Getting a GRIP (DRAFT)

Kevin Hammond

Department of Computing Science, Glasgow University

Abstract

This paper describes a portable implementation of the Graph Reduction in Parallel (GRIP) software, built on the Parallel Virtual Machine (PVM) process control system. Using this system, it is easy for *anyone* to "get a GRIP" without needing to invest in special-purpose hardware. An important contribution of this paper is its detailed description of GRIP's Intelligent Memory Unit (IMU) software.

1 Introduction

GRIP Architecture

GRIP is a "shared-memory" multiprocessor built from off-the-shelf components, first described by Peyton Jones et al [PCSH87]. The GRIP machine comprises a number of Processing Elements (PEs) and Intelligent Memory Units (IMUs) linked by a 2-level bus structure. The communications system is a hardware implementation of a packet-switched message system with guaranteed delivery and which preserves packet order between 2 components. The communications network is relatively high-bandwidth (20MB/s and 100MB/s) and has low latency (~10μs for a round-trip, including software packet construction). Latency is aided by direct ports between the processor and the message buffering hardware. See Figure 1.

The most innovative feature of GRIP is probably its use of IMUs (which are microcoded bit-sliced devices). The IMUs manage shared global memory (including distributed garbage collection) and also maintain the global task pools. The fact that the IMUs are microcoded has proved invaluable in allowing the GRIP machine to evolve as our ideas have changed over time. They have even been used for a parallel logic implementation.

GRIP has several disadvantages, most of which stem from the very fact that it is a novel architecture. This means that the GRIP software is not widely available (and compilers etc. must be adapted to GRIP), it has very limited PE and IMU memory (1MB per PE, 5MB per IMU), and it uses processors which are no longer state-of-the-art (16MHz Motorola MC68020s). Even so, the performance results which have been reported for GRIP [e.g. HP92, Mat93] are quite respectable compared with other parallel implementations, and the relative speedups which can be achieved are very encouraging for a dynamically scheduled system.

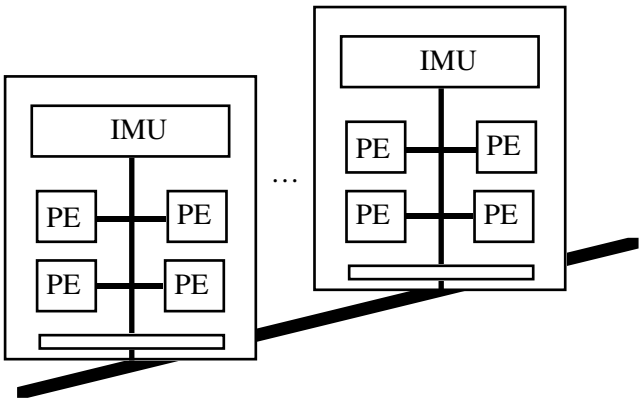


Figure 1: The Architecture of the Real GRIP

We have therefore decided to port the GRIP software to a more widely available environment, the Parallel Virtual Machine (PVM) communications harness [PVM93]. In doing this we hope to achieve two things: firstly to make our ideas more widely available and easily accessible, and secondly to produce a system that is acceptably efficient on a variety of architectures. It is not clear that this second objective is attainable, hence this project represents a serious piece of research as much as a simple software port.

The remainder of this section describes the PVM virtual machine architecture and provides an overview of the PVM port. Section 2 describes global memory management operations. Section 3 describes thread operations. Sections 4 and 5 discuss implementation and performance issues, respectively. Section 6 concludes.

The PVM Architecture

PVM provides a number of useful high-level facilities, which include simple process creation, abstract message-passing, message broadcast, and process groups. Most importantly PVM is designed to work for many different kinds of parallel machines, ranging from shared-memory multiprocessors (e.g. Sequent Symmetry), through closely-coupled distributed-memory machines (e.g. KSR1, Cray MPP), to loosely-coupled distributed networks of machines (e.g. Suns connected by Ethernet running TCP/IP). The system allows for dynamic configurations of heterogeneous machines. Messages are guaranteed delivery (assuming the destination process is still running), but may have high latency and low-bandwidth — the opposite case from GRIP.

