GRAPH for PVM: Graph Reduction for Distributed Hardware

Hans Wolfgang Loidl* Kevin Hammond †
Department of Computing Science
University of Glasgow, Glasgow G12 8QQ, U.K.
Tel.: +44 41 339 8855 x {8332, 5619}
Fax: +44 41 330 4913
E-mail: {hwloidl,kh}@dcs.glasgow.ac.uk

Functional Languages Implementation Workshop. Paper 27

Abstract

We describe a version of the GRAPH system (Graph Reduction for an Assortment of Parallel Hardware) designed to execute parallel functional programs on a range of distributed-memory MIMD machines running the PVM communications harness. GRAPH was developed from the runtime system for the novel GRIP multiprocessor. Although this system has proved highly successful, being a novel architecture it is hard to compare results with other architectures or implementations. The CPUs used in the GRIP design are also rather old.

The principal extensions from GRAPH for GRIP to GRAPH for PVM are intended to handle high latencies more efficiently, by exploiting asynchronous communication, multi-threaded scheduling, and by grouping packets into larger entities where possible. The main innovation is the development of new, sophisticated *packet flushing* and *packet fetching* algorithms whose purpose is to allow graphs to be exported to and fetched from global memory without inter-processor synchronisation. As communication and reduction can be interleaved, a new form of synchronising communication and reduction is necessary.

We also give a short outlook on the design of a new portable distributed graph reduction system, GRAPH for UMM, that adepts the ideas and techniques originally developed for the GRIP system to a distributed memory environment.

GRAPH for PVM has been implemented on a network of SUN workstations and it is currently being tested and debugged. Although no extensive performance measurements have been performed yet, the available data shows a significant decrease in the overall communication and thus an improvement in the overall performance of the system.

1 Introduction

Background

The functional programming model offers a high degree of inherent parallelism since sub-expressions can always be reduced in parallel [16]. However, in order to achieve high performance, a runtime system has to exploit this parallelism without imposing large overheads on the execution of the program. Non-strict semantics, as specified for languages such as Haskell[5] can also limit parallelism, through introducing demand constraints in a graph reduction implementation. These are some of the issues which parallel functional implementors must contend with.

At Glasgow, we have already developed a fast sequential Haskell compiler based on graph reduction[19]. This compiler has been adapted for parallel execution, and a number of interesting scheduling and other results obtained for our novel architecture GRIP[4]. We now plan to develop this parallel implementation for other, more widely-available architectures: the GRAPH system (Graph Reduction for an Assortment of Parallel Hardware) is intended to be a single generic runtime system covering a variety of MIMD hardware, ranging from true shared-memory machines such as the Sequent Symmetry, through virtual shared-memory machines such as GRIP[18] or the KSR-1[20], to closely-coupled distributed machines such as the ICL EDS/Goldrush[7].

*Supported by a Kurt-Gödel scholarship (GZ 558.012/38-IV/5a/93) of the Austrian Ministry of Science and Research, by a KIP scholarship of the government of Upper Austria and by a 'Förderungsstipendium' of the Johannes Kepler University Linz.
†Supported by a SOED Research Fellowship from the Royal Society of Edinburgh and the UK SERC Parade project.

Motivation

The purpose of the work described here is to move from our original low-latency, high-bandwidth machine (GRIP) to a more generic model, suitable for higher-latency, lower-bandwidth communication as found on conventional distributed memory machines. As a secondary goal, we plan to study the implications of targetting GRAPH to loosely-coupled systems based on networks of workstations. In order to maximise portability, our work exploits (but is not limited to) the widely-used PVM (Parallel Virtual Machine) communications harness [15].

A prototype implementation of GRAPH for PVM has already been produced based directly on GRAPH for GRIP[3]. In a more distributed setting, this implementation suffers from a number of problems:

1. Much communication is synchronous.
2. Packets are generally small.
3. There are potentially too many control packets.

Synchronous communication is tolerable and even desirable in a low-latency setting, since it simplifies the runtime environment and avoids expensive context-switching overheads, but it is impractical for high-latency networks since much delay will be introduced. Small packets are also more suitable in a low-latency, high-bandwidth setting, where packet construction and communication times are broadly similar, but can be disastrous in a high-latency, low-bandwidth setting since network saturation and long delays can be introduced. Similarly, excessive control packets can lead to starvation and increased latency for a low-bandwidth system, but are essentially irrelevant to GRIP.

The work described here attempts to address the first of these problems by moving to an *asynchronous communication model*, and the second by *combining small packets* into larger ones wherever practical. We do not yet have sufficient experience with GRIP-style load management to know whether the level of control packets needs to be changed, with a corresponding introduction of new control mechanisms, but we suspect that this may be the case.

Extensions to Previous Work

The adoption of asynchronous communication means that GRAPH for PVM must be re-designed to allow context switches to occur between messages. This change introduces several major complications (described in Section 3), not the least of which is the need to synchronise PEs on locally created *locked* nodes. It is also possible that certain implicit scheduling assumptions must be overturned, and that previously obtained performance results will no longer apply. For example, GRAPH for GRIP attempts to conserve memory by exectuting threads *unfairly*: preference is always given to the previously executing thread if it can be run, or to threads which appear earliest in the thread queue otherwise. The new asynchronous communication scheme will cause more threads to be started and executed concurrently on one PE, leading to a fairer schedule but with increased memory consumption, and perhaps longer execution time.

