Towards a Shared Memory Hypercube

Donald C. Lindsay
28 November 1988
CMU-CS-88-190

Donald C. Lindsay
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890
lindsay@k.gp.cs.cmu.edu

Towards a Shared Memory Hypercube

Donald C. Lindsay
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890
lindsay@k.gp.cs.cmu.edu

Abstract

Early generation hypercube computers have shown great promise, but only message-based programs have been successful in exploiting the potential of these machines. The article shows that shared memory programming techniques can be efficient on a conventional hypercube with appropriate communications support. Specific hardware features are proposed, based on double-ended circuit transactions.

Introduction

Although hypercube computers were constructed in the 1970's *[6]*, they have only recently become attractive. They are now seen as a way to assemble our increasingly powerful board-level (or chip-level) microsystems into products having high aggregate bandwidths. This avoids the mainframe approaches to high computational bandwidth, high memory bandwidth, and high I/O bandwidth, all of which involve diminishing returns. ‡ There is a bright promise that hypercubes can be scaled to large sizes without difficulty. The high speed and sophistication of recent microprocessors shows that a hypercube may be composed of quite powerful elements, instead of being "an army of ants". Hypercubes are already supporting large disk farms, and it seems possible to eliminate disk cabling by dispersing today's increasingly tiny disk drives throughout a machine. A large number of efforts have shown that hypercubes can be programmed, and even time-shared, and that many applications can exploit the power of these machines *[8,9,15]*.

If there is a problem, it is that the early generation machines were unforgiving. For example, performance was quite dependent on the programmer arranging that messages mostly went to directly adjacent nodes. In order to achieve this high locality, tasks and data had to be mapped onto the machine with great care. Although there is now a record of experience and success in this area, it is clear that this was limiting.

In a similar vein, the use of message based programming paradigms has often resulted in high latencies, which can easily impact performance if the nodes are not multiprogrammed. It has become important to measure how synchronous an application can be, since highly synchronous codes can carefully overlap all message waits with computation.

Message based programming is not without drawbacks, since control transfer must occur in code which would otherwise only do data transfer. Messages impose a style on programming, which can be intrusive on the application logic. (For example, porting a shared-memory program involves recasting data references into pairs of procedure calls - one to start the reference, and one to obtain the answer. The programmer may want to distort the logic so as to introduce computation between the two calls.)

Communications Alternatives

All first-generation hypercubes used store-and-forward communication. The hardware didn't have to know about forwarding, since it simply

‡ Some examples: almost any cooling method; thin film interconnection; the extensive crossbars of a multiported, highly interleaved memory.

interrupted the processor at each intermediate node. However, this reduced performance, by consuming both processor cycles and memory cycles at each stage. One solution is to move message buffering out of the node memory, but then the communications hardware at each node must have increased resources (local RAM), or else message sizes must have a very small upper bound. Such bounds will cause message fragmentation. This has costs, which may be large if the destination must deal with reassembling the fragments (which may arrive out of order). And, in any case, buffered schemes have particularly high latency if there is any network contention.

Circuit switching is an attractive alternative. In these schemes, a path to the destination is constructed, and later "torn down" when the message has been sent. These schemes require a data crossbar at each node, which is quite easy to build. (The node must switch six to twelve paths, which may be as little as one wire wide. Commercially available nonblocking crossbar chips are as large as 32 x 32 bits.)

At first glance, circuit switching would appear to offer absolutely minimum latency, particularly when a message must travel several hops. After all, the data simply flows through the intermediates, and is not stored until it reaches the destination. However, the circuit setup phase can be done in a variety of ways. In some systems, a fixed path is chosen, and the circuit is not established until all the elements of the path are simultaneously free *[2,13]*. A more desirable method would attempt to do dynamic adaptive routing. This would retain the (potential) advantages of store-and-forward, such as tolerance of node failures, tolerance of network "hot spots", self-adaptation to missing hardware, and so on.

