to apply such a transformation to a subexpression of the form

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

That is, we require a function

```
    $\mathit{maybe\text{-}assign} : \mathit{Key} \rightarrow \mathit{Maybe\ \ Value} \rightarrow$
        $\mathit{DB} \rightarrow \mathit{DB}$
```

such that for all fully defined b, x and d,

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

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

```
    $\mathit{decide} : \mathit{Trans} \rightarrow$
        $\mathit{Maybe\ \ Value} \rightarrow \ldots \rightarrow \mathit{Maybe\ \ Value} \rightarrow$
        $(\mathit{Maybe\ \ Value} \times \ldots \times \mathit{Maybe\ \ Value})$
    $\mathit{decide}\ t\ u_1\ \ldots\ u_5 =$
        $\textbf{let } \mathit{all\text{-}ok} \triangleq \mathit{ok}\ u_1 \wedge \ldots \wedge \mathit{ok}\ u_5 \textbf{ in}$
            $\textbf{if } \mathit{all\text{-}ok} \wedge c$
            $\textbf{then } (\mathit{Yes}\ e_1, \ldots, \mathit{Yes}\ e_5)$
            $\textbf{else } (\mathit{No}, \ldots, \mathit{No})$

    $\mathit{apply}\ t\ db_0 \triangleq$
        $\textbf{let } u_1 \triangleq \mathit{lookup}\ x_1\ db_0 \textbf{ in}$
        $\ldots$
        $\textbf{let } u_5 \triangleq \mathit{lookup}\ x_5\ db_0 \textbf{ in}$
        $\textbf{let } (u'_1, \ldots, u'_5) \triangleq \mathit{decide}\ t\ u_1\ \ldots\ u_5 \textbf{ in}$
        $\textbf{let } db_1 \triangleq \mathit{maybe\text{-}assign}\ x_1\ u'_1\ db_0 \textbf{ in}$
        $\ldots$
        $\textbf{let } db_5 \triangleq \mathit{maybe\text{-}assign}\ x_5\ u'_5\ db_4 \textbf{ in}$
        $db_5$
```

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 equation by induction over the data structure. In the process, the appropriate definition of the function will often emerge, as happens in this case: the proof requires the following definition of *maybe-assign*:

```
    $\textbf{or} : \mathit{Maybe}\ \alpha \rightarrow \alpha \rightarrow \alpha$
    $\mathit{Yes}\ x \textbf{ or } y \triangleq x$
    $\mathit{No} \textbf{ or } y \triangleq y$

    $\mathit{maybe\text{-}assign}\ x\ u\ (\mathit{Leaf}\ k\ v) \triangleq$
        $\mathit{Leaf}\ k\ (\textbf{if } k = x \textbf{ then } (u \textbf{ or } v) \textbf{ else } v)$
    $\mathit{maybe\text{-}assign}\ x\ u\ (\mathit{Node}\ l\ k\ r) \triangleq$
        $\textbf{if } x \leq k$
        $\textbf{then } \mathit{Node}\ (\mathit{maybe\text{-}assign}\ x\ u\ l)\ k\ r$
        $\textbf{else } \mathit{Node}\ l\ k\ (\mathit{maybe\text{-}assign}\ x\ u\ r)$
```

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

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.

A well-known transformation [4] 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 : Key → Maybe Value → DB →
        Maybe Value × DB
update k v' d ≜
        (lookup k d, maybe-assign k v' d)
```

This definition is expanded by cases (unfolding) and recursive calls are discovered (folding), giving a revised definition for the new function:

```
update x u' (Leaf k v) ≜
        (u, Leaf k v')
        where (u, v') ≜
                if k = x then (Yes v, 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
```

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* threads 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 $i = 2, \ldots, 5$. Thus we can restate the transaction processor as follows:

```
apply t db₀ ≜
        let
                (u₁, db₁) ≜ update x₁ u'₁ db₀
                ...
                (u₅, db₅) ≜ update x₅ u'₅ db₄
                (u'₁, ..., u'₅) ≜ decide t u₁ ... u₅
        in
        db₅
```

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 on u'_i is the data value in one leaf.

4 Simulation of parallel graph reduction

In order to evaluate the parallel performance of the new form of the transaction processor, its behaviour has been studied on an implementation of parallel graph reduction. This in turn runs on an execution-driven simulation of a shared-memory multiprocessor. The advantage of using simulation is that is allows the behaviour of the system to be closely monitored without affecting its behaviour.

PGR supports the "call-by-need" parameter passing mechanism required by lazy functional languages by using "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 are used in several places in the new formulation of the transaction processor. In the definition of the *apply* function above, each call of *update* is packaged into a closure which is then sparked (*i.e.* added to a pool of available tasks for distribution to other processing elements). 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.

PGR is inherently a dynamically scheduled scheme, and mechanisms are required to distribute sparked tasks around the machine. We have adopted a work stealing scheme in which each processor has a local task queue to which tasks it sparks are added and from which it fetches new work when necessary. Only when a processor's local queue is empty will it attempt to find work from another processor.

4.1 Simulated architecture

A simple model of a shared-memory parallel architecture has been adopted: the system consists of a set of processors, each tightly coupled to a large cache. The PEs are interconnected by network. 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 a latency of one processor cycle. Note that 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.

Two different memory models are used: a perfect shared-memory is simulated in the next section in order to determine ideal speedup times and to study the behaviour of the program without it being disturbed by network delays. A more realistic multicache implementation of shared-memory is used later in order to assess scheduling policies appropriate for a real machine.

5 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 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. By sparking the first component of the result of the function *update*, we ensure that a single thread evaluates the *update* all the way down to the leaf.

For our experiments, we used the following parameters:

- Size of the database: 10000 consecutively numbered records. This size was adopted in order to produce a reasonable execution time for each simulation run.

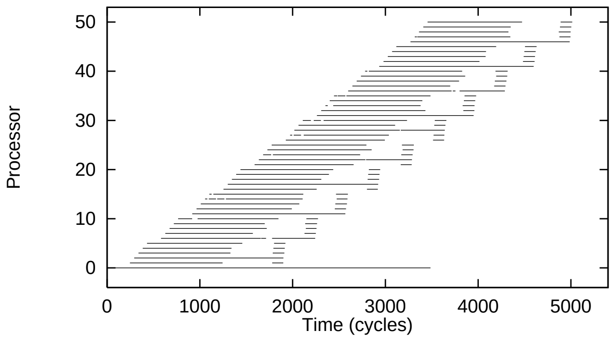

Figure 1: Activity on a 64-node Machine

- Initial distribution of the database: the entire database is divided into subsets of consecutively numbered databases which are allocated to individual PEs.
- Number and size of transactions: 10 transactions each operating on 5 randomly selected records.
- No disks are simulated, *i.e.* the database is entirely RAM-resident.

The concurrency of the program is illustrated by Figure 1, showing the activity of each processor when the program was run on our simulator, using a large number of processors and an ideal memory model to give a simple picture. As can be seen from the figure, the master thread spawns 5 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, if two successive transactions require the same record, the evaluation of *decide* in the second will block until the value is updated.

From Figure 1 we can also predict 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

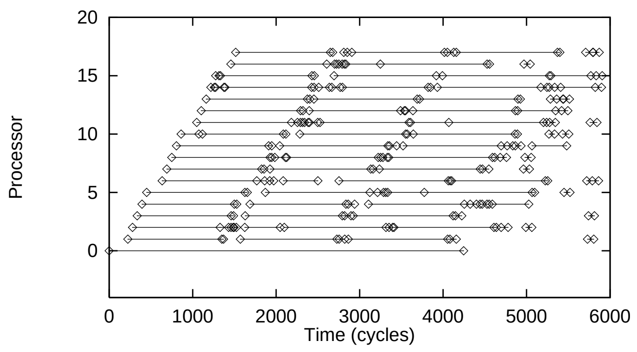

Figure 2: Activity and State Transitions on a 18-node Machine

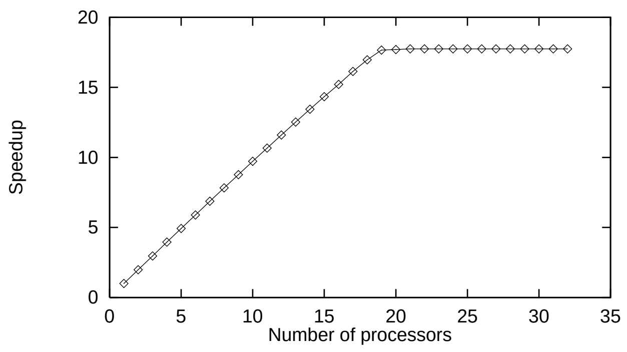

Figure 3: Relative speedup on an ideal shared-memory

of the leaf data can be done later; only another *update* requiring the same leaf will be blocked. On the other hand, if the transient blocks higher in the tree (usually near the root) are treated in the same way, any new thread taken up will be likely to block quickly, so that the processor will accumulate 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.