In this paper we show how GRAPH for PVM can be extended to allow both *asynchronous communication* and *multi-threaded scheduling*. The principal complication other than context-switching comes from the process of making local heap objects from one processor globally available, a process we call *flushing*. In order to preserve the important property of locally-independent garbage collection, if a graph node is made global, anything it (transitively) refers to must also be global. This process can be expensive. GRAPH for GRIP flushed a graph bottom-up in an incremental fashion. GRAPH for PVM attempts to use scarce bandwidth more efficiently by flushing a complete graph as a single packet whenever possible. This introduces a number of interesting flushing/reduction synchronisation problems, which we address here (Section 4.2). Similarly, the fetching of a global graph into local heap is done by constructing and sending a packet.

The structure of the paper is as follows: Section 2 describes the GRAPH architecture and compares the characteristics of communication in a GRIP-like and in a PVM-based system. Some basic principles about the underlying memory manager are also discussed. Section 3 discusses the new multi-threaded scheduling strategy. Section 4 describes the packet flushing and fetching algorithms. Section 5 gives some performance measurements of the single-threaded prototype and the new multi-threaded system. Section 6 refers to related work in this area. Section 7 summarizes the essential parts of this work. Section 8 gives an outlook on further work.

2 Characteristics of GRAPH

2.1 The GRAPH Architecture

The original GRAPH for PVM followed the GRAPH for GRIP design in being single-threaded: it executes one thread of computation, which reduces some part of the graph, until this thread is finished. Except for special cases like garbage collection there is no possibility of suspending the execution of the current thread and resuming it later. Consequently, a process gets blocked whenever a thread sends a message and immediately needs a reply to it.

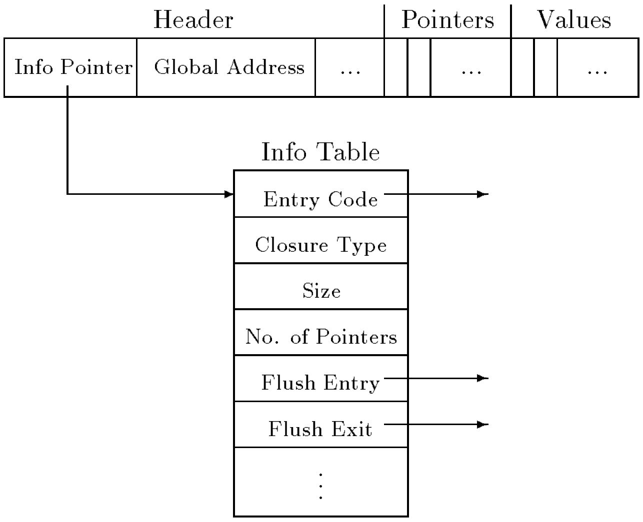

Figure 1: Structure of a Closure

This approach exactly models the behaviour on the GRIP machine [18], for which the runtime system was originally written. GRIP is a novel "shared-memory" MIMD multi-processor designed for parallel graph reduction. It contains two main kinds of components connected by a fast 2-level bus: Processing Elements (PEs) perform graph reduction, and microprogrammable Intelligent Memory Units (IMUs) hold the shared graph, and manage global task pools.

The communication can be characterized as low-latency and high-bandwidth. On GRIP packet-switched communication is used with a 100 MB/s Futurebus and a 20 MB/s local bus on the two levels of communication. The latency for transmitting a one-word packet and getting a reply is about 10 μs. In contrast, the communication in a typical PVM based system shows high-latency and low-bandwidth. Tests with PVM have shown that typically around 10 ms are needed for sending a packet. This time increases only slowly with increased packet size.

GRAPH for PVM uses essentially the same structure as GRAPH for GRIP. PEs and IMUs are realised as separate PVM processes, which communicate by message-passing. The most important messages between PEs and IMUs are:

- Flush for flushing a whole subgraph to an IMU. This operation has to be done for example when an explicit spark is encountered or when an update should be performed.
- Enter for entering a closure. For this operation the closure has to be fetched by the thread and locked by the IMU. If the closure is locked already, the thread gets blocked.
- Update for updating a closure with the result of a reduction. By this operation threads that are locked can be resumed.
- Resume for awakening a thread that has been blocked on a locked node.
- Get Thread for requesting work from an IMU.
- New Thread for sending a sparked graph to be evaluated to a PE.

For more details on the general architecture of GRAPH see [3].

2.2 Memory Layout

In this section we briefly describe the layout of objects in the heap of the STG-machine. This layout shows a tagless representation of closures; i.e. no tags are used to distinguish between the different kinds of closures. Instead this information is implicit in the closure. The main advantage of this concept is that no tags have to be checked to perform actions such as reducing the graph or garbage collection; these operations are encoded into standard places in a type-specific "info table".

A functional program is represented as a graph in the STG-machine. The nodes in this graph ("closures") all have a common structure: the first part is a header with a pointer to the *info table* and a *global address* field. This field contains the unique global address (GA) of that closure on one of the IMUs. A local closure is indicated by a GA of 0. Two bits in the GA are used to mark global constants (called permanent local closures) and closures that form an indivisible unit with their parent closures. The next part of the closure contains pointers to the node's children, and the final part of the closure contains basic values. This structure is shown in Figure 1.