The question about any such routing method, is whether it can achieve these advantages, without paying too high a price in search times, link traffic, or risk of deadlock. The author has written a simulation which uses the **K(K-1)** method, since the Caltech/JPL Concurrent Processor Group is constructing hardware support for this method *[3,14]*. Some parameters of the simulation are listed in Table 1, and Figure 1 shows some encouraging results. An overall conclusion is that the **K(K-1)** method (described in the Appendix) appears to have the claimed properties of stability and low latency.

Coherence

A memory scheme is said to be coherent if the value returned by a LOAD instruction is always the value stored by the last STORE instruction with the same address *[5]*. This is obviously a crucial property, but on a multiprocessor, this definition is actually imprecise, and we get into a realm of "strong" and "weak" orderings. When an application is spread across unshared memories, this property is in some trouble. (For example, messages may cross in flight.)

This does not cause semantics trouble, since message protocols can be found for any desired semantic property. The problem is one of efficiency, since messages not only travel, but must be sent, received, and decoded, at the cost of memory traffic, procedure calls, kernel entries, and task switches. It is in this manner that a simple reference may wind up costing as much as several milliseconds.

This cost does not have to be paid for every variable. However, it certainly has to be paid for every variable which synchronizes the application's tasks.

Double Ended Circuits

A critical aspect of synchronization is that it involves a double-ended flow of information *[11]*. (Note, for example, the *test-and-set* primitive, commonly found in instruction sets. This instruction leaves data in two places, not in one.) Given that a circuit has been established between two nodes, it can reasonably be held open for some short time, and data flow can occur in either direction, or in both, promptly and naturally.

The statement above glosses over some aspects of implementation. It may be that each bidirectional channel is built as two unidirectional channels, as on the NCUBE/ten and iPSC/2 *[10,13]*. (In this case, the setup phase must instruct all participating nodes that both channels must be seized.) It may be that channels need to be "turned around", in which case participating nodes must respond to backchannel status information, or to per-channel state

machines, or perhaps respond to control information embedded in the data *[13].* In general, these issues appear to have adequate solutions. I therefore conclude that it is both feasible and reasonable to use the built circuit in one direction, and then in the other direction, before tearing the circuit down.

Test and Set

Given that the communications system offers double-ended circuits, the first and most obvious idea is to implement some flavor of test-and-set. A simulation of this was run, and the distribution of latency found by this simulation is shown in Figure 2. It was assumed that twenty bytes of information are sent, and that the same amount is returned. (This number was chosen since it seemed adequate to convey a value: a virtual address: an address space identifier: a serial number: and a checksum.) It was also assumed that the destination could perform the test-and-set (or compare-and-swap, or fetch-and-add) locally, in four microseconds. (This number was chosen since it allows enough memory cycles to perform table walks.) As the figure shows, the simulated 128-node system can support some tens of thousands of these synchronizations per second per node. As a general conclusion, it seems fair to say that the simulated system is very efficient.

Since the **K(K-1)** method propagates a 32 bit header during the setup phase, it might be possible to (conditionally) extend this header by twenty bytes, with the aim of speeding up these transactions. This piggybacked implementation has also been simulated, and essentially it reduced the modal latency from nine microseconds to seven. The cost of this scheme would be increased complexity in the communications logic, plus extra buffering to hold the header. (It was assumed that the logic is smart enough to begin acting in the normal way as soon as the normal header has arrived, but without waiting for the next twenty bytes to arrive.)

The efficiency of this operation is not heavily dependent on the link bandwidth. If the simulated bandwidth is reduced by a factor of eight, then the various measures of latency increase by less than a factor of four. If the piggybacked operation is not worthwhile in a specific real system, it would probably be because the system's software overheads are completely dominant. We shall return to this topic in the section on protocols.

