GRAPHing the Future

Kevin Hammond Hans Wolfgang Loidl Jim Mattson
Andrew Partridge Simon Peyton Jones Phil Trinder *

Glasgow University
Email: {kh,hwloidl,mattson,ap,simonpj,trinder}@dcs.glasgow.ac.uk

Abstract

At Glasgow our research into parallel functional programming has been moving away from our novel architecture, GRIP towards the provision of a general parallel runtime environment. We call this GRAPH (Graph Reduction for an Assortment of Parallel Hardware).

This paper describes the design of a new memory and load management model for GRAPH, which is intended to match shared- and distributed-memory machines better and to provide a framework for our research into parallel functional databases and granularity control. This model is currently under implementation. No performance results can therefore be provided at the present time.

1 Introduction

Research into parallel functional programming at Glasgow has for some time been centred around the novel GRIP architecture, a specialised graph reduction machine which was designed and built at University College, London by Peyton Jones et al [5].

While this machine has proved useful as a basis to explore many aspects of real-machine functional parallelism, including advanced dynamic load management techniques. It is clear that for our work to be generally useful a more portable, less specialised environment is required. This is also mandated by the age of the GRIP design: although the GRIP processors were state-of-the-art when the machine was designed, the pace of advance in computer architecture means that a single modern RISC processor can outperform even half-a-dozen of GRIP's processors, and a modern commercial parallel machine has the potential to significantly outperform GRIP.

In a companion paper [4] we describe our plans for a portable parallel runtime system based on the Glasgow Haskell compiler, which we call GRAPH (Graph Reduction for an Assortment of Parallel Hardware). The same paper also describes the design and implementation of GRAPH for PVM, a generic message-passing communication harness. This is intended as a first cut towards our ultimate parallel system, and is aimed to gain experience with more distributed

*This work is supported by the EPSRC AQUA and PARADE projects, by an SOED Research Fellowship from the Royal Society of Edinburgh, and by a Kurt Gödel Scholarship and a KIP scholarship

machines, while retaining much of the GRAPH for GRIP design. In particular, to ease the implementation route, that design is built around an emulation of the GRIP Intelligent Memory Units. This is not necessarily an ideal model for other parallel machine designs.

This paper builds on our experience with the GRAPH for PVM design, and describes a new *unified* memory model which eliminates the need for separate intelligent memory units, while retaining many of those aspects of their functionality which have been found to be important in the GRIP environment. In particular this model retains the logical two-level heap which was present in GRIP, while eliminating much of the associated overhead and complexity by combining IMU and PE operations in a single logical process. We call this system GRAPH for UMM (Unified Memory Model).

The use of a two-level heap is important (even in a shared-memory machine) because most garbage is local and can be collected independently by a PE, i.e. in parallel with other PEs performing reduction. The model also allows us to manage duplicate copies of shared data structures without running the risk of excessive heap usage. It is highly desirable to retain local copies of data structures (and even perhaps to recompute the functions which produce them), but if not managed carefully, memory consumption may be excessive due to the presence of multiple copies of a single data structure. The existence of a single master copy of each shared graph node allows duplicate copies to be destroyed and replaced by pointers to the single master copy whenever necessary.

We also plan to retain non-preemptive unfair scheduling. Compared with conventional round-robin scheduling, this has the advantage of reducing task-switching and memory overheads. It also has the prospect of improving average-case performance while not worsening worst-case performance. In order to avoid some pathological behaviour with coarse-grained tasks, task migration is necessary [3].

Our experience with the simple sparking mechanisms provided with GRAPH for GRIP has been remarkably positive, due largely to the sophisticated dynamic load management system which we have developed over the years, and we intend to retain this basic model. In order to cope with improved compilation techniques such as the granularity analysis we are developing, and to allow experimentation with data and task placement for our database application, however, we will need some extensions to the basic GRIP `par` and `seq`.

The remainder of this paper is structured as follows. Section 2 describes our target architecture. Section 3 describes the distribution of data in GRAPH. Section 4 describes load management. Section 5 describes parallel garbage collection. Section 6 considers the demands which the PARADE database engine places on this design. Section 7 concludes with a summary of the design.

2 Target Architecture

GRIP is a virtual shared-memory machine built from a variety of components connected by two levels of bus through a fast packet-switching network. The most important components are the Processing Elements (PEs), which perform independent graph reduction and which have their own RAM, and the Intelligent Memory Units (IMUs), which provide access to globally addressed shared heap and to the shared spark pools.