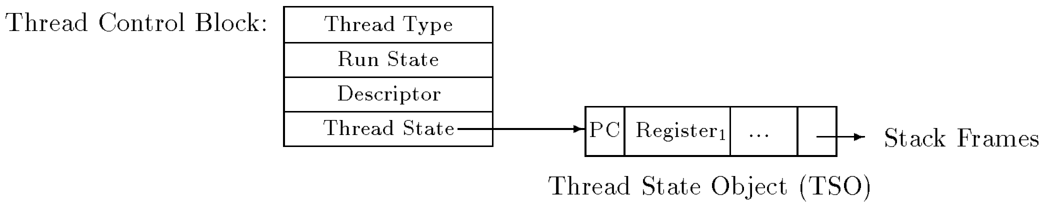

Figure 2: Thread State Object (TSO)

The info table contains information specific to the type of closure. In particular, it contains a pointer to the entry code at the beginning, which is executed when this node is evaluated as part of the graph reduction process. The type of closure, its size and the number of pointers it contains are also stored in the info table together with pointers to various specialised routines. Two of these routines are especially interesting, since they are used by the flushing algorithm:

- The *flush entry* routine is executed in the flushing algorithm when entering a closure. It manages the graph-traversal using a pointer-reversal algorithm.
- The *flush exit* routine is executed when leaving a closure. This routine performs the actual flushing by sending a message, containing the current closure to the IMU.

Finally the info table contains pointers to various other routines that are needed during garbage collection, profiling, etc. For a more detailed discussion of the heap management in the STG-machine see [17].

3 Multi-threaded Scheduling

In this section we discuss the scheduler used in GRAPH for PVM. In contrast to the scheduler in GRAPH for GRIP it is multi-threaded, thereby allowing asynchronous communication between processes. This extension requires changes in the structure used for representing threads. Generally, after every communication the current thread has to be suspended. When receiving a reply to this communication it will be enabled again so that it can be resumed by the scheduler.

The basic structure of the scheduler is the following: It first checks if special actions like garbage collection, sparking graphs or updating global closures should be performed. If this is not the case a reduction process is started, where implicit priorities are given for the following kinds of tasks:

1. The previously executing thread.
2. A locally runnable thread.
3. A new thread created from a local spark.
4. A runnable thread that has been exported by another PE.
5. A new thread created from a global spark.

Note that this realises an unfair scheduler, as the previously executing thread has the highest priority.

In the multi-threaded scheduler a rescheduling operation will be performed whenever the reduction process performs some communication for which the reply is immediately needed. In such a case the current thread is suspended, requiring a context switch.

For performing a context switch the current state of the thread has to be saved. All registers and the program counter of the current thread are copied into a special data structure, the thread state object (TSO; see Figure 2). Each thread has one such TSO attached to it. As this step is quite similar to the actions that have to be performed when garbage collection is done, some existing code can be reused here.

The TSO itself is an ordinary closure with the same layout as all closures. This representation has the advantage that the flushing mechanism can be used in the case where a thread, together with its TSO, should migrate to another PE.

The overall state of the program execution is represented by a list of *thread control blocks*, each of which represents one parallel thread. Figure 2 shows the structure of one such thread control block, consisting of the following fields:

- The *thread type*, which may be Main for the main thread, Normal for normal reduction threads or Fail for threads producing error results.
- The *run state*. The most important run states in the GRAPH for GRIP scheduler are:
  - Runnable for threads that may be executed by entering the node they are pointing to.
  - Global Blocked for threads blocked on a global closure.
  - Local Blocked for threads blocked on a local closure.

  - Reserved for new threads that are requested by the PE from an IMU.

  Additionally to these run states, the following are needed for a multi-threaded scheduler:

  - Flushing for threads flushing a local graph to an IMU.
  - Blocked on Flush for threads that are trying to evaluate a closure that is just being flushed.
  - Fetching for threads that are fetching a global graph from an IMU.

- The *descriptor* field, which has different meanings for the different run states (e.g. in the case of Blocked on Flush it points to the node that is being flushed).
- The *thread state* points to the TSO of this thread.

In summary, the following actions have to be performed when suspending or resuming a thread:

- **Suspending:**
  1. Save appropriate registers and program counter to the TSO of the current thread.
  2. Change the run state of the current thread.
  3. Reschedule i.e. jump into the top level scheduling loop.
- **Enabling:**
  1. Change the run state to Runnable.
- **Resuming:**
  1. Restore registers and program counter from the TSO.
  2. Enter the current closure by jumping to the program counter.

4 Packet Flushing/Fetching

The flushing algorithm is based on a pointer-reversing mark-scan garbage collection algorithm [21]. This algorithm avoids the usage of a stack for traversing the graph by reversing the pointers in a closure. For that the flush entry and flush exit codes in the info table are used.

In the old flushing algorithm one message is sent to the IMU for each closure in the graph. This communication is synchronous, because the response (a global address) is needed for flushing the next closure.

The straightforward idea of keeping the structure of the pointer reversing flushing algorithm and rescheduling after every send operation would lead to a huge number of context switches with a large overhead for saving and restoring registers in the TSO. Therefore, the flushing algorithm is changed in such a way that the whole graph is first copied into a flush packet[1] that is then sent to the IMU. Normally, this approach reduces the number of messages from the number of nodes in the graph to one. This reduction of the message transfer for the important and expensive routine of flushing a graph already increases the performance of the whole system significantly.

In the packet flushing algorithm two more messages have been introduced that are actually combinations of common message pairs in the system:

- The Flush & Spark message performs a packet flushing and also notifies the IMU that the flushed graph is a sparked one.
- The Flush & Update message performs a packet flushing and also performs an update of a global closure with the flushed graph.

By introducing these new messages, the sending of separate Spark and Update messages can be avoided.

4.1 The Packet Flushing Algorithm

In the packet flushing algorithm the basic structure of the pointer reversal routine is retained. However, instead of sending a message at the end of every flush exit routine, the closure is just put into a *flush packet* at that point. The flush packet is sent at the end of the traversal of the graph, allowing the current thread to continue execution.