Possible solutions to this scheduling problem include timesharing between runnable threads, and thread migration. Both are expensive, and neither is a complete solution. However, it is apparent from the program that

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

Thus we have introduced a new annotation, indicating that if a thread blocks waiting for value of the annotated expression, the processor should not schedule extra work, as the value will soon be available. 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 [7].

With this modification there are some momentary delays high in the tree, but otherwise the processors are almost completely utilized. For example, Figure 2 shows activity and state transitions on a 18-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 3 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 uneven finish times seen in Figure 2. As one might expect from Figure 1, 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 [12]. For our sample tree of 10,000 nodes, the asymptotic speedup is 18.

6 Program behaviour with a multi-cache shared-memory

In a real shared-memory parallel machine, memory accesses which require use of the network incur latency. We have simulated a multicache implementation of shared-memory, since our previous experiments have shown that programs running under PGR exhibit considerable locality of reference, which can be exploited by a caching scheme.We have used a new PGR-specific coherency protocol [2] that exploits locality to a greater extent than a conventional invalidation coherency protocol by taking advantage of the disciplined memory reference characteristics of PGR. Two memory timing models are used with this protocol: the low-latency model represents a first generation shared-memory multiprocessor such as the Symmetry [10], in which the latency of a shared-memory access which requires use of the network is 10 times that of an access which can be satisfied by the local cache, whereas the high-latency model represents a more

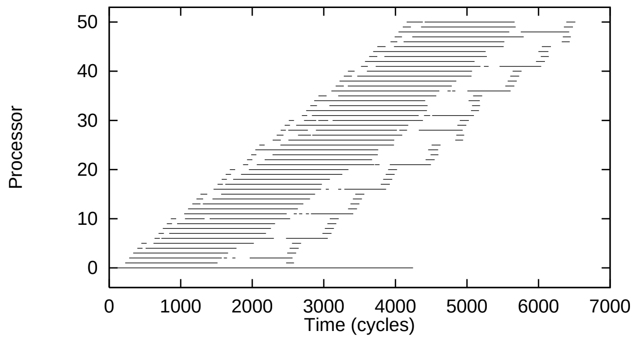

Figure 4: Activity on a low-latency network

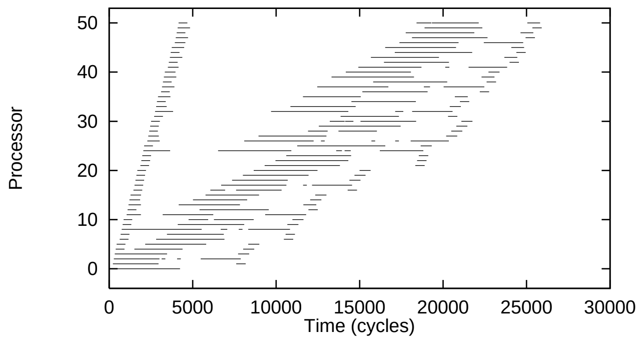

Figure 5: Activity on a high-latency network

modern multiprocessor, in which the remote:local ratio is an order of magnitude larger.

Our program has a great deal of locality within threads, the arguments and results of each call being essentially local data. The nodes of the tree must be shared between threads. However, each *update* thread generates new nodes along a path from the root, so there is a 50% probability that each node is stored near its parent.

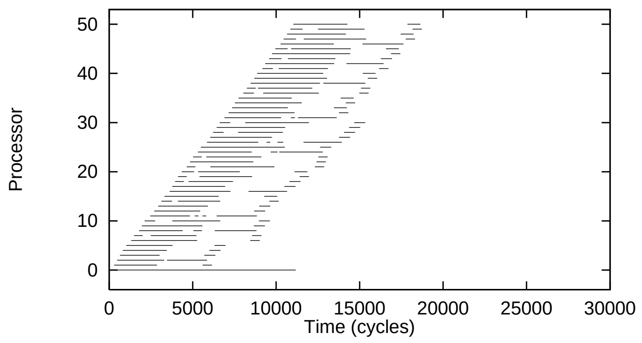

Figure 6: Activity on a high-latency network with delayed **spark**s

The behaviour of our program on a low-latency network is shown in Figure 4. As expected, everything takes longer, but the overall shape of the computation is much as before. Note, however, that the transient delays that occur near the top of the tree are becoming longer. If the network latency is greatly increased, we have the situation of Figure 5. The slope of the leading edge, the thread creations, is identical with that in Figure 4, but the delays have become a bottleneck. In fact, there are two bottlenecks: the thread resumptions form two lines as the threads queue for the use of the two tree nodes at the second level in the tree.

The solution is to move the **spark**s further down the tree, effectively broadening the root node. This has two effects: firstly, the work done by processor 0 on each *update* increases, a throttling effect decreasing the slope of the leading edge in the figure. Secondly, 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, then the average inter-arrival time in each queue is $k2^kt_n$, while the service time is $l + t_n$. To achieve a stable system, we must have

$$k2^kt_n > l + t_n$$

For the network parameters reflected in Figure 5, a choice of $k = 3$ achieves this balance, leading to the behaviour seen in Figure 6. The processors are used for shorter periods, and the overall time has also improved. This may be taken further, using standard queueing theory to obtain the expected time spent in the queue, and hence an estimated time for the update.

7 Summary and conclusions