In many respects, GRIP has the feel of a distributed machine: communication is through message passing rather than normal memory accesses, and the relative costs to the PE of local versus global memory accesses are closer to those for distributed-memory than shared-memory machines. However, GRIP has the characteristics of a shared-memory machine in that the communication network is insensitive to processor locality, and in that the design is not scalable (beyond adding more levels of bus structure). The original design allowed for up to 20 boards of 4 PEs and 1 IMU each.

Our primary target architecture is a distributed-memory machine such as the ICL Goldrush or Fujitsu AP1000. Compared with GRIP, these machines have fast processors, but as a consequence the communication systems are relatively slow, and the overheads are relatively high. For these machines it is vital to limit the frequency with which messages are transmitted and to maximise the information transmitted in each packet so as to minimise overall execution time. Importantly for our work on parallel functional databases, these machines also have integral disks.

We do not want to limit our attention to distributed-memory machines, however. We also have in mind shared-memory targets such as the multiprocessor Suns which are now becoming common and even loosely coupled systems such as networks of workstations.

Eliminating even logical IMUs from the design avoids the need for separate processes to handle global memory accesses. It is still possible to exploit communication processing hardware if it is present, though, by separating out the message-handling component of the GRAPH system. In this paper, we will ignore the possibility that communications processors are present in order to simplify the description. It should be remembered, however, that in order to reduce the load on the main processors, we do aim to exploit such hardware if it is available.

We will not elaborate these descriptions or our general goals for GRAPH here. More detail can be found in the companion paper in this volume [4].

3 Distributing Data

In GRAPH for GRIP every object has a global address (GA), which occupies one word. Since the average size of an object in the sequential world is 3–4 words, this implies a space overhead of 20–25%. For local objects (which we hope are in a majority), this address field is held blank and thus represents wasted space.

In GRAPH for UMM, we instead maintain separate tables of incoming and outgoing global pointers, with weighted reference counts. Incoming global pointers represent nodes which are owned by the local PE. Outgoing pointers represent references to nodes which are owned by other PEs. The use of weighted reference counting [2] should allow most global nodes to be garbage collected without needing to halt the machine.

3.1 Global Addresses

In GRAPH for UMM, a GA is a pair of a PE number and a unique address on that PE. The PE indicated in the GA holds the *master* copy of the object referenced by the GA. The unique address is an index to an entry in a table of incoming pointers. Each entry records the local address of the closure which represents the global object. This is shown in Figure 3. The primary function of this table is to permit a PE to use copying garbage collection schemes without needing to inform other PEs of the new location of the closure.

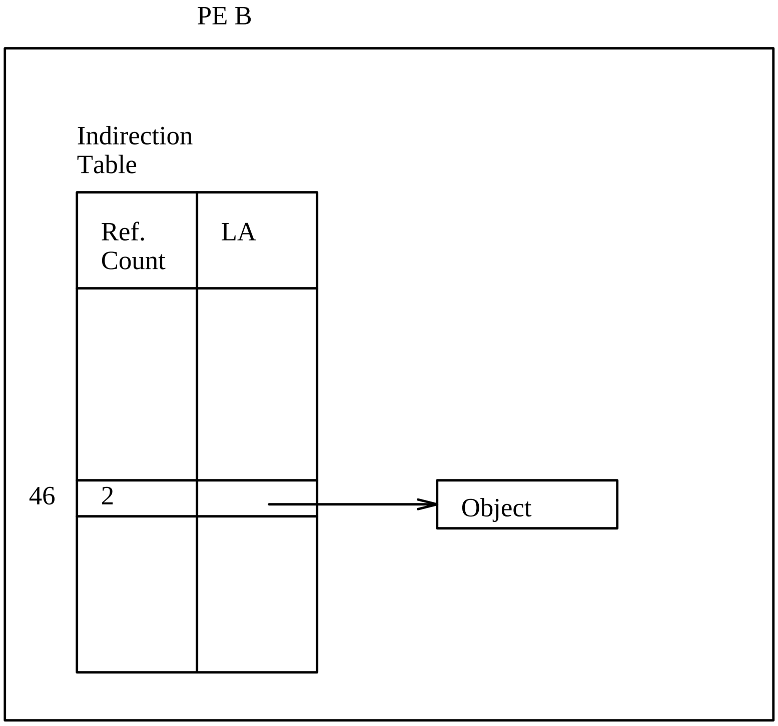

Figure 1: Indirection Table Entry for Global Address B.46

Each PE also maintains two hash tables. The first table maps local addresses to global addresses (LA $\rightarrow$ GA). If an object has an entry in this table it is a global object. This is used to guarantee that a local object has at most one GA, and to keep track of global pointers to other PEs for global garbage collection purposes. The second table maps GAs to LAs (GA $\rightarrow$ LA). This can be used to avoid making multiple local copies of a remote objects and hence preserves shared sub-graphs if there are several local references to the object.