The whole flushing process can be divided into the following steps:

1. *Traverse the graph* by calling the flush entry and flush exit routines stored in the info tables of the closures. In each flush exit routine add the contents of the closure to the flush packet.
2. *Send the flush packet* to an IMU. Usually the target IMU is changed after having allocated a certain heap size. In contrast to the old flushing algorithm the new one guarantees to allocate all parts of the flushed graph on the same IMU.

[1] For the sake of simplicity we assume that the graph fits into one fixed size packet. If not, only the packet offsets have to be changed to include the number of the packet also to make them unique.

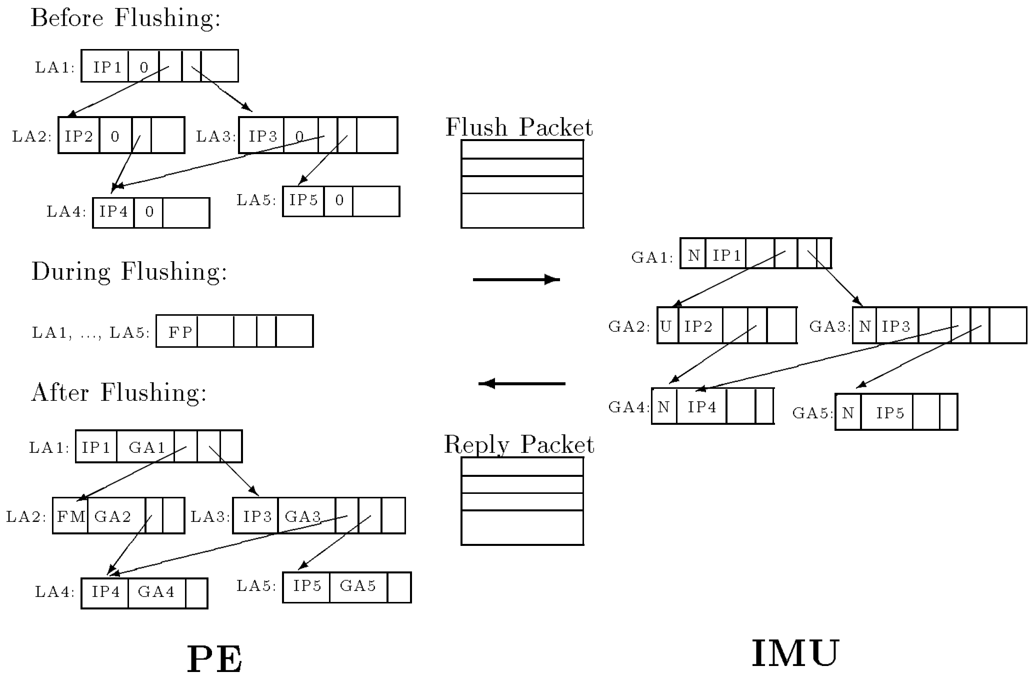

Figure 3: Packet Flushing

3. *Unpack the flush packet* as soon as it arrives at the IMU and *construct a reply packet*.
4. *Send the reply packet* to the PE that sent the flush packet as soon as the reply packet is constructed.
5. *Unpack the reply packet* as soon as it arrives at the PE and put the information from the reply packet into the graph. This requires another traversal of the graph.

The structure of the packet flushing algorithm is sketched in Figure 3. Below we discuss some important details of these steps.

The Structure of the Flush Packet: Figure 4 shows the overall structure of the flush packet. The first entry in the packet is the local address of the root of the graph that is being flushed. This information is not needed by the IMU itself, but it is needed in the reply packet for the PE to associate the reply with the right local closure. Note that normally the local address of the closure might change due to garbage collection between sending the flush packet and receiving the reply packet. We circumvent this problem by adding the roots of graphs that are being flushed to the list of closures that are recognized by the garbage collector as the roots of live graphs and indirecting through this graph table when we receive a reply.

The NU-mask at the end of the flush packet associates with each closure in the packet a flag indicating whether this closure is updatable (U) or in normal form (N). This NU-mask conveys important information to the IMU. For example, normal-form closures don't need to be locked when they are evaluated and can be ignored if they are sparked.

The two pointer fields at the end of the packet are used when a FLUSH & SPARK or a FLUSH & UPDATE message is sent instead of a plain FLUSH message. In these cases the pointer fields contain a pointer to the sparked closure or pointers to the source and the target of an update, respectively.

Cycles in the graph are broken in the same way as in the old flushing algorithm: A hole is allocated for the node that closes the cycle in its flush entry routine. After having copied the actual closure to the flush packet in its flush exit code, this hole is filled with an indirection to this closure. The only difference to the old flushing algorithm is the fact that now this allocation and filling of holes takes place in the flush packet rather than in the IMU itself.

Unpacking the flush packet (IMU): When receiving a flush packet the IMU allocates memory for each of the closures in the global heap. This allocation is implicitly compacting as it basically keeps the block structure of the flush packet. The global address of the closure and the info pointer of the closure, which is a part of the closure itself, are put into a reply packet that is constructed while unpacking the flush packet[2]. The structure of the reply packet is also shown in Figure 4. For a discussion of why the info pointer

[2] Actually, an info pointer is only necessary for non-updatable closures. For updatable closures the info pointer has to be replaced by a FetchMe info pointer anyway.