Generalizing

We have argued that double-ended circuit transactions between non-adjacent nodes can have efficient hardware support. It seems easy enough for a communications chip to perform a test-and-set at a physical memory address: after all, the chip will have DMA, and gate arrays of 100,000 gates are now on the market. But in a sophisticated system, test-and-set is not enough, and physical addresses are not enough. The chip may have to deal with virtual circuits, channel identifiers, and serial numbers. The operating system may require other operations, such as compare-and-swap or fetch-and-add. In short, programmability is needed. If the communications chip is microcoded, then this useful generality can be achieved. If the microcode can be easily modified, then the functions can be modified in the light of suggestions from the operating system programmers, and from the applications programmers. The communications system can also be more readily adapted to suit different node processors.

No matter how general the communications logic becomes, there may be limitations on its action. The chip might receive virtual addresses, and need to do MMU translations. It may need to manipulate the processor's cache - for example, to cause cache flushes. Therefore, the needed generality may be best provided by interrupting the node's processor, which is not limited in this regard. This approach gives access, not only to the MMU and to the cache, but to an arithmetic engine.

The interrupt approach has the disadvantage that it places requirements on the processor, both of suitability and of speed. The speed issue is not simply a question of impact on the processing bandwidth. There is also the issue that the circuit is being held, and if it is not released with some promptness, then network contention may result. Also, the application may choose to have all of the nodes referencing a single node, perhaps to obtain work from a centralized queue. As with any potential bottleneck, we must be quite careful about efficiency.

Luckily, some recent microprocessors have quite excellent interrupt characteristics. For example, the Motorola 88000 can store the state of its pipelines sideways into "shadow" registers, and then restore them in parallel *[12]*. This allows a very small (but non-null) interrupt routine to run in a dozen-odd clocks. Even better, this time is not the time until the currently executing process resumes, but the time that the process is slowed by. (This of course assumes that the added memory accesses did not generate cache faults or translation buffer faults.)

If the interrupt approach is viable in a particular system, then the way is open to more general things, such as an Ada rendezvous, or a remote procedure call (RPC). In this case, there is less need of hardware complexity, since system software can be modified as the need arises. The correct semantics of an operation may be much more easily obtained because the return path down the circuit is present and guaranteed. However, for more extended operations, there is incentive to release the circuit, and then send results as a later reply. In any case, longer operations are probably best done by having the interrupt routine (or the communications logic) place them into the processor's scheduling queue. Short operations, such as test-and-set, should be both useful, efficient, and semantically desirable when implemented over a single circuit.

Paging

With the availability of operations such as fetch-and-add, an application can use standard shared-memory techniques with fair efficiency. These conceptually simple mechanisms can do much to ease the software task. However, synchronization is not the only area where the message paradigm intrudes into application logic. On shared memory machines, virtual memory techniques have become increasingly popular, because they do not intrude: they are decoupled from the application level, and hence amenable to tooling, to selectable policies, and in general to the later addition of sophistication.

Given adequate bandwidth, it is not difficult to handle a page fault by having the operating system send a message. The recipient of the message might then return a page as a second (much larger) message. This should work well enough if contention is not high and if the communications network can handle the message (non)locality which the system achieves.

A page fault message is quite important, because the originating process cannot proceed until the page fault has been satisfied. The message latency may not be crucial, particularly if the nodes (which may, after all, be multiprocessors) have many other processes to run. However, fault latency may be directly or indirectly reflected in the time-to-solution, and therefore it is important to explore any possibilities of improvement.

A now obvious idea is that one might *pull* a page, rather than having some other node *push* it. This could be done by adding a communications mode which increases the size of the K(K-1) header, from 32 bits to perhaps 96. Intermediate nodes along the built path would understand that the mode's data flow, although unidirectional, would be backwards. The destination node would obtain the appropriate table entry, and then simply stream the page out of its memory.

