Futurespace: a coherent, cached, shared abstract data type

Paul H J Kelly
Manor Askenazi
Andrew J Bennett
John Darlington
Imperial College of Science, Technology and Medicine *

October 20, 1993

Introduction

Shared memory is popular among parallel applications programmers because it makes it easy to build shared data structures, avoiding the problems of managing placement and consistent replication of data. It is generally implemented using multiple caches under a coherency protocol which ensures a guaranteed level of consistency between different processors’ views of the common store. In large configurations such systems are unattractive because of their erratic performance and poor latency hiding.

Well-structured programs use abstract data types (ADTs) to structure access to memory. The work presented here concerns the implementation of tools for parallel programming which implement shared ADTs, instead of shared memory.

We present

- simulation results which demonstrate that there is potential for very large performance benefits from sharing ADTs instead of RAM.
- a prototype implementation of an example ADT, called Futurespace, which demonstrates how the idea can be used for general-purpose portable parallel programming in conventional languages.

Simulation results

This work originated with recent research on caching mechanisms to support parallel functional program execution ([Ben93], [BK93]). The simulation studies undertaken compared the performance of a suite of parallel functional benchmark programs using three different implementations of shared memory: an ideal shared memory (CRCWPRAM), a multicache under a conventional directory-based invalidation coherency protocol (similar to that used, for example, in the DASH prototype [LLG+91]), and a multicache system using a coherency scheme which takes advantage of the specific needs of the parallel graph reduction model, most notably referentially transparent structures.

*Inquiries to Paul Kelly, phjk@doc.ic.ac.uk, Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ, UK.
The study concentrated on architectures with an interconnection network which offers very high bandwidth, but which imposes a substantial latency on each message. Current and likely future large multi-computers fall into this class. As expected, the study showed that because of spatial locality, communication costs could be dramatically reduced by using very large cache lines. Unfortunately, with the conventional protocol, this benefit is very limited because contention for write access to cache lines grows rapidly with their size. (The key point about parallel graph reduction is that a node of a reduction graph cannot be overwritten with inconsistent values.) Bennett’s improved protocol avoids this “serialisation” effect: the number of messages required falls monotonically as the line size is increased, reflecting the increased spatial locality which can be successfully exploited with the new scheme. This is illustrated in Figure 1.

**Futurespace**

In order to make the technique more accessible to mainstream (e.g. C and FORTRAN 90) programmers we designed a portable library which provides the same service. We have very recently completed a prototype implementation of a very simple library which maintains a shared heap of “futures”. A future is constructed with the call

```c
mkfuture(fun, paramsize, paramp, resultsize)
```

The parameter `fun` specifies a function to be applied to the parameter structure specified. This is copied into “futurespace”, where an idle processor will hopefully evaluate the function and overwrite the future with the result. Any processor can then call `getfuture` to wait until the result is available and return it. The other parameters define the storage requirements of the argument and result components. Our current implementation runs on a Fujitsu AP1000 and is being evaluated with benchmarks from Bennett’s suite and also with a transaction processing example which is described below.

The key idea in the implementation is that a future is accessed *only* using the block-
ing \texttt{getfuture} operation, and once a future is evaluated it will not be changed until the space is reallocated. We therefore can avoid the two main problems in shared memory implementations: there is no invalidation, so once a processor has acquired an evaluated copy of a future it can use it freely, and there is no oversynchronisation: a thread is delayed waiting only for the arrival of the contents of the specific future it has requested. This is to be contrasted with a general shared memory system where a process must wait for all write operations to complete (at least when synchronising) to ensure coherency.

These properties make it attractive to employ large cache lines – that is, to piggyback additional evaluated closures in the messages which are exchanged in just the same way that traditional cache lines may hold several cached memory blocks. Bennett’s simulations showed that in some applications spatial locality leads to a very significant performance benefit. The important point is that doing this does not result in the risk of having to issue invalidations of the piggy-backed data, as would be the case in a general-purpose coherency protocol.

\textbf{Example: pipelined tree update}

As part of our pilot study we have investigated a transaction processing example inspired by the work of Akerholt, Trinder et al. [AHPT93] on a pipelined B-tree approach using Haskell on the GRIP machine.

The B-tree is represented in futurespace with futures as the nodes. An update to the tree proceeds as usual, traversing the path from the root to the selected leaf using \texttt{getfuture} to establish the state of each intermediate node, so blocking until any preceding update is complete. The update begins by creating a new future representing the root of the updated database. When this is evaluated it is overwritten with pointers to the subnodes. Normally most of these are pointers to nodes from the original database. If a subnode might have to be changed, it is represented by a new future representing the updated subtree. Thus each node is updated as soon as its value is known, and so a later update can use it as soon as possible. This is illustrated in Figure 2. The result is that several updates can be in progress simultaneously, blocking only when they refer to the same node. As is shown in Figure 3,

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{tree_update.png}
\caption{An update to produce a third database need wait only for the root node of Database 2 to be produced}
\end{figure}

our prototype implementation gives a modest speedup despite an extremely naïve implementation. Obviously we can improve performance by using a hybrid data structure with hash tables at each node. A large database would probably be mainly disk resident, leading to the idea that there might be opportunities for paging policies tuned to the future ADT.

\textbf{Summary}

We have introduced the idea of shared abstract data instead of shared RAM. We presented simulation results indicating the possible benefits for a “future” ADT,
and showed some preliminary results from an implementation running on a Fujitsu AP1000.

Acknowledgements

This work was made possible by the provision by Fujitsu of access to their AP1000 parallel processing facilities. Dr Bennett was supported by a SERC research studentship for which we also offer thanks.

References