For this implementation, we have decided to adopt a fairly general process model, but one that is broadly consistent with GRIP. There are a number of reduction tasks (PEs) and global memory tasks (IMUs). We also introduce (as on GRIP) a system manager task (SysMan) whose purpose is to control the startup/shutdown/failure sequences, synchronise global garbage collection and handle statistics collection. See Figure 2.

For consistency with the real GRIP, the two GRIP IMU functions of global memory management and global thread control are managed in a single IMU task. It would be possible to combine the IMU functions with a PE task, or to separate these two functions

into separate tasks if preferred for efficiency on a particular machine architecture. This would require the software to be restructured, and if PE and IMU functionality was combned would lose the property that reduction is never interrupted by communication. Interrupting reduction is undesirable because it forces an expensive context-switch to service each interrupt.

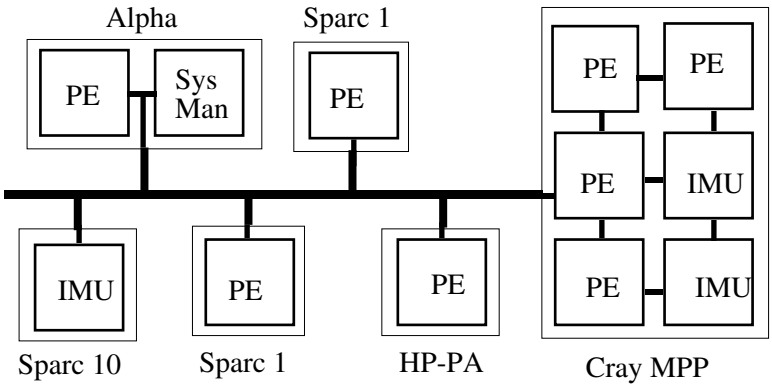


Figure 2: Architecture of a Virtual GRIP

2 Global Memory Operations

Tables 1 and 2 summarise the global memory operations which are provided for global memory management, including their parameters, all possible results, and which part of the runtime system uses the operations. How these operations are used is described in more detail in the remainder of this section. The *ga* fields are global addresses. The *tso* fields are globally addressed thread state objects, used to identify individual threads across the system. Other IMU operations manage global threads (described in section 3) or system initialisation/termination (the INIT, MAIN and FINISH operations, not described here).

Class	Opcode	Parameters	Result	Used in
Allocation	ALLOC	size ptrs IP arg_0 … argn	OK ga | FULL	Flush
	ALLOC_NF	size ptrs IP arg_0 … argn	OK ga | FULL	Flush
	ALLOC_HOLE	size ptrs IP arg_0 … argn	OK ga | FULL	Flush
	ALLOC_STACK	astack bstack …	OK tso | FULL	Stack Flush

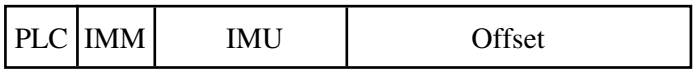
Table 1: Global Memory Operations — Allocation

Class	Opcode	Parameters	Result	Used in
Update	UPDATE	oldga newga	OK	Update
	ALLOC_UPDATE	oldga size ptrs $arg_0 \ldots arg_n$	OK | FULL	Flush
	ALLOC_NF_UPDATE	oldga size ptrs $arg_0 \ldots arg_n$	OK | FULL	Flush/ Update
	DEFER	ga	OK | FULL	Update
	UPDATE_REQUEST	ga	UPDATE …	IMU Enter
Enter	ENTER	ga	OK size … | BLOCKED	Enter
	UNLOCK	ga	OK	Enter
	READ_STACK	tso	OK astack …	Stack Fetch
GC	GC_POLL	<none>	OK finished	Global GC
	EVACUATE	ga	OK ga	Global GC

Table 2: Global Memory Operations — Update, Enter, GC

Global Addresses

Every object has as part of its header a global address. Global Addresses are essentially <IMU,offset> pairs, where the offset is not necessarily a physical address in the IMU, but serves to identify an object uniquely. One bit is used to indicate whether a global address is actually a global constant (a Permanent Local Closure, also known as a CAF). A second bit is used to indicate that the pointer is logically part of a larger object which should be fetched immediately. A global address of 0 indicates that the object is local to the PE.