Protocols

The idea of pulling data across the hypercube is most attractive for smaller pieces of data, since the reduced overheads of a pull operation are more relevant when the overheads are not dominated by large transfer times. This brings up the idea of *peek* and *poke* operations, which pull or push as little as a single word. Although these operations can replace certain message exchanges, their semantic power is actually much greater. They implement directly shared memory, without any wait for a message server to choose to answer. They allow non-invasive status monitoring, whereby non-busy nodes can check around without putting a processing load on the busy nodes. (Dynamic load balancing could be highly effective in such a system.) They allow ideas such as *remote pages*, wherein a node writes to a page by doing pokes to it. We can imagine an operating system offering replicated pages, which nodes read from by doing local reads, and write to by doing pokes to all the other copies. (This could actually be quite efficient, if writes were sufficiently dominated by reads.) Even better, we can imagine an operating system which replicates pages, or which migrates pages, or which load balances, with a

minimum of explicit involvement from the application logic.

All of the primitives suggested above could be used directly by application programs. However, applications really should live in a virtual memory, and really should go through task scheduling, and so on, for quite good reasons. These reasons are at odds with obtaining flat-out efficiency, and in fact software generality can cause large software overheads. These overheads are a problem if we wish to exploit the high efficiencies of the new primitives.

There is an answer in the idea of protocols. Rather than merely offering primitives, a kernel should offer (and use) entire protocols, supported by low-level code. The reason is that a protocol can "know" what it is doing, and therefore can be heavily tuned. For example, a protocol may be able to do synchronizations with short physical addresses, because only some specific kernel data structure is being addressed. (A load-balancing protocol, or a clock-synchronizing protocol, might do probes involving no addresses at all.) Low-level code can be hand crafted, and can do things like spin (rather than task switch), knowing that an answer will return in microseconds. In short, kernel support which is specific, rather than general, can in many cases realize the potential efficiency of the primitives. For these reasons, the operating system features which we have suggested, could have good or even excellent performance.

Caches

If page faults can operate across the cube, then we would like to know if cache faults can do the same. At first glance, it seems reasonable enough, since the pull of a cache line should have approximately the same performance as a test-and-set operation, and Figure 2 shows this to be quite fast. Cache protocols exist that don't need to do broadcasts *[1]*.

There a number of problem areas. In general, they can all be solved, but they must be solved very efficiently, or else a coarser-grained operation, such as page faulting, will be more effective.

The obvious way to obtain high efficiency is to limit or remove the involvement of software in the most frequent cases. The simplest hardware arrangement would be to have the communications logic in the path of signals from a node's cache/MMU to its main memory. If certain high-order bits of the "physical address" are non-zero, then the communications logic would know that the referenced memory is off-node, and that the reference must be communicated.

The page tables of each node could specify that certain virtual addresses were at the special kind of physical addresses. If the address range of a node is large enough to address the collective physical memories, then a quite simple mapping exists to be used. In any case, there is no need that the "physical address" generated at one node, be interpreted as a physical address by the next node. There are also games that can be played by placing context registers in the communications logic, although these registers might have to be modified at each task switch.

The suggestions presented above are incomplete, because they have not dealt with the actions to be done on a write, or the actions required at the receiving end of a reference. There has to be a protocol for the cache directory information, even if multiple copies are disallowed. Page map information has to propagate from node to node, or else be bundled into load modules.

The evaluation of cache protocols is not completely straightforward. For example, spin loops on semaphore variables can make protocols thrash, but rather than search for a perfect protocol, it might be better to have the users not do that. (They could keep semaphores on uncached pages, and use remote fetch-and-add operations.)

In general, it seems possible that the above suggestions can be extended in a workable manner, without requiring any further hardware support. Simulations of such a solution will have to be detailed, since they will be affected by details of the host (such as the hardware-supported cache line size). Cache simulations should also be trace-driven, and not based on random number generators.

If further hardware were to be added, it would probably be an extra cache, in parallel with a node's normal one, and matching cache control logic that would be integrated with the communi-

cations control. The reason for this is that the normal cache has presumably been optimized under assumptions that are being violated. Adding a communications cache opens up the possibility of choosing a cache organization, and in particular a cache line size, which are well matched to the specific system.