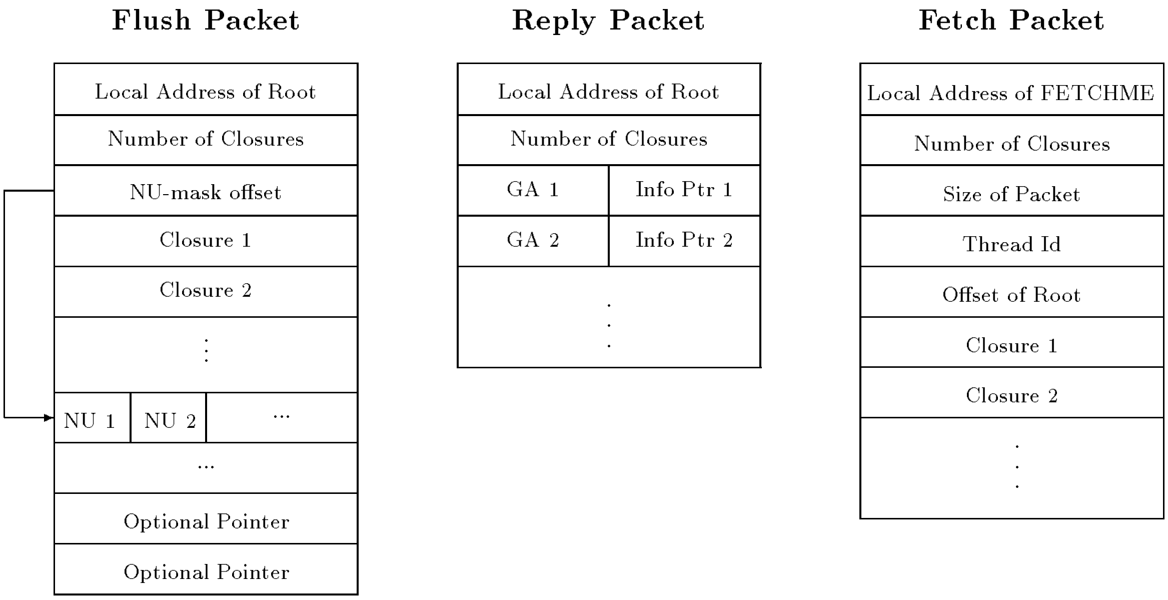

Figure 4: Structure of Flush, Flush-Reply and Fetch Packets

of the original closure is needed in the reply packet see Section 4.2. Note that only the local address of the root of the flushed graph is needed in the reply packet, because the graph is traversed in the same way when filling in the global addresses as in the flushing algorithm.

Filling in the global addresses (PE): The following steps are taken as soon as a PE reads one of the special reply packets.

1. *Unpack* the reply packet header to get the local address of the root of the graph that has been flushed.
2. *Traverse* the graph in the same order as in the flushing algorithm. At each closure do the following:
   (a) Insert the new global address taken from the reply packet.
   (b) Insert the old info pointer taken from the reply packet.
   (c) Unblock all threads that have been blocked due to the closure being flushed (see Section 4.2 for an explanation of this step).
   (d) If the closure is updatable then it has to be replaced by a FetchMe closure pointing to the global address of the closure.

Note that in this algorithm every updatable closure is transformed into a *FetchMe* closure. This transformation is necessary as parts of the flushed graph might be shared with another local graph. Therefore, it does not suffice to just change the root of the flushed graph into a *FetchMe* closure.

A further complication arises with the new packet flushing algorithm in connection with a global garbage collection. If a global garbage collection occurs as the result of trying to flush a closure, then the closure must be reinstated locally before the global garbage collection can proceed. This restoration prevents roots to global nodes from being accidentally lost.

4.2 Simultaneous Flushing and Reduction

One important new problem that is introduced with the new flushing algorithm is the possibility of simultaneous flushing and reduction of a subgraph. This situation could not arise in the old flushing algorithm, as the thread that flushes a graph could not be suspended. However, in the packet flushing algorithm a PE just initiates the flushing process by sending the flush packet and then continues execution. Therefore, another thread might try to reduce a subgraph of the graph that is being flushed. If it does so, and it updates the subgraph with the result of the reduction, the structure will have changed when the reply packet arrives and so the global addresses can't be filled in correctly. Such an inconsistency between the local graph and the global graph must be avoided.

The solution to this problem is to 'mark' each node in the graph that is being flushed. Due to the principle of 'taglessness', no special flag is used for marking. Instead the info pointer of the closure is changed to point to a new info table[3]. We call this info table 'FlushPending' (FP in Figure 3).

The entry code for this new FlushPending info table performs the following steps:

[3]Actually, a whole set of new info tables is needed — one for each old info table – because the new tables must contain the same

1. *Block the current thread*. Analogous to the new info table a new runstate (BLOCKED ON FLUSH) for a thread is introduced. A thread which is in this runstate also keeps a pointer to the closure upon which it is blocked.
2. *Add this thread to the 'blocking table'*. The 'blocking table' associates with each 'FlushPending' closure those threads that are blocked on this closure. When filling in the global addresses for the flushed closures, the threads in the corresponding entry of the blocking table are released (see Section 4.1).
3. *Save the thread state*. The registers and the program counter are saved in the TSO of the current thread.
4. *Reschedule*. Jump into the top level scheduling loop. The current thread remains blocked until a reply packet for the graph being flushed arrives and the global addresses are filled in.

In the case of a garbage collection all closures in the blocking table are made live roots. Thereby, it is guaranteed that they are not moved and that the information in the blocking table remains consistent through garbage collections. The same holds for the root of the graph that is flushed.