Global Address Structure

At present PLCs are treated somewhat naïvely: they are not shared between PEs! This may be correct for simple PLCs such as integer constants, but is a potential disaster for large computations. Be very wary of a fully lazy lambda-lifter in such a system! It would be fairly straightforward to extend the update mechanism to allow cross-PE updates on large PEs, but this has not yet been implemented in the GRIP system.

Structure of Global Objects

While the PEs maintain many different kinds of object, in the IMUs these have been simplified to just 6: normal forms (NF), updatable objects (UPDATABLE), objects being evaluated (LOCKED), indirections (IND), deferred updates (DEFERRED) and stacks (STACK). Examples of each of these are shown in Figure 3.

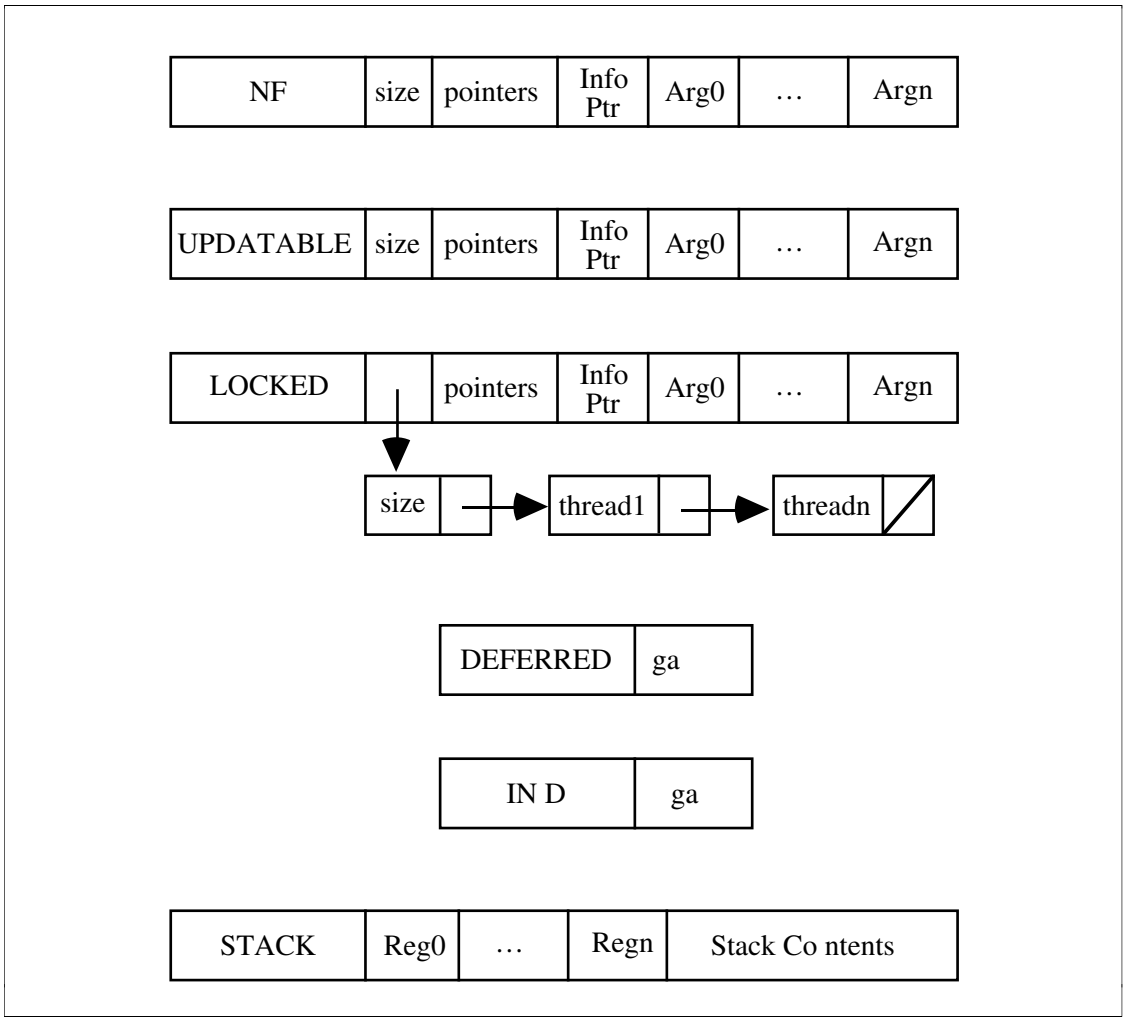


Figure 3: Types of Global Node

The LOCKED object structure is by far the most complicated, due to the need to maintain a queue of threads which are blocked on the object. This queue is created dynamically as PEs repeatedly enter a single UPDATABLE object.

