5 Conclusions and further work

We have tried to show that pairwise shared memory opens up for elegant and efficient programming, but that portability and maintainability aspects command encapsulation. We have then described one possible abstraction which we in due course will go on to implement and evaluate closer. We are particularly interested in the programmability aspects of VNM, and how efficient it can be implemented. We will also look for other ways of achieving the same goals: programmability, maintainability and efficiency.

References

[AS88] Athas and Seitz. Multicomputers: message-passing concurrent computers. *IEEE Computer*, August 1988.

[AT90] Ole John Aske and Øystein Torbjørnsen. Communication on HC16 - a study of methods and performance in a hypercubic network based on dual port RAM. In *Proceedings of The Fifth Distributed Memory Computing Conference*, The University of South Carolina, April 1990.

[BG89] Bjørn Arild W. Baugstø and Jarle Fredrik Greipsland. Parallel sorting methods for large data volumes on a hypercube database computer. In H. Boral and P. Faudemay, editors, *Proceedings from Sixth International Workshop on Database Machines, IWDM '89, Lecture Notes in Computer Science 368*, pages 127 – 141, Springer-Verlag, June 89.

[BLRA80] Kjell Bratbergsengen, Rune Larsen, Oddvar Risnes, and Terje Aandalen. A neighbor connected processor network for performing relational algebra operations. In *The papers of the Fifth Workshop on Computer Architecture for Non-numeric Processing*, pages 96 – 105, ACM: SIGARCH, SIGIR, SIGMOD, in cooperation with IBM and The Ohio State University, March 1980.

[Bra87] Kjell Bratbergsengen. Algebra operations on a parallel computer - performance evaluation. In *Fifth International Workshop on Database Machines*, 1987.

[Bra89] Kjell Bratbergsengen. The development of the parallel database computer HC16-186. In *Proceedings of The Fourth Conference on Hypercubes, Concurrent Computers and Applications*, March 1989.

[FJL*88] Geoffrey C. Fox, Mark A. Johnson, Gregory A. Lyzenga, Steve W. Otto, John K. Salmon, and David W. Walker. *Solving Problems on Concurrent Processors, General Techniques and Regular Problems.* Volume 1, Prentice-Hall International, Inc., 1988.

[HB85] Kai Hwang and Fayé A. Briggs. *Computer Architecture and Parallel Processing. Computer Science Series*, McGraw-Hill, 1985.

[TSF90] Ming-Chit Tam, Jonathan M. Smith, and David J. Farber. A survey of distributed shared memory systems. March 1990. mtam@eniac.seas.upenn.edu, mar. 90, ftp from dsl.cis.upenn.edu.

critical regions. Some kind of locking may still be necessary for atomicity reasons, this issue will be considered closer.

4.2 Shared synchronous variables

In this mode the memory looks just like real shared memory, although the sharing is still limited to two neighbours. All reads and updates will take place in critical regions. Both processors can read at the same time, unless one of them has locked for write. This will provide a means of two-way communication between the nodes.

4.3 Drawbacks

One would often like to pass data by reference, i.e. just send a pointer. However, a pointer into the address-space of one node will not mean much on the other side of the connection. For simplicity, this scheme should only be used for informing of data by value, not by reference. This is a in some cases severly limitation compared with the SM approach, where dereferencing will be handled automatically.