3.2 Fetching Data

A task executing on a PE may enter a node which has a global address, but no value. Such nodes have a special *Fetch-Me* info-pointer, and contain the GA of the master copy of the object. If there is already a copy of the object on the local PE (if the PE component of the GA is this PE, or there is an entry in the PE's GA $\rightarrow$ LA table), then the *Fetch-Me* is replaced by an indirection to the existing copy. Otherwise, a copy of the node must be requested from the PE which owns the master copy. The current PE issues a FETCH message to this PE, and the current thread then becomes blocked pending a reply.

The action of the owner when it receives the FETCH message depends on the nature of the requested object. If the object is

- **A Fetch-Me:** The FETCH is invisibly forwarded to the owner of the corresponding global node.

- **In WHNF:** The object is returned to the PE which requested it in a RESUME message. Some "nearby" graph is also packaged into the RESUME as described in Section 3.3.

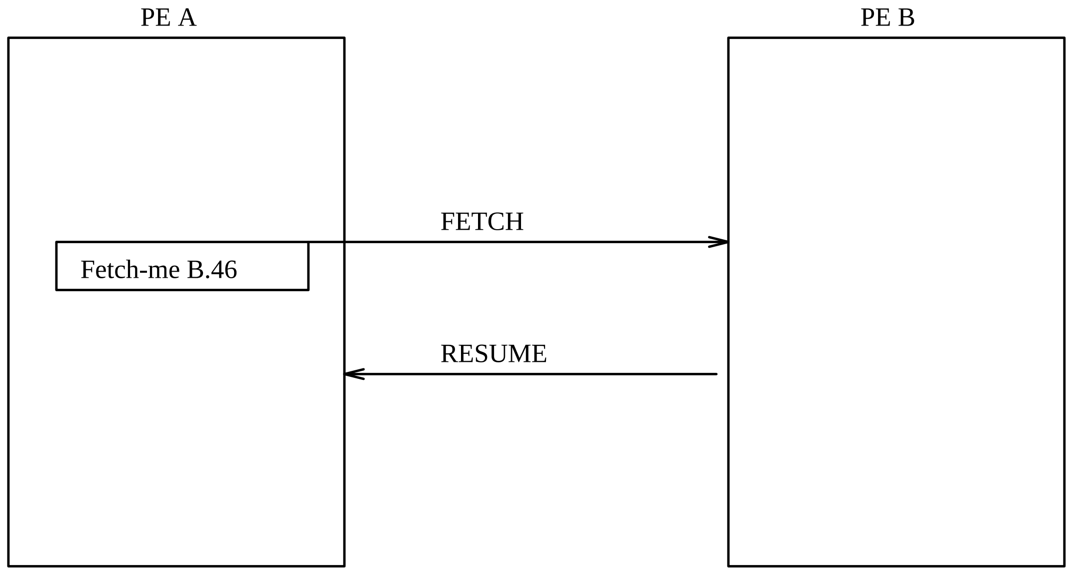

Figure 2: Fetching an Object

- **Being Evaluated:** The FETCH is recorded and a RESUME message is sent when the evaluation is complete.

- **Unevaluated:** If the PE which requested the node is lightly loaded and the owner is heavily loaded then a RESUME is issued as if the node was in WHNF, and the node becomes a *Fetch-ME*. Otherwise the owner evaluates the closure and sends a RESUME once the closure is in WHNF.

On receipt of a RESUME, the requesting PE unpacks the data, updates its global address tables and makes the requesting task runnable again. The message sequence is depicted in Figure 2.

3.3 Packaging

In GRAPH for GRIP, only the closure actually requested is sent in response to a FETCH. This would be extremely expensive on a distributed machine, where messages are subject to a long transmission delay. A standard technique to reduce the impact of high latency is to increase message size, for example, by sending ten 1000-byte messages instead of a thousand 10-byte messages.

In GRAPH for PVM, all of the graph reachable from the object being flushed must be made global and flushed. This is necessary because, to permit independent local garbage collection, an IMU cannot retain pointers into a PE's local memory. This strategy introduces some complications, as described in [4]. The IMU must return a reply packet with the requisite number of global addresses and until it does the flushed graph is in a special flush-pending state.

The global addresses in GRAPH for UMM ships allow us to explore intermediate packaging strategies. Graph *near* to the requested object is also shipped. Near means the objects located by a breadth-first search of the of the graph. The search stops once a standard packet has been filled or there are no more appropriate objects to send. The graph is copied with care to avoid unfolding cyclic structures, as with GRAPH for PVM.