To simplify garbage collection, each UPDATABLE, NF and LOCKED object contains a record of its size and the number of pointers it contains. A “black hole” (created by e.g. ALLOC_HOLE) is just an UPDATABLE object of size 0 with 0 pointers.

Evaluation

Evaluation proceeds as normal on the PE which has been nominated to evaluate the main closure until an explicit SPARK is encountered. Then,

i) The sparked closure is *flushed* to global memory;

ii) The spark is made globally available to an IMU using the SPARK operation;

iii) If there is an idle PE, the IMU notifies this PE using a NEW_THREAD operattion, otherwise the closure is added to the global spark pool;

iv) The idle PE creates a new thread state object for the sparked closure, and begins to execute it. When a global closure is evaluated the PE notifies the owning IMU with an ENTER operation. The IMU takes the appropriate action (which may involve blocking if the closure is LOCKED).

v) When a PE has finished evaluating a global closure, it notifies the owning IMU using a DEFER operation.

Flushing Nodes

The process of making a node globally available is known as flushing the node. In order to flush a node, its children must have global addresses. This is achieved using a algorithm which recursively flushes the node's children, then flushes the contents of the node to global memory. On GRIP and in the current PVM implementation this is achieved using a pointer-reversal algorithm based on Sansom's pointer-reversing Mark-Scan Garbage Collection [San91] (a similar algorithm is described in [Gro93]). This implementation has the advantage of requiring only a small fixed amount of memory beyond that available in the program graph, and is reasonably fast. Sansom's scheme uses the STG-machine *info. pointer* to record the position of the child being marked for nodes with a small number of children. This results in a large number of special garbage collection routines. To avoid extending the number of such support routines, the flush code uses the GA slot of the node (which is normally zero) to record the argument which is being flushed. This is also used in the check for cyclic structures (handled by a special mark bit in Sansom's scheme).

When flushing, we have the additional complication compared with Sansom's scheme that it may be necessary to abort the flush operation if there is insufficient memory to store the flushed object. This is achieved by "unwinding" the flush "stack", resetting the GAs of any nodes (but not their children) to 0.

Fetching Nodes

The process of reading a node from global memory to a PE's memory is known as *fetching* that node. Fetching normally proceeds incrementally, with the children of a

node being fetched only when they are entered. This is achieved using *FETCHME* nodes which contain a global address but no data. Nodes with immediate global addresses (e.g. array components) must be fetched immediately rather than incrementally using *FETCHMEs*. This is handled by the PE fetch code.

3 Global Thread Operations

Table 3 summarises the operations which are provided for maniplating global threads, Mattson describes the use of these operations in more detail, and gives some performance results on GRIP [Mat93b].

Class	Opcode	Parameters	Result	Used in
Work Location	GET_THREAD	<none>	<none>	Idle
	NEW_THREAD	SPARK ga | THREAD tso	<none>	IMU
Spark	SPARK	ga	OK	Spark
Export	EXPORT_RUNNABLE	ga	OK | FULL	Busy GC
	EXPORT_BLOCKED	tso	OK | FULL	GC
Poll	HUNGRY	Bool	RECEIVED	IMU
	THREAD_POOL	threads sparks	<none>	IMU

Table 3: Thread Management Operations

4 Implementation

Part of the code for the IMU implementation is given in Appendix I.

Messages

The message implementation is fairly straightforward. For example, to send a message to an IMU task:

```
IMUOp2(task,opcode,arg1,arg2)
GLOBAL_TASK_ID  task;
OPCODE  opcode;
int  arg1,  arg2;
{
      pvm_initsend();
       pvm_pkint(&arg1,1,1);
```

```
            pvm_pkint(&arg2,1,1);
            pvm_send(task,opcode);
        }
```

Status

The runtime system has now been fully rewritten for PVM with the exception of code to flush/fetch stack objects. All the IMU functionality except global garbage collection and stack objects has been implemented, but not yet fully tested. Threading code has been written for the SPARC and 68020. An example run on a network of Sun Sparcstations is shown in Appendix II.