4.4 Advantages

- VNM underlines the hypercubic topology of the HC-machines, thus helping to structure programs in a systematic way.
- The programming paradigm can in some applications be more natural than the typical message sending/receiving model of distributed memory computers.
- Since this programming model is somewhat abstract, it can survive into a multi-process scheme, providing the semantics of DPRAM without the practical problems.
- The implementation will deal with the atomicity of updates, leaving the user with one problem less.
- VNM does not assume anything about the number of updates or reads of a variable, whereas n sends in the send/receive model will assume n receives.
- A new value of a variable will only be shipped across when it has actually been changed. For applications where the number of reads is higher than the number of writes, this is an advantage. However, programs reading very rarely but writing all the time, will loose out on this approach.
- It should be possible to make an implementation of the VNM scheme execute faster than "Distributed Shared Memory", due to the reduced generality. The number of processors sharing a part of memory is limited to two, and the amount of memory shared is kept under stricter control.
- Restricted use of the VNM (where everything fits in the physical DPRAM) will allow for very fast executions.
- Applications using more VNM than there is actually physical DPRAM available will still benefit from the abstract model, and can still be executed. Clever allocation schemes in the physical DPRAM will give high performance, and if more DPRAM gets available, the performance will increase even more. The allocation may also be altered at runtime to improve performance.
- This programming paradigm is useful on DM machines, which scale better than the SM class of machines. VNM does not limit this scaling.
- We have described the VNM scheme as a connection between neighbours. In fact, it can also be used between any pair of processors, but we fear that this may invalidate some assumptions about order of updates, and the logic of time ordering. However, VNM can also provide a useful way of interprocess or interjob communication *within* one node. We will look into the possibilities and consequences of this.

Portability to other architectures

Some program systems may be worthwhile programming for one specific architecture. If the need for computing power or recoverability is great enough, and the benefits achieved by exploiting the architecture significant, this decision may make sense. In many cases, however, one wishes to use a program on different architectures, so that portability remains an important issue.

As long as the DPRAM is used as a pure communications channel, the programmer will see an interface offering sends and receives. Programs written using these calls can then trivially be ported to other architectures. Programs exploiting the special features of the DPRAM, (direct access to common memory and interprocessor interrupts), will have to be rewritten, since these concepts are architecture-specific. This is a very serious portability restriction, and may limit the use of the DPRAM.

Maintenance and porting to similar architectures

The physical DPRAM will be limited in size, the current version has 4 KB of shared memory over each neighbour connection. Programs using facilities beyond send/receive will depend on this size. Different machines from this machine series will have DPRAMs of different sizes, thus restricting portability within the same machine series. Even when maintaining programs on one machine, datastructures may be altered and suddenly not fit in the DPRAM area any more.

Usefulness in complex systems

The software engineering aspects of this are gloomy, to say the least. The last nail in the coffin is hammered in as multiple processes are introduced on each node. The processes and the operating system will then at any given time not know anything about the total wishes for DPRAM for the system as a whole, at some time in the future. Dedicated use of this memory area would have to be allocated on the basis of heuristics or quotas, and would probably be far from optimal.

As long as the DPRAM is used as a pure communication channel, the applications programmer will not be aware of the details of the DPRAM, but programs using special properties of the architecture are essentially not portable.

4 Virtual Neighbour-shared Memory

We have been looking for ways to exploit our special hardware. The experience with the earlier machines have shown some useful methods of programming, as mentioned above, and we want to let these survive into later versions.

We would like to give the programmer the functionality of the DPRAM, but encapsulated and abstracted in a suitable way. The goal is to keep a good correspondence between the abstraction and the underlying model, to allow efficient implementations. At the same time it should be general enough to be useful. We therefore introduce the concept of "Virtual Neighbour-shared Memory"(VNM).

The program will declare memory in the normal way, and may then use a system call to make it virtual dual-ported. From then on the memory can be used just as any other, the operating system will trap accesses and take care of updates and locking. This memory will logically behave just as if it was shared. The operating system is also free to move the actual allocation of the shared segment around. If it actually resides in physical DPRAM, control is trivial. If not, the system will keep different copies in the local memory of the processors in question consistent, using suitable protocols. This enables the operating system to reallocate to improve the performance of the system, e.g. by placing VNM regions in physical DPRAM during times of heavy usage. All the effects of this (re-)allocation should be invisible to the user. We will also let the programmer terminate the "dual-portedness" of any specific part of memory.

4.1 One-way VNM

In this mode we let two nodes share an area of memory, or at least think that they do. Both nodes may read from this memory area, but one node may write to it. This will provide a way of peeking into a neighbours work-space, and will be useful for "status control" and one-way messages. Updates will be performed asynchronously, but all updates from one processor to one specific neighbour will be performed in sequence. Since only one processor can update this area, there is no need for

[TSF90]). Since the HC16 machines offer the functionality of the message-passing system on these multi-computers, we could also offer DSM.