It is not sensible to send every kind of object in a RESUME. In the current design Fetch-Mes, evaluated closures and unevaluated, but unshared, closures are all shipped. Potentially-shared unevaluated closures and closures under evaluation are not shipped. An alternative under consideration is always to ship at least one unevaluated closure. This alternative requires the PE issuing the FETCH to allocate and send an indirection-table entry for the closure.

4 Load Management

The previous section described one instance where work is distributed between PEs, namely, when a lightly-loaded PE FETCHes an unevaluated closure from a heavily-loaded PE for execution on the lightly-loaded PE.

In general load management is provided by idle PEs seeking work. When a PE has no runnable tasks it issues a FISH message, requesting a task. The FISH is initially directed to the PE where that its most recent task came from, in the belief that a PE that once generated a task is likely to generate more[1].

When a PE receives a FISH message then if it has a spark, or potential task, it issues a SCHEDULE message to the FISH originator. SCHEDULE is very similar to RESUME, in that it contains the unevaluated closure which was sparked plus some nearby graph. If the receiving PE has no sparks it reissues the FISH to another PE (this may be done at random, so that all PEs are eventually visited, or perhaps to the PE's own favourite food source).

There is a limit to the number of PEs that a FISH visits, which is fixed depending on the size and topology of the machine. The last PE which is visited returns an unsuccessful FISH to the originating PE. On receipt of its own, starved, FISH the sending PE may immediately reissue a FISH. Alternatively, if the PE has seen many FISH recently, indicating that the machine is lightly loaded, then it delays issuing the FISH in order to avoid swamping busy PEs with FISH messages.

A FISH could also record information about the loading of the PEs visited and this could be used by subsequent PEs. We do not intend to implement this initially, however.

5 Garbage Collection

A primary aim of the design is that each PE can garbage collect locally. Local garbage collection is substantially the same as the current GRIP garbage collection. The exception being that the indirection, LA $\to$ GA and GA $\to$ LA tables are all rebuilt. More specifically, a new entry is made whenever a global object is moved. As on GRIP, garbage collection may in desperation revert a copy of a global object into a fetch-me.

Every global object has a weighted reference-count in its indirection-table entry. When the weight reaches zero the indirection-table entry can be reused and the object will be collected by the local collector. A global collector is required to collect cycles of global objects that span more than one PE. In our current design this entails synchronising the entire machine.

[1]Our intuitive model here is one of a school of hungry fish circling a source of easy food, such as a boat.

6 Database Requirements

One major purpose of this work is to create a basis for our work on functional databases. There are several control aspects we wish to explore. These include introducing annotations for

- Keeping a task local to a PE if possible. This should help avoid proliferating tasks if they are known to be small.
- Sending a task to the PE where the data resides. This is important for the database application where a PE may hold large quantities of data. In the current implementation, like most other parallel functional implementations, any PE can evaluate a task. For example a PE may evaluate a task that requires data held entirely at a second PE. The two PEs will engage in an expensive sequence of FETCH/RESUME messages.
- Locking data to a PE if possible.
- Exporting compressed data if the cost of re-converting it is less than the communication cost.

Obviously, excessively fine a level of control can lead to poor resource usage, and we wish to avoid this.

7 Related and Future Work

The design of GRAPH for UMM is a development of the GRAPH for PVM described in the companion paper [4]. Indeed [4] contains a summary of related work.

The work is at an early stage. The design is nearly complete and implementation will begin in the very near future. We hope that GRAPH for UMM will provide a robust and portable parallel implementation of Haskell. The immediate target machines are an ICL Goldrush machine and the Fujitsu AP1000. The immediate application is a transaction-processor and we may extend the design to include annotations to support the transaction application.

References

[1] Akerholt G. Hammond K. Peyton Jones S.L. Trinder P.W. Processing Transactions on GRIP: A Parallel Graph Reducer. Proceedings of PARLE '93, Springer-Verlag LNCS 694 (1993), 634-647.

[2] Bevan D.I. Distributed Garbage Collection Using Reference Counting. Proceedings of PARLE'87, Springer-Verlag LNCS 259 (1987), 176–187

[3] Burton F.W. Rayward-Smith V.J. Worst Case Scheduling for Parallel Functional Programming. To appear in Journal of Functional Programming (1994).

[4] Loidl H.W. Hammond K. GRAPH for PVM: Graph reduction for Distributed Hardware. Proceedings of IFL '94 (September 1994).

[5] Peyton Jones S.L. Clack C. Salkild J. Hardie M. GRIP – A High Performance Architecture for Parallel Graph Reduction. Proceedings of FPCA '87, Springer-Verlag LNCS 274 (1987), 98–112.

[6] Runciman C. Wakeling D. Functional Languages Applied to Realistic Exemplars: the FLARE Project. UCL Press (1994).