5 Performance

It is too early to give accurate performance figures. However, inspecting the PVM source code shows that apparently simple functions such as adding an argument to a packet are in fact rather costly because of the need to translate data into the internal PVM form. For the Sun implementation, the inter-processor communication via TCP/IP adds a further layer of potential inefficiency. In fact, Nikhil reports that his group obtained a factor of 5 performance improvement on his machines by directly using the much more primitive UDP protocol instead of high-level PVM calls [Nik93]. In general, this would only be an acceptable implementation if the underlying hardware guarantees packet delivery, which is unlikely on a LAN, but it does suggest the magnitude of perforance which can be gained by a careful reimplementation of the PVM routines.

The broader issues of whether a GRIP-style implementation is practical on a LAN and what scheduling strategies must be adopted to achieve this remain open at the present time. We hope to report on our experiences in due course.

6 Conclusions

This paper has described a reimplementation of the GRIP software for non-GRIP-like machines (e.g. networks of workstations) using PVM as a message control sub-strate. The PVM implementation is more accessible both in the sense that it is now available on more hardware platforms, and in the sense that the software is no longer hardware-dependent (e.g. the GRIP microcode). Although performance results are not yet available, after tuning we hope that this system will prove to be practically useful. We may need to exploit techniques that have not yet been tried on GRIP (e.g. speculative evaluation [Par92,Mat93a]) in order to achieve good performance. It will probably also be necessary to use asynchronous message passing rather than the synchronous communication used on GRIP.

References

[Gro93] van Groningen J, “Optimizing Mark-Scan Garbage Collection”, *Proc. 1993 Workshop of Implementation of Functional Programming Languages*, Nijmegen, 1993.

[HP92] Hammond K and Peyton Jones SL, "Profiling Scheduling Strategies on the GRIP Multiprocessor", *Proc. 1992 Workshop on Parallel Implementation of Functional Languages, Aachen, September 1992*.

[Mat93a] Mattson JS, "An Effective Speculative Evaluation Technique for Parallel Graph Reduction", *PhD Thesis* University of California at San Diego, February, 1993.

[Mat93b] Mattson JS, "Performance of Parallel Schedulers for Distributed Graph Reduction", *Proc. 1993 Workshop on Impl. of Functl. Prog. Langs.,* Nijmegen, September, 1993.

[Nik93] Nikhil RS, *Private communication to the authors*. September 2nd 1993.

[Par92] Partridge A, "Speculative Evaluation for Lazy Functional Languages", *PhD Thesis*, University of Tasmania, Australia.

[PCHS87] Peyton Jones SL, Clack C, Salkild J and Hardie M, "GRIP — a High Performance Architecture for Parallel Graph Reduction", *Proc. FPCA 87*, Portland, Or., pp. 98—112.

[PVM93] *Parallel Virtual Machine Reference Manual, Version 3.2*, Internal Report, University of Tennessee, August 1993.

[San91] Sansom P, "Dual-Mode Garbage Collection", *Proc. 1991 Workshop on Parallel Implementation of Functional Prog.Langs.*, Southampton, Sept. 1991, pp. 283–310.