The info pointer of such a closure is changed back to its old value when the graph is traversed again in order to fill in the global addresses and info pointers that are part of the reply packet.

4.3 The Packet Fetching Algorithm

When a 'FetchMe' closure is entered during graph reduction, the closure to which it refers must be fetched from an IMU. In GRAPH for GRIP this has been done lazily; i.e. only this closure is fetched[4]. In the local copy the children of the closure are replaced by FetchMes for the corresponding global closures.

In the packet fetching algorithm the whole subgraph of the current closure is fetched. This strategy is obviously very eager. In the design of GRAPH for UMM we are considering alternatives like fetching a graph of fixed size or a graph with a fixed number of updatable closures. We assume that extensive tests with these different fetching strategies will be necessary to evaluate their efficiency in the GRAPH system.

The basic structure of the packet fetching algorithm is the same as for the packet flushing algorithm. Particularly, the same routines for packing nodes into the fetch packet, for breaking cycles etc. can be used. The fetching of a graph is started when an ENTER message for an updatable or a normal-form closure arrives at an IMU. In this case, a fetch packet is constructed (Figure 4 shows the structure of such a packet) by traversing the graph in the IMU. The local address of the FetchMe closure and an identifier for the thread that initiated the fetch are sent together with the ENTER message and put into the header of the fetch packet.

The principal *differences between the fetching algorithm and the flushing algorithm* are the following:

- The graph traversal is done by a recursive function. No pointer reversal algorithm is used in packing the fetch packet.
- When a locked closure is found during the graph traversal, it is not copied into the fetch packet. Instead, a FetchMe closure for this locked closure is put into the packet. So, it might be the case that when this global closure is actually entered it will aready have been unlocked as the other processor that was evaluating it has done an update in the meantime.
- Similarly, a FetchMe closure is put into the fetch packet for a closure that actually resides on another IMU. This prevents the sending of further messages from one IMU to another.
- In contrast to the packet flushing algorithm, the packet fetching algorithm is not implicitly compacting. Such compaction only seems to be advantageous after having performed reductions on the graph in the PE which might have changed the overall structure of the graph.

5 Performance Measurements

This section presents a comparison of the performance of the prototype GRAPH for PVM system with single-threaded scheduling [3] and the new system with asynchronous communication and multi-threaded scheduling. As the new software is still in the testing and debugging phase, only a small set of performance measurements can be given. Particularly, the test programs we have running are too short to allow a detailed evaluation of the presented extensions in GRAPH for PVM.

As a simple test example the well-known nfib program has been used in two slightly different variants, shown in Figure 5: In the basic variant one subexpression, `nf2`, is evaluated in parallel for all calls to `nfib1`, except for the base case. In the `nfib2` variant a sequential `seqfib` function is called as soon as the input becomes smaller than a certain bound. This variant limits the amount of parallelism and increases the

function pointers for garbage collection as the old tables. However, for the sake of simplicity we deal in this paper only with one new info table.

[4] The only exception from this rule are special children that are atomically joined joined to their parent closures. Such closures are fetched immediately.

```
nfib1 n = if n < 2 then 1 else nf2 `par` nf1 `seq` nf2 + nf1 + 1
                                 where nf1 = nfib1 (n-1)
                                       nf2 = nfib1 (n-2)

nfib2 n = if n < bound then seqfib n
                       else nf2 `par` nf1 `seq` nf2 + nf1 + 1
                            where nf1 = nfib2 (n-1)
                                  nf2 = nfib2 (n-2)
```

Figure 5: Example Programs: Two variants of nfib

Synchronous Communication		
Message	nfib1	nfib2
Flush	180	40
Enter	90	22
Spark	88	20
Update	88	20
Get Thread	2	2
New Thread	1	1
Reply	356	84
Total	805	189

Asynchronous Communication		
Message	nfib1	nfib2
Flush & Spark	88	20
Flush & Update	4	4
Flush Reply	92	24
Update	84	16
Enter	92	25
Fetch	91	21
Get Thread	4	4
New Thread	2	2
Total	457	116

Figure 6: Messages sent using synchronous and asynchronous communication

average granularity of the parallel threads. In the experiments discussed below we have chosen `n`= 10 and `bound`= 5. In all cases 2 PEs and 2 IMUs have been used in total on 2 workstations.

The absolute runtimes of `nfib2` for these input values range from 0.13 to 0.15 seconds, using asynchronous communication, and from 0.18 to 0.20 seconds using synchronous communication. The `nfib1` variant is in average about 3 to 3.5 times slower. However, these results of first experiments have to be taken with extreme care, as the runtimes are very short and their deviations for the same input are quite high due to the PVM based communication.

Therefore, we mainly base our measurements on the number of messages that are sent in both variants. Because of the high latency in a PVM based system the number of messages is the most important factor influencing the overall performance of the system.

Figure 6 shows the number of messages that are sent during the execution of `nfib1` and `nfib2`, using synchronous and asynchromous communication. In the synchronous communication protocol every message to the IMU, except for an Enter message, is answered with a Reply message. In contrast, in the asynchronous communication protocol, no reply messages are needed except for the case of flushing[5]. Therefore, eliminating Reply messages that are not needed in the multi-threaded version any more reduces the total number of messages by about 30 % in these experiments.

It should also be noted that almost all of the flushes are done in combination with sparking a new closure, thereby justifying the introduction of a new message for that purpose.

The ratio between Flush messages and Flush & Spark messages shows, that for this small test program we can expect an average flush packet size of only 2 closures. In general this ratio is crucial for the improvement in performance that can be achieved with the packet flushing/fetching algorithms.