Conclusions

This article has presented some corroboration of published results on **K(K-1)** routing, which appears to be of practical use. Further, implementations of this method seem to be quite reasonable starting points for implementations of double-ended circuit transactions. These operations may allow efficient remote synchronization, remote procedure calls, and a variety of shared-memory and virtual-memory arrangements.

The basic ideas described in this article do not require particularly expensive hardware. The major difficulty would be the complexity of the communications controllers, but the proposed JPL controller would not be affected to the point of requiring two large gate array chips instead of one. The bandwidth requirements are not as high as that used in the simulation, since a lower bandwidth would mostly change the page fault rate that causes network contention. A low bandwidth might affect whether cache faulting could ever be supported, and it would certainly affect the design of any communications cache.

The approach suggested in this article addresses the fundamental barriers to the progress of hypercubes, by allowing more generalized systems, which offer broader programming paradigms. It is hoped that this approach will allow existing and future applications to be more easily and directly cast into efficient parallel forms.

Appendix I

Adaptive Routing

If the message traffic in a network is composed of tiny messages (such as synchronization actions), then the best scheme is to give up easily, free the seized resources (to prevent deadlock), and then retry a very short time later. This is true because a path which is busy at this moment, will be free shortly. It simply does not pay to investigate numerous alternative paths.

If the message traffic is composed of huge messages (such as large data pages), the best scheme is to investigate every possible alternative path, since the paths that don't work now, are going to still not work a moment from now. The number of paths is factorial in the path length K, but it can be best to try all K! paths.

The parallelism of the available hardware determines how much an attempt will lock out attempts by other nodes. The author's simulation assumes that an intermediate node can be attempting N path setups at once, one for each incoming link. This simulated system can occasionally benefit from searching K! paths. When the assumption is changed, and an intermediate node is only allowed one path setup at a time, then excessive searching quickly becomes counterproductive.

In a real system, the messages vary in size, and the mix varies in time. In this situation, the best approach may be the avoidance of extremes. That appears to be the virtue of the scheme described in Appendix II.

Appendix II

Implementing a K(K-1) Search

The "K" family of routing schemes have the property that of K! possible paths, they will search K paths, K(K-1) paths, K(K-1)(K-2) paths, and so on, all the way to the family member which searches all possible paths. The family has the pleasant property that a single implementation offers the entire family. To choose a particular member, it is only necessary to supply a parameter at the originating node.

As described in *[3,7,14]*, a "hyperswitch" node attempts to construct a path by sending a probe to one of its directly connected neighbors. It chooses this neighbor by two criteria. The first criterion is the obvious, that the link is free, and that this step represents progress towards the goal. Progress is easy to determine. The basic, defining property of a hypercube is that one numbers each node so that

nearest neighbors differ in one bit position. Further, the bit position in which the numbers differ, shows which link either should use to talk to the other. This means that the distance between two nodes is simply the Hamming distance of their node numbers. The exclusive-or of the source's node number, and the destination's node number, is thus a bit mask giving the cube dimensions that must be traversed. We may traverse the dimensions in any order: hence there are K! paths.

The second probe criterion is the special contribution of the K family. A history tag is kept, which is a bit mask showing the dimensions that are thought to be workable. A source node creates a fresh history tag, and sends it as part of the probe.

When a destination node receives a probe, it knows this because the incoming header has a destination field which matches the local node number. An *ACK* status signal is sent back (if possible), and this status propagates through any intermediates to the sender, who then knows that the circuit has been built.

When an intermediate node accepts a probe, it chooses an outgoing link which is free, and which is permitted by the incoming history tag. A probe is sent, and if it results in an *ACK*, then the setup is done.

If a node receives a *NAK*, then it will turn off the corresponding bit in its copy of the history tag. It then attempts to choose another outgoing link to probe. If there are no free links permitted by the history tag, then a *NAK* status is returned.