Appendix I: IMU Code

```
main(argc,argv)
int  argc;
char  **argv;
{
       …
         InitIMU(…);
         DoIMUOp();
          TerminateIMU();
}


/*
       DoIMUOp schedules an IMU operation.   If necessary, the hunger
       status is sent to the controlled PEs.
*/

int  rbufid,  nbytes,  opcode,  sender_id;

DoIMUOp()
{

    while(TRUE)
     {
          if(rbufid  =  pvm_probe(ANY_TASK,ANY_OPCODE))  <=  0)
            BroadcastHunger();

          rbufid  =  pvm_recv(ANY_TASK,ANY_OPCODE);
          pvm_bufinfo(  rbufid,  &nbytes,  &opcode,  &sender_id  );

         switch(opcode)
         {
             case PM_ALLOC:
                IMUAllocate();
              break;

           …
         }
     }
}


/*
       IMUAllocate allocates an UPDATABLE node in the IMU, and returns
      the GA of the new object or a FULL reply.
*/

IMUAllocate()
{
    GLOBAL_ADDR  object  =  Alloc_Words(UPDATABLE);
   if(object  >  0)
       Reply1(object);
   else
      ReplyFull();
}


/*
     Alloc_Words allocates a new object and fills it with the contents
     of the incoming packet.   The tag is either NF or UPDATABLE.
*/

GLOBAL_ADDR   Alloc_Words(tag)
{
    IMU_ADDR  object;
    int  size,  pointers,  vhs;
```

```
    pvm_upkint(&size,1,1);

    if((object=ALLOCATE_HEAP(size+IMU_HS+1))  >=  0)
     {
         TAG(object)  =  tag;
         SIZE(object)  =  size;
           pvm_upkint(&PTRS(object),size+2,1);
          return(MKGLOBAL(object));
     }
   else
      return(0);
}


/*
      Send a message with opcode opcode to task.
*/

SendTo(task,opcode)
int  task,  opcode;
{
    pvm_initsend(PvmDataDefault);
    pvm_send(task,opcode);
}

/*
      Send a message with opcode opcode and one data word data to task.
*/


SendTo1(task,opcode,data)
int  task,  opcode,  data;
{
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&data,1,1);
    pvm_send(task,opcode);
}


/*
     Send a normal reply.
*/

ReplyOK()
{
    SendTo(sender_id,REPLY_OK);
}


/*
      Send a FULL reply.
*/

ReplyFull()
{
    SendTo(sender_id,REPLY_FULL);
}


/*
      Send a reply with one data word.
*/

Reply1(data)
int  data;
{
    SendTo1(sender_id,REPLY_OK,data);
}
```

Appendix II: A Sample Run

The following trace was taken from running a simple program on multiple IMUs and PEs, with allocation and sparking, but without global updating or remote threading. The program thus ran sequentially on the main PE. The source program was a trivial *nfib*:

```
main = print (nfib 3)
nfib :: Int -> Int
nfib n = if n < 2 then 1 else n2 `par` nf1 `seq` nf2 + n1 + 1
           where nf1 = nfib(n-1)
                 nf2 = nfib(n-2)
```