Almost all flushes are done for sparking a graph. By using the `par` construct the programmer can explicitly decide where to spark a graph and therefore indirectly determine the size of the Flush & Spark packets. A rather coarse granularity is therefore not only favourable for efficient computation but also for efficient communication.

The `nfib2` variant drastically reduces the number of Flush messages that are sent, because of the larger granularity. The size of sparked graphs remains unchanged compared to `nfib1`. As the number of the other messages is the same for synchronous and for asynchronous communication, the ratio of the total number of messages between the two variants is lower in this case.

[5]Note that in the synchronous communication protocol Fetch messages are of the same type as Reply messages and are therefore not listed explicitly.

6 Related Work

The new packet flushing/fetching algorithms bear some resemblance to the strategy used in the parallel ABC machine [14], used in the Concurrent Clean project, for copying graphs between processors. In [9] a detailed discussion of the costs for copying graphs in a distributed memory environment is given. This analysis includes the costs for creating a local copy of the graph in memory efficient representation, sending the graph and unpacking the graph again. It is shown that these costs form a bottleneck for a large class of parallel programs. By choosing arrays as underlying data structures in the copying process the overall costs could be decreased significantly.

Similar problems to ours related to the whole process of sending a graph (memory efficient representation, shared nodes, cycles etc.) are described in [8]. Contrary to our approach, an explicit tag is used in the parallel ABC machine to synchronise simultaneous evaluation and flushing. Furthermore, the scheduling in this machine is fair.

The ZAPP system [2] is a work distribution scheme, designed to exploit especially divide-and-conquer parallelism. Concurrent Clean is used as programming language for this system. The graph copying process used there is mainly based on the structure of a two-space garbage collector with two traversals of the graph [13], whereas in our algorithm only one graph traversal is needed.

In the implementation of the Parallel Abstract Machine PAM [11], the asynchronous communication scheme is realised on a Transputer network by using three independent processes that handle the communication. This mechanism is necessary as the underlying language only provides synchronous communication. Like our runtime system PAM uses a passive process distribution strategy where idle processors explicitly ask for new work.

The HDG-machine [10], another abstract machine for implementing parallel graph reduction in a distributed memory environment, is implemented on a network of Transputers. One characteristic of the scheduling strategy in this machine is the existence of three different task pools, where active, migratable and blocked tasks are stored, respectively.

The parallel $\nu - STG$ machine [6] is another example of extending the set of info tables and the number of entries in the info tables for realising parallel graph reduction. Contrary to our approach, the stack for a thread is distributed over the processors in the $\nu - STG$ machine. Similarly to the HDG machine, two kinds of spark mechanisms are available: one for mandatory and one for optional sparks. In the GRAPH system, however, we only have optional sparks as the scheduler may decide, based on the processor load, not to turn a spark into a thread.

In contrast to our approach, the $< \nu, G >$ machine decribed in [1] is based on a shared memory model of computation. Therefore, sparking only requires a pointer to be placed into a global spark-pool whence it can be accessed by all processors. No flushing and no explicit routines for packing and unpacking graphs are needed in this environment. The mechanism for suspending threads is discussed in detail there. The main differences of this mechanism to that in GRAPH are mainly due to the underlying shared-memory architecture.

7 Summary

The GRAPH system is in the process of transformation from a specialised runtime environment for a single novel parallel architecture to a generic environment suitable for programming a wide variety of parallel architectures. This paper has described one aspect of that work: the expansion from GRIP to a PVM-based system through the provision of asynchronous communication, multi-threaded scheduling and larger packets. Collectively, these improvements allow us to deal more efficiently with the high latencies found in most distributed memory implementations.

The main problem in implementing these extensions is in connection with the flushing and fetching algorithms, which have been modified to flush and fetch graphs as a single packet whenever possible. The whole process of flushing and fetching graphs is generally quite expensive. Initial performance measurements, although quite crude, already show a significant decrease in the number of messages that are sent.

A new problem introduced by this system is the possibility of simultaneous flushing and reduction of a subgraph, which requires synchronisation within a PE. We have solved this problem by introducing new "info table" entries for flushing, in the spirit of other "tagless" implementations.

As the implementation of this design is still in its testing and debugging phase, the performance measurements presented are quite crude. Clearly, the number of messages has been reduced with the packet flushing/fetching algorithms. However, further experiments with bigger test programs will be necessary to evaluate details in the design of GRAPH for PVM. Furthermore, we plan to merge the extensions presented in this paper with optional features of the runtime system that previously have also proven to improve the performance of the system significantly: Local sparking and deferred updates. Local sparking keeps a spark at the local PE instead of immediately flushing it to an IMU. Thereby unnecessary flushes might be avoided. Deferred updates defer the sending of an update message to an IMU until the value is needed by another processor.

8 Further Work

Our overall plan is to extend this system to GRAPH for UMM, covering a variety of other less GRIP-like machines. For a description of the new memory and load management system in GRAPH for UMM see the accompanying paper [22]. An immediate target is a PVM implementation for loosely-coupled machines, which this work is a first step towards. In the slightly longer term, we expect to target shared-memory systems such as multi-processor Suns running Solaris 2.

It is now commonplace to think of inter-process communication in terms of message-passing, even if it is actually implemented by (physically or virtually) shared memory. We will therefore not defend this model here.

We believe the separation of the heap into global and local components to be an important design decision, since it permits independent garbage collection without synchronisation most of the time. This benefit carries through even to shared-memory implementations, where it has been found that synchronised garbage collection can have a profound effect on overall performance and can prove to be a scalability bottleneck [12]. Separation into global and local heap also reduces bus contention and cache coherence problems when a majority of local nodes are being reduced.