The method described above needs two refinements. The first is a relaxation, whereby a probe that is "near enough" to the destination will reset the history that was sent to it. (This is determined by a distance field in the header, and is the choice that distinguishes between the members of the K family.) The second refinement is a pruning step. It uses the observation that if there are too few one-bits left in the history tag, then further probes will be stopped short of the destination, and therefore needn't be even attempted.

For example, on a 256-node 8-cube, the average message would want to go 4 hops, and this average message could try up to 4(4-1) = 12 paths, by doing up to 21 probe actions. JPL reports that a probe action will be done in well under one microsecond.

References

1. A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An Evaluation of Directory Schemes for Cache Coherence" in *Proceedings of the 15th Annual International Symposium on Computer Architecture* (*Computer Architecture News* Vol. 16 #2 P.280, May 1988)

2. Ametek Computer Research Division, "Series 2010 System General Description", March 1988

3. E. Chow, H. Madan, J. Peterson, D. Grunwald, and D. Reed, "Hyperswitch Network for the Hypercube Computer" in *Proceedings of the 15th Annual International Symposium on Computer Architecture* (*Computer Architecture News* Vol 16 #2 P.90, May 1988)

4. M. Denneau, P. Hochschild, and G. Shichman, "The Switching Network of the TF-1 Parallel Supercomputer" in *Supercomputing*, P.7, Winter 1988

5. M. Dubois, C. Scheurich, and F. Briggs, "Memory Access Buffering In Multiprocessors" in *The 13th Annual International Symposium on Computer Architecture* (*Computer Architecture News* Vol. 14 #2 P.434, June 1986)

6. Electronics, "Hypercube 'federates' Eight Microcomputers", 10 May 1979 P.70

7. D. Grunwald, and D. Reed, "Networks for Parallel Processors: Measurements and Prognostications" in *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications,* 1988, P.610

8. J. Gustafson, G. Montry, and R. Benner, "Development Of Parallel Methods For A 1024-Processor Hypercube" in *SIAM Journal on Scientific and Statistical Computing,* Vol. 9 #4 P.609 (July 1988)

9. J. Gustafson, "Reevaluating Amdahl's Law" in *Comm. ACM* Vol. 31 #5 P.532 (May 1988)

10. J. Hayes, T. Mudge, Q. Stout, S. Colley, and J. Palmer, "A Microprocessor-based Hypercube Supercomputer" in *IEEE Micro*, October 1986 P.6

11. M. Herlihy, "Impossibility and Universality Results for Wait-Free Synchronization", *Technical Report CMU-CS-88-140,* also in *Proceedings of the Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing,* August 1988

12. Motorola Inc., "MC88100 Technical Summary", Document BR588/D, 1988

13. S. Nugent, "The iPSC/2 Direct-Connect Communications Technology" in *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications,* 1988 P.51

14. J. Peterson, E. Chow, and H. Madan, "A High-Speed Message-Driven Communication Architecture" in *Proceedings of the 1988 ACM International Conference on Supercomputing,* Saint-Malo, France, July 1988, P.355

15. C. Seitz, "The Cosmic Cube", in *Comm. ACM* Vol. 28 #1 P.22 (January 1985)

Table 1. Parameters of the simulation, as derived from *[3]*.

Bandwidth per link	128 Mbits/second
Simultaneous DMA streams to/from memory	3
Header size	32 bits
Switch time, per hop	780 nanoseconds