The execution trace follows:

```
[t80040000]  [t40089]  IMU 1:  initialising...
[t80040000]  [t40085]  PE 3:   WaitForPEOp: op = INIT, who = ANY
[t80040000]  [t40086]  PE 2:   WaitForPEOp: op = INIT, who = ANY
[t80040000]  [t40087]  PE 1:   WaitForPEOp: op = INIT, who = ANY
[t80040000]  [t40088]  PE 0:   WaitForPEOp: op = INIT, who = ANY
[t80040000]  [t40089]  IMU 1:  4 local PEs, first is #0
[t80040000]  [t40089]  IMU 1:  PE[0] =   [t40088]
[t80040000]  [t40089]  IMU 1:  PE[1] =   [t40087]
[t80040000]  [t40089]  IMU 1:  PE[2] =   [t40086]
[t80040000]  [t40089]  IMU 1:  PE[3] =   [t40085]
[t80040000]  [t40088]  PE 0:   Exception: Received MAIN, sender 40084
[t80040000]  [t40088]  PE 0:   Starting main program...
[t80040000]  [t40085]  PE 3:   GET_THREAD sent from 40085 to 40089
[t80040000]  [t40087]  PE 1:   GET_THREAD sent from 40087 to 40089
[t80040000]  [t40089]  IMU 1:  GET_THREAD from [t40085]
[t80040000]  [t40089]  IMU 1:  GET_THREAD from [t40087]
[t80040000]  [t40086]  PE 2:   GET_THREAD sent from 40086 to 40089
[t80040000]  [t40089]  IMU 1:  GET_THREAD from [t40086]
[t80040000]  [t40088]  PE 0:   FlushEnt: SPEC: Flushing
[t80040000]  [t40088]  PE 0:   Node: 0x7e9b4
[t80040000]  [t40088]  PE 0:    [GA: 0x0]
[t80040000]  [t40088]  PE 0:    IP: (SPECU), size 2, 1 ptrs
[t80040000]  [t40088]  PE 0:   Data:
[t80040000]  [t40088]  PE 0:     0x0007e9a4[P]
[t80040000] [t40088] PE 0:                0[D]
[t80040000]  [t40088]  PE 0:   Jumping to Flush Entry Routine at 0x3d370
[t80040000]  [t40088]  PE 0:   FlushEnt: SPEC0:   Flushing
[t80040000]  [t40088]  PE 0:   Node: 0x7e9a4
[t80040000]  [t40088]  PE 0:    [GA: 0x0] IP: (SPECU), size 2, 0 ptrs
[t80040000] [t40088] PE 0:  Data:            3[D]          0[D]
[t80040000]  [t40088]  PE 0:   Jumping to Flush Routine at 0x3e660
[t80040000]  [t40088]  PE 0:   FlushSPEC_0_NU: Flushing Node: 0x7e9a4
                                   [GA: 0x0] IP: (SPECU), size 2, 0 ptrs
[t80040000] [t40088] PE 0:  Data:            3[D]          0[D]
[t80040000]  [t40089]  IMU 1:  ALLOC from [t40088]
[t80040000]  [t40089]  IMU 1:  Request for 2 Words
[t80040000]  [t40089]  IMU 1:  Allocating 6 words at 0x0
[t80040000]  [t40089]  IMU 1:  Sent Reply [10000000] to [t40088]
[t80040000]  [t40088]  PE 0:   Received Reply, size 4 from [t40089]
[t80040000]  [t40088]  PE 0:   Flushed 7e9a4, GA 10000000
[t80040000]  [t40088]  PE 0:   Jumping to Flush Routine at 0x3dd5c
[t80040000]  [t40088]  PE 0:   FlushSPEC_M_NU: Flushing Node: 0x7e9b4
                                   [GA: 0x1] IP: (SPECU), size 2, 1 ptrs
[t80040000] [t40088] PE 0:  Data:  0x0007e9a4[P]           0[D]
[t80040000]  [t40089]  IMU 1:  ALLOC from [t40088]
[t80040000]  [t40089]  IMU 1:  Request for 2 Words
[t80040000]  [t40089]  IMU 1:  Allocating 6 words at 0x6
[t80040000]  [t40089]  IMU 1:  Sent Reply [10000006] to [t40088]
[t80040000]  [t40088]  PE 0:   Received Reply, size 4 from [t40089]
[t80040000]  [t40088]  PE 0:   Flushed 7e9b4, GA 10000006
```

```
[t80040000]  [t40088]  PE  0:    Jumping  to  Flush  Routine  at  0x3d9f4
[t80040000]  [t40088]  PE  0:     SPARK  [10000000,10000006]
                                      sent from 40088 to 40089
[t80040000]  [t40089]  IMU  1:  SPARK  10000000  10000006  from  [t40088]
[t80040000]  [t40089]  IMU  1:  Sent  Reply  (0)  to  [t40088]
[t80040000]  [t40088]  PE  0:    Received  Reply  (0),  size  0  from  [t40089]
[t80040000]  [t40088]  PE  0:     FlushEnt:  SPEC:  Flushing  Node:  0x7e9e4
                                      [GA: 0x0] IP: (SPECU), size 2, 1 ptrs
[t80040000]  [t40088]  PE  0:   Data:   0x0007e9d4[P]            0[D]
[t80040000]  [t40088]  PE  0:    Jumping  to  Flush  Entry  Routine  at  0x3d370
[t80040000]  [t40088]  PE  0:    FlushEnt:  SPEC0:    Flushing  Node:  0x7e9d4
                                    [GA: 0x0] IP: (SPECU), size 2, 0 ptrs