We are less attached to the *physical* separation of global and local heap (administered by PEs and IMUs, respectively). By maintaining tables of incoming and outgoing global pointers for each PE, we can retain the important property of independent local garbage collection while eliminating the need for separate IMU and PE tasks. This design also allows the elimination of explicit global addresses on each node, since this information can be held in the local table of outgoing global pointers. The advantages of this are two-fold: a reduction in memory use, through not needing to store global addresses, and a reduction in execution time, through not needing to load/store and examine these addresses during normal execution.

Because of all this, a two level-heap will also be used in the GRAPH for UMM system, although the separation into PEs and IMUs will be dropped there.

Finally, although our implementation is described in terms of PVM, for portability, we expect to rewrite parts of the GRAPH message-passing system to exploit native communication modes for some important target architectures, for maximum efficiency.

References

[1] L. Augustsson and T. Johnsson. Parallel Graph Reduction with the $\langle v, G \rangle$-machine. In *FPCA '89, The Fourth Int. Conference on Functional Programming Languages and Computer Architecture*, pages 202–213, Imperial College, London, September 11–13, 1989. ACM Press, New York.

[2] FW. Burton and MR. Sleep. Executing Functional Programs on a Virtual Tree of Processors. In *FPCA '82, Int. Conference on Functional Programming Languages and Computer Architecture*, pages 187 – 194, Portsmouth, New Hampshire, October 1982.

[3] K. Hammond. Getting a GRIP. In *Fifth Int. Workshop on the Parallel Implementation of Functional Languages*, Nijmegen, The Netherlands, September 1993.

[4] K. Hammond and SL. Peyton Jones. Profiling scheduling strategies on the GRIP parallel reducer. Technical report, Department of Computing Science, University of Glasgow, August 1991.

[5] P. Hudak, SL. Peyton Jones, P. Wadler, and *et al.* Report on the Programming Language Haskell — A Non-strict, Purely Functional Language. Technical Report 1.2, March 1992.

[6] S. Hwang and D. Rushall. The $\nu - STG$ Machine: a Parallelized Spineless Tagless Graph Reduction Machine in a Distributed Memory Architecture. In H. Kuchen and R. Loogen, editors, *Fourth Int. Workshop on the Parallel Implementation of Functional Languages*, pages 411 – 437, RWTH Aachen, Germany, September 1992.

[7] ICL. The EDS Machine – System Introduction and Design Overview. Internal Document, December 1990.

[8] M. Kesseler. Communication Issues Regarding Parallel Functional Graph Rewriting. In H. Kuchen and R. Loogen, editors, *Fourth Int. Workshop on the Parallel Implementation of Functional Languages*, pages 139 – 157, RWTH Aachen, Germany, September 1992.

[9] M. Kesseler. Reducing Graph Copying Costs. In *PASCO '94, First Int. Symposium on Parallel Symbolic Computation*, Linz, Austria, September 26 – 28, 1994. World Scientific Publishing.

[10] H. Kingdon, DR. Lester, and G. Burn. The HDG-machine: a highly distributed graph-reducer for a transputer network. *The Computer Journal*, 34(4):290 – 301, 1991.

[11] R. Loogen, H. Kuchen, and K. Indermark. Distributed Implementation of Programmed Graph Reduction. In *PARLE 89, Proceedings of the Conference on Parallel Architectures and Languages Europe*, volume 365 of *Lecture Notes in Computer Science*. Springer, Berlin, 1989.

[12] J. Mattson. *An Effictive Speculative Evaluation Technique for Parallel Graph Reduction.* PhD thesis, University of California at San Diego, February 1993.

[13] DL. McBurney and MR. Sleep. Concurrent Clean on ZAPP. In MJ. Plasmeijer, editor, *Third Int. Workshop on the Parallel Implementation of Functional Languages*, pages 73–113, Nijmegen, The Netherlands, October 1990.

[14] E. Nöcker, M. Plasmeijr, and J. Smetsers. The Parallel ABC Machine. In *Third Int. Workshop on the Parallel Implementation of Functional Languages*, University Southampton, U.K., June 1991.

[15] Oak Ridge National Laboratory, University of Tennessee. *Parallel Virtual Machine Reference Manual*, version 3.2 edition, August 1993.

[16] SL. Peyton Jones. Parallel Implementations of Functional Programming Languages. *The Computer Journal*, 32(2):175–186, April 1989.

[17] SL. Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127 – 202, April 1992.

[18] SL. Peyton Jones, C. Clack, J. Salkild, and M. Hardie. GRIP — a High-Performance Architecture for Parallel Graph Reduction. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 98–112, Portland, Oregon, September 14–16, 1987. Springer, Berlin.

[19] SL. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell Compiler: A Technical Overview. In *Joint Framework for Information Technology Technical Conference*, Keele, 1993.

[20] G. Ramanatha and J. Oren. Survey of commercial parallel machines. *Computer Architecture News*, 21(3):13–33, 1993.

[21] P. Sansom. Dual-Mode Garbage Collection. In *Third Int. Workshop on the Parallel Implementation of Functional Languages*, pages 238 – 310, University Southampton, U.K., September 1991.

[22] P. Trinder, K. Hammond, H-W. Loidl, J. Mattson, A. Partridge, and S. Peyton Jones. GRAPHing the Future. In *Sixth Int. Workshop on the Implementation of Functional Languages*, Norwich, U.K., September 1994.