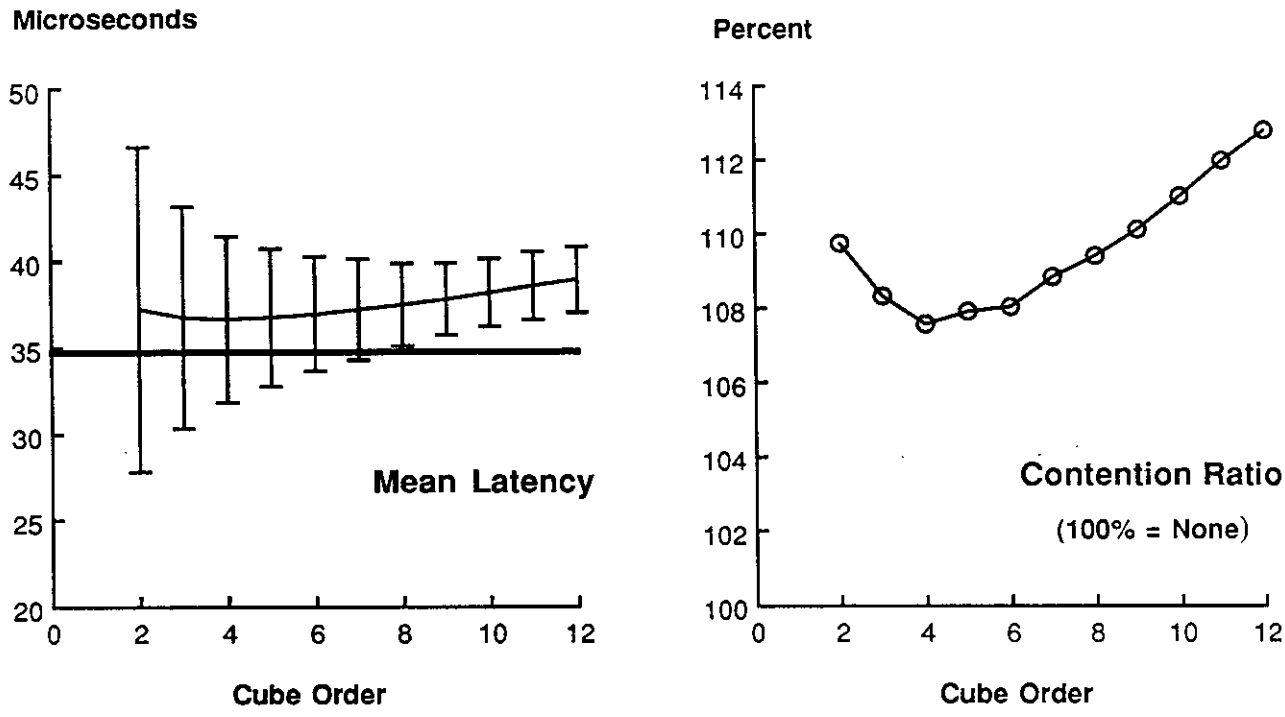

Figure 1. Effects of the hypercube order on message latency. Message size was 512 bytes plus 16 bytes. Messages were generated by a Poisson process, with a mean of 2500 messages per second per node. (Therefore, message interarrival time had a negative-exponential distribution, with a mean of 400 microseconds.) Message source and destination were equiprobable. Each simulation was run until 11,000 messages had exited; the first 1,000 were ignored for statistical purposes. Each data point is the average of two runs, using different random number seeds.

The **left hand graph** represents the mean latency of the messages, in microseconds. (Note that the latency axis does not start at zero.) The horizontal line represents the minimum possible latency, that is, the latency of a message which travels one hop without contention. (Since some of the error bars extend below this, the distribution must be skewed. See, for example, Figure 2.)

The **right hand graph** represents the ratio between the mean latency, and the latency which would have been observed (for that mean message distance) if the network had had no contention.

Note that a cube of order 12 contains 4096 nodes.