[t80040000]  [t40088]  PE  0:   Data:            2[D]           0[D]
[t80040000]  [t40088]  PE  0:   Data:            2[D]           0[D]
[t80040000]  [t40089]  IMU  1:  ALLOC  from  [t40088]
[t80040000]  [t40089]  IMU  1:  Request  for  2  Words
[t80040000]  [t40089]  IMU  1:  Allocating  6  words  at  0xc
[t80040000]  [t40089]  IMU  1:  Sent  Reply  [1000000c]  to  [t40088]
[t80040000]  [t40088]  PE  0:    Received  Reply,  size  4  from  [t40089]
[t80040000]  [t40088]  PE  0:    Flushed  7e9d4,  GA  1000000c
[t80040000]  [t40088]  PE  0:    Jumping  to  Flush  Routine  at  0x3dd5c
[t80040000]  [t40088]  PE  0:     FlushSPEC_M_NU:  Flushing  Node:  0x7e9e4
                                      [GA: 0x1] IP: (SPECU), size 2, 1 ptrs
[t80040000]  [t40088]  PE  0:   Data:   0x0007e9d4[P]            0[D]
[t80040000]  [t40089]  IMU  1:  ALLOC  from  [t40088]
[t80040000]  [t40089]  IMU  1:  Request  for  2  Words
[t80040000]  [t40089]  IMU  1:  Allocating  6  words  at  0x12
[t80040000]  [t40089]  IMU  1:  Sent  Reply  [10000012]  to  [t40088]
[t80040000]  [t40088]  PE  0:    Received  Reply,  size  4  from  [t40089]
[t80040000]  [t40088]  PE  0:    Flushed  7e9e4,  GA  10000012
[t80040000]  [t40088]  PE  0:    Jumping  to  Flush  Routine  at  0x3d9f4
[t80040000]  [t40088]  PE  0:     SPARK  [1000000c,10000012]
                                      sent from 40088 to 40089
[t80040000]  [t40089]  IMU  1:  SPARK  1000000c  10000012  from  [t40088]
[t80040000]  [t40089]  IMU  1:  Sent  Reply  to  [t40088]
[t80040000]  [t40088]  PE  0:    Received  Reply,  size  0  from  [t40089]
[t80040000]  [t40088]  PE  0:    ENTER  [1000000c]  sent  from  40088  to  40089
[t80040000]  [t40089]  IMU  1:  ENTER  from  [t40088]
[t80040000]  [t40089]  IMU  1:  ENTER  1000000c
[t80040000]  [t40089]  IMU  1:  [1000000c]:  UPDATABLE  size  2  ptrs  0
                                         IP (SPECU) 0x2 0x0
[t80040000]  [t40088]  PE  0:    Received  Reply,  size  20  from  [t40089]
[t80040000]  [t40088]  PE  0:    ENTER  [10000000]  sent  from  40088  to  40089
[t80040000]  [t40089]  IMU  1:  ENTER  10000000  from  [t40088]
[t80040000]  [t40089]  IMU  1:  [10000000]:  UPDATABLE  size  2  ptrs  0
                                         IP (SPECU) 0x3 0x0
[t80040000]  [t40088]  PE  0:    Received  Reply,  size  20  from  [t40089]
[t80040000]  [t40088]  PE  0:    5
[t80040000]  [t40088]  PE  0:    PEOp:  FINISH  sent  from  40088  to  40084
[t80040000]  [t40088]  PE  0:     execution  time:  0.06ms
[t80040000]  [t40085]  PE  3:    Received  FINISH,  size  0  from  [t40084]
[t80040000]  [t40085]  PE  3:     Exception:  Received  FINISH,  sender  40084
[t80040000]  [t40085]  PE  3:    FINISH  sent  from  40085  to  40084
[t80040000]  [t40085]  PE  3:    WaitForPEOp:  op  =  FINISH,  who  =  40084
[t80040000]  [t40086]  PE  2:    Received  FINISH,  size  0  from  [t40084]
[t80040000]  [t40086]  PE  2:     Exception:  Received  FINISH,  sender  40084
[t80040000]  [t40086]  PE  2:    FINISH  sent  from  40086  to  40084
[t80040000]  [t40086]  PE  2:    WaitForPEOp:  op  =  FINISH,  who  =  40084
[t80040000]  [t40087]  PE  1:    Received  FINISH,  size  0  from  [t40084]
[t80040000]  [t40087]  PE  1:     Exception:  Received  FINISH,  sender  40084
[t80040000]  [t40087]  PE  1:    FINISH  sent  from  40087  to  40084
[t80040000]  [t40087]  PE  1:    WaitForPEOp:  op  =  FINISH,  who  =  40084
[t80040000]  [t40088]  PE  0:    Received  FINISH,  size  0  from  [t40084]
[t80040000]  [t40088]  PE  0:     Exception:  Received  FINISH,  sender  40084
[t80040000]  [t40088]  PE  0:    FINISH  sent  from  40088  to  40084
[t80040000]  [t40088]  PE  0:    WaitForPEOp:  op  =  FINISH,  who  =  40084
[t80040000]  [t40089]  IMU  1:  FINISH  (8)  from  [t40084]
[t80040000]  [t40089]  IMU  1:  terminating...
```