Transaction processing can be formulated as a simple functional program operating on a stream of transaction requests and a tree-structured database. Meaning-preserving transformations have been used to derive a more efficient form which reduces the number of traversals of the tree, and offers better parallel performance by allowing multiple transactions to be in progress simultaneously. The performance and behaviour of this new form of the program has been evaluated using a simulation of a shared-memory multiprocessor. The ability to monitor the behaviour of the system in detail without affecting its behaviour has allowed us to study scheduling and locking issues in detail.

Our results include the following:

- Significant algorithmic parallelism is available.
- Transformation leads to a form of the program with the same level of concurrency as the previously published version [1], but with better grain size and locality.
- A new annotation was found to be necessary for the optimum scheduling of pipelines. The underlying analysis is similar to that involved in the generation of serial combinators [7].
- Contention for the upper levels of the tree becomes significant when network latency is high, resulting in unnecessary sequentialization. This contention can be reduced by increasing the size of the root of the tree (*i.e.* by moving sparks down the tree).
- A simple queueing model can be used to determine the optimum level to place spark annotations.

Future work will include modifying the program to more closely match the processing part of the debit-credit benchmark [11], and a detailed study of the interaction between the program and multicache shared-memory systems. Also we plan to provide the future [8] construct (very similar to a closure) to C programmers via a library in order to allow the program to be expressed in an imperative language, and we have targetted an initial implementation to the Fujitsu AP1000. Other ideas include the study of other shared coherent abstract data types which can be used for general-purpose portable parallel programming in conventional languages.

Acknowledgements

This work was funded by the U.K. Engineering and Physical Sciences Research Council under grant number GR/J 14448 (*Compaqt*: Combined program and query optimization for parallel database processing).

References

[1] G. Akerholt, K. Hammond, S. Peyton Jones, and P. Trinder. Processing transactions on GRIP, a parallel graph reducer. In Arndt Bode, Mike Reeve, and Gottfried Wolf, editors, *PARLE 93 Parallel Architectures and Languages Europe, Munich, June 1993*, volume 694 of *Lecture Notes in Computer Science*, pages 634–647, Berlin, 1993. Springer-Verlag.

[2] Andrew J. Bennett and Paul H. J. Kelly. Eliminating invalidation in coherent-cache parallel graph reduction. In C. Halatsis, D. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *PARLE 94 Parallel Architectures and Languages Europe, Athens, July 1994*, volume 817 of *Lecture Notes in Computer Science*, pages 375–386, Berlin, 1994. Springer-Verlag.

[3] P. Bernstein and N. Goodman. On concurrency control in distributed database systems. *Computing Surveys*, 13(2):185–221, June 1981.

[4] Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[5] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[6] D. P. Friedman and D. S. Wise. A note on conditional expressions. *Communications of the ACM*, 21(11), November 1978.

[7] Benjamin F. Goldberg. *Multiprocessor Execution of Functional Programs.* PhD thesis, Yale University, New Haven, 1988.

[8] Robert H. Halstead. Implementation of Multilisp: Lisp on a multiprocessor. In *1984 ACM Symposium on Lisp and Functional Programming, Austin, August*, pages 9–17, 1984.

[9] Paul Hudak, Simon L. Peyton Jones, and Philip Wadler. Report on the programming language Haskell — a non-strict purely functional language, version 1.2. *SIGPLAN Notices*, 27(5):1–162, May 1992.

[10] Tom Lovett and Shreekant Thakkar. The Symmetry multiprocessor system. In *1988 International Conference on Parallel Processing, Pennsylvania, August*, pages 303–310, 1988.

[11] Transaction Processing Performance Council (TPC). TPC Benchmark, a standard. Technical report, ITOM International, Los Altos, CA, 1989.

[12] Phil Trinder. *A Functional Database.* PhD thesis, Computing Laboratory, Oxford University, Oxford, U.K., 1989.