FIFO emulation

The DPRAM can simulate one or more interprocessor FIFOs. This will require some internal variables and control effort. In short, some of the work done in the hardware of the FIFO will have to be done in software when replacing FIFOs with DPRAM. This effort can be hidden from the user. Since the DPRAM is more general than the FIFO, functionality can be added, like partial resets or reuse of data.

Intermemory access

The Cross-8 and HC16-186 machines typically had one process running on each node. This allowed explicit control of the dual-ported neighbour-shared RAM. One could place a variable at a physical address known to be in the shared area, and use this variable from both sides of the connection. This gives interprocessor communication at the cost of memory access, clearly a lot better than anything else offered by multi-computers. These shared memory areas give the user full freedom to either implement some kind of protocol of his own, or just see it as memory. We have, in some iterative programs, put "border-line variables" in the shared memory, without paying any attention to the rate of updates. This will work in cases where the program works towards some kind of convergence, and at any time will prefer the latest value.

Specific Interprocessor Interrupts

The DPRAM have some locations where writing will give an interrupt to the neighbour. Writing X into a location gives interrupt X on the other side. This can be exploited to give synchronous communication or anything else the programmer can think of.

Broadcasts

As these machines are based on the transfer of messages, no hardware broadcast facilities are available. Nevertheless, software can manage this, removing redundant communication, to give the user an impression of broadcasts. Without the special hardware support, this does not give particularly cheap communication, and the broadcast messages will not reach all nodes at the same time. In general we have preferred using messages with specified recipients.

3.2 Considerations

Speed and usability

As we have tried to show above, the DPRAM concept gives the programmer tremendous flexibility, and very simple communication. It is also very fast [AT90], we have shown the communication of a Intel 80186-based machine to perform significantly better in many cases than several other system with much stronger processors, the iPSC2 being one.

Topology

The hypercube topology has quite a few nice properties. For an early evaluation see [BLRA80]. Let the number of nodes be N, and the dimensions of the cube D. Then $D = log_2(N)$. The average number of hops from one node to any other is low: $D/2$. The number of connections on each node and the maximum distance from one node to any other is D, i.e. $log_2(N)$. The number of arcs in the cube is fairly low, $(D * 2^{D-1})$. As we see, the topology scales well.

Equally important, a lot of very different and general problems can be mapped on to the hypercubic topology, see [FJL*88]. The combination of some nearest neighbour connections with some connections to nodes further away is shown to provide a very natural and intuitive model. Among problems demonstrated to fit well we find things like general matrix mathematics, finite element methods, fast Fourier transforms, sorting, and many others. The structuring of problems in a fashion well suited for hypercubic communication has been a major key to the success of some of our own software projects, like those described in [BG89] and [Bra87].

	Distributed Memory	Shared Memory
Communication	Expensive, by messages	Cheap, by reference
Jobs	Separate queues	Common queue
Parallelisation	Explicit	Trivial, with task queue
Memory	Large, separate	Common
Memory Contention	No	Increases with number of processors
Scaling	Well	Limited
Synchronisation	By messages	In critical regions

2.1 Distributed Memory

The nodes of computers in this class have no shared memory. Each processing node has its own memory, and all communication is based on messages.

All communication takes the form of messages sent across the communication channels. Every message has to be composed, sent, received and decomposed, involving work on both sides of the communication. The communication will be relatively expensive, and the message-passing structure restricts the programmer severly. This limitation can be abstracted, but the delay and cost will still be present.

Since each node has its own memory, the accumulated local memory may be of a substantial size, i.e. the number of nodes times the local memory of each node. All this memory is controlled locally, and can be accessed without memory contention. The local memory accesses can therefore be quick and cheap.

Small configurations of these machines may be a little slower than shared memory machines, but they will scale better into larger configurations.