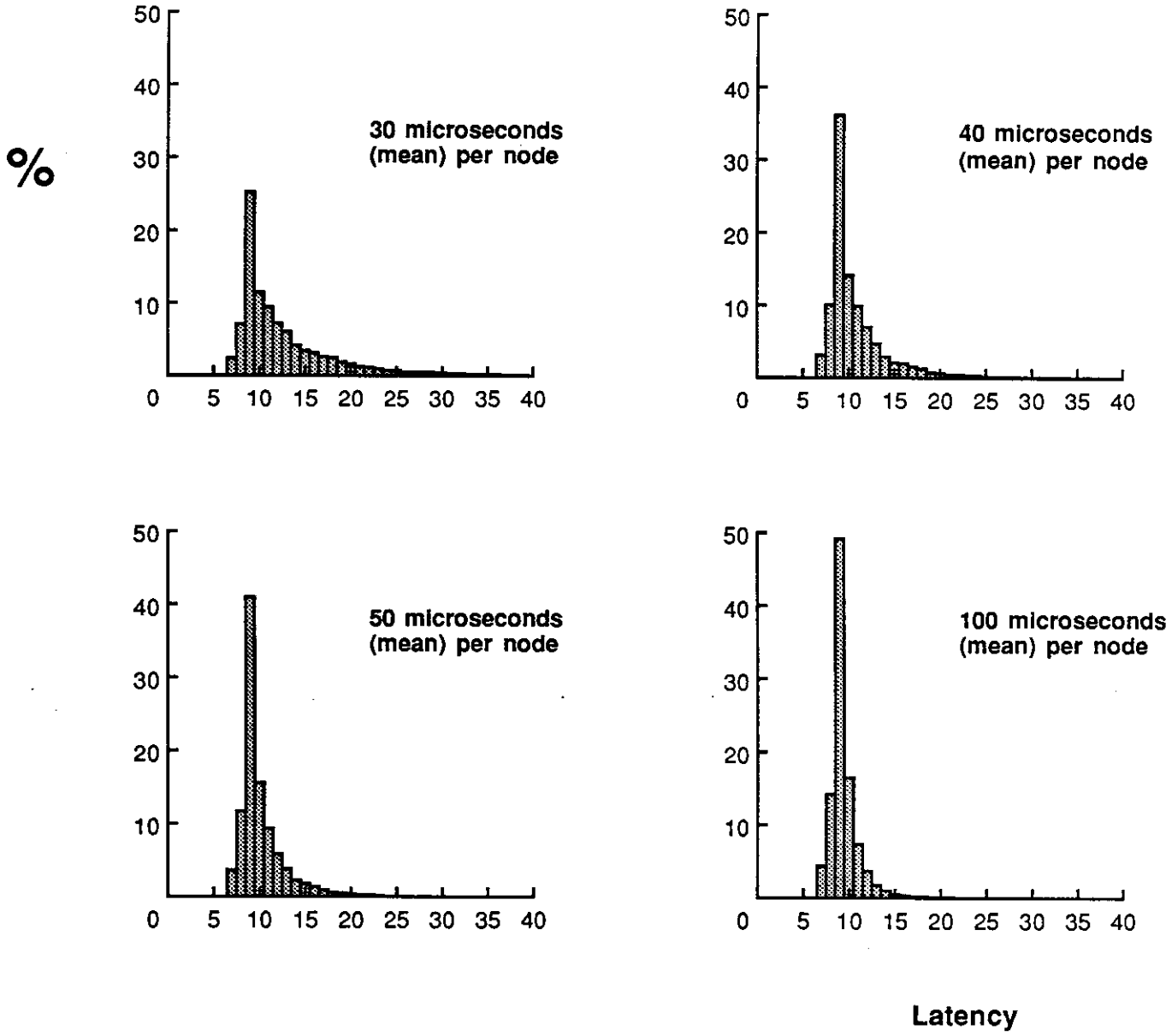

Figure 2. These histograms represent the **latency for a test-and-set** operation. Each graph shows percentage of messages versus latency (to the nearest microsecond).

The four graphs differ in the **arrival rate of requests**. The mean interarrival time, per node, was set at (respectively) 30, 40, 50, and 100 microseconds. In all cases, the minimum latency is 7.5 microseconds: the mean is (respectively) 13.9, 11.4, 10.7, and 9.9 microseconds: the mode is 9 in each case.

The simulation parameters are as in Figure 1, except that the cube order has been fixed at 7 (that is, 128 nodes). The operation sends 20 bytes down the built circuit, and receives 20 bytes back down the same circuit, before the circuit is torn down. It is assumed that the destination has a turnaround time of four microseconds in which to do the test-and-set on its local memory.