2.2 Shared Memory

Since shared memory (SM) computers operate in one common memory-area, they may pass data by reference. This saves a lot of data copying in many cases. They can also have one shared job-queue, so that all processors gets its work out of one task pool. This means that a n-processor machine theoretically can run n times faster than a one-processor machine, without any need for explicit parallelisation. That is, a one-process job will run at the speed of one processor, no matter how many processors it has available. A program written as a set of independent tasks or using asynchronous operating system calls will gain speed. Also, the execution of several one-process programs concurrently will speed up the system as a whole. Some SM machines also have local memory, e.g. the C.mmp multiprocessor, reducing some of the scaling problems (see [HB85]).

However, as the number of processors increases, all shared resources can turn into bottle-necks, severly limiting the speedup of the machine. For this reason SM-machines usually keep the number of processors down to a few or in the order of tens.

3 Pairwise shared RAM

3.1 One-process per node programming

A parallel computer with neighbour-shared RAM can be programmed in different ways. This section describes the ones we have thought of. On the early machines, MIMD behaviour, intermemory access and interprocessor interrupts have been most frequently used.

MIMD behaviour

One may choose to view the shared memory as a mere communication channel, and use it for sending messages across. The DPRAMs will have a superset of the functionality of communication channels, and can behave exactly like them. Anything that can be implemented on a pure distributed-memory message-passing multi-computer can be implemented on top of neighbour-shared RAM.

DSM behaviour

Multi-computers can offer "Distributed Shared Memory" (simulating a shared-memory architecture on a distributed-memory machine) using messages to control the updates to the shared memory(see

Programming the Hypercube 16/386 Database Computer; Utilizing Pairwise Shared Memory

Petter Moe
Program Systems Group
Department of Computer Systems
The Norwegian Institute of Technology
N-7034 Trondheim, Norway
e-mail: petter@idt.unit.no

October 18, 1990

Abstract

In this paper we describe the database machines built at the University of Trondheim, and focus in on their communication facilities. We then discuss the programming model, showing its usefulness, elegance and also its implications from a software engineering point of view. The most serious of these include a potential lack of portability and maintainability. We will conclude by suggesting one way of keeping the benefits while avoiding the problems, using a concept which we have called "Virtual Neighbour-shared Memory".

1 The Hypercube-16 project

The HC16 project is a follow-up to the CROSS8 project.[1] The purpose of the project is to look into methods and algorithms used in database systems on multicomputer systems. We have also looked into some related fields, e.g. image processing, ray-tracing, numeric operations, parallel sorting, and parallel programming environments.

The project group has so far built three prototype machines: one eight-node 80186-based cross-bar computer, one sixteen-node 80186 hypercube computer, and now an 80386-based machine, also with sixteen nodes and hypercubic interconnections.

Each node of these computers has its own memory and its own disk. Database relations are split into fragments, with one part stored on the local disk of each node. During database operations, the tuples of relations may be redistributed, based on a hashing scheme. The project and some of the methods are described in [Bra89].

The sixteen nodes of the HC-machines (HC16-186 and HC16-386) are connected as a four-dimensional hypercube. This means that each node has four connections, and the longest shortest path between any pair of processors will be four "hops". Each interconnection is a chip of dual-ported memory (DPRAM), where both nodes can read and write in all locations. On the HC16-186 the size of each DPRAM is two kilobytes, on the HC16-386 each chip can hold four kilobytes.

The original idea of building a dedicated database server has now been extended into building a server which also has the functionality of a general parallel computer.

2 Communication Hardware, programming models

We will discuss two different classes of parallel computers, and briefly mention some of their stronger and weaker sides. The discussion is limited to coarse-grained computers, and exotic/intermediate designs are left out. [AS88] clarifies and argues that these types of machines are similar enough to run programs written for the other. However, a program exploiting the facilities of one architecture will often not achieve a very good performance if executed on a different one. The purpose of the discussion is merely to prepare for a description of the neighbour-shared RAM class of machines.

[1] This work is supported by the Norwegian Government Research Agency, NTNF