Simulating Network Traffic In An Associative Processing Environment

Claude S Noshpitz

UCSC-CRL-93-50
7 December 1993

Baskin Center for
Computer Engineering & Information Sciences
University of California, Santa Cruz
Santa Cruz, CA 95064 USA

ABSTRACT

This thesis considers issues related to the design of a distributed computing system optimized for research in the area of artificial intelligence applications. We introduce the Associative Processing Environment, a computing model explicitly designed to provide support at the machine level for systems that learn. Issues relating to the implementation of a multicomputer based on this model are discussed with a focus on the development of an effective processor interconnection network. We describe APES, a software tool for the simulation of message traffic in such a system, and present a series of experiments testing the behavior of a parallelized AI application on several topologies. We conclude that the behavior of the application is such that an assumption of uniformly distributed random traffic fails to capture essential aspects of the program's communication activity.

Keywords: interprocessor communications, computer architecture, network traffic, artificial intelligence, simulation

Contents

List of Tables

List of Figures

1. Introduction

Necessity is an evil; but there is no necessity for continuing to live subject to necessity.

- Epicurius

1.1 Overview

The computational requirements of large-scale Artificial Intelligence (AI) applications differ considerably from those of conventional non-symbolic applications. To date, most parallel computer systems have been designed with numeric applications in mind; the greatest optimizations have been made in the area of numeric performance and regular interprocessor communications. AI applications, however, are typically not numerically intensive and may have highly irregular and variable communication patterns.

An architecture optimized for the support of AI applications should provide efficient pattern matching and set-oriented operations and allow good performance in the face of irregular patterns of access. The **Associative Processing Environment** (APE), a closely-coupled distributed computing environment based on the system described in [Hughey and Roberts, 1993] is presented as a prototype for such an architecture. The APE provides support at the instruction-set level for the associative operations that are the critical 'inner loop' for many AI applications.

The behavior of interprocessor communications (IPC) in such an environment is expected to vary considerably for different application domains, particularly in the case of systems intended to 'learn' over time and with experience. IPC behavior within a single application may be highly nonuniform due to levels of structure within the data being manipulated by the system.

A standard design technique used in developing multicomputer interconnection networks (ICNs) is to assume a uniform traffic model in which processors generate messages at random times with uniformly distributed destinations. We suggest that in the case of certain classes of AI applications, excessive reliance on the uniform traffic assumption may lead to inappropriate design decisions.

The Associative Processing Environment Simulator (APES) is presented as a facility for the simulation of arbitrary interconnection topologies and message traffic distributions. APES supports the use of communication profiles generated by trace analysis of an existing program, allowing the behavior of parallelized serial applications to be evaluated.

A series of experiments are described that use APES to explore the IPC behavior of a parallelized AI application. The results indicate that the AI application engenders an IPC traffic pattern that contains behaviors not modeled effectively by the assumption of uniform traffic. We speculate that spatiotemporal locality in the AI program's data access patterns leads to transitory hotspots which can dramatically affect message latency and thus system performance.

We conclude that patterns of activity based on semantic relatedness among objects in the associative database give rise to communication patterns that are not modeled with complete effectiveness using a uniform message distribution. Hotspots are likely to be unavoidable, and designing ICNs in the hope that the hotspots will not have a great impact

on latency is not likely to be a good idea. Some adjunct to uniform analysis is appropriate; we suggest that simulations along the lines of those presented here are a step in the direction of a more complete design methodology.

1.2 Organization

Section 2 presents a brief overview of some important classes of AI applications and a philosophical[1] overview of some high-level issues in AI research. The question of what a platform for the support of experimental AI should be is discussed in terms of allowing researchers to concentrate on exploring *algorithms* by providing efficient support for AI primitives (i.e. as abstract data types) rather than forcing extensive *implementation* effort by shoehorning applications into inappropriate platforms.

Section 3 provides a general introduction to the APE architecture at the processor and programming model levels. Emphasis is placed on the pattern matching and associative primitives provided by the massively parallel content-addressable memory array in each processor; specific details about processor internals and instruction set are omitted in the interest of generality. Issues involved in the design and topology of the interprocessor communications network are described.

Section 4 describes in detail the requirements, design, and operation of the APE simulator. It is presented as both an analytical tool for examining communication patterns of existing programs and a design tool for experimenting with profiles for new applications. A key feature of the simulator is the ability to run the same simulation using different interconnection schemes. The process of mapping instruction traces onto the simulated architecture for analytical simulation is described.

Section 5 describes a series of APES experiments run on a number of different potential interconnection networks (ICNs) using trace data derived from the Morph [Levinson *et al.*, 1992] application. Communication patterns derived from the program traces were run on each ICN and the results compared to those derived from similar experiments using a uniformly distributed random traffic model. The results were examined with respect to assessing the validity of the uniform traffic assumption and characterizing the variations in the application's communication behavior due to topology.

In section 6 we explore several areas for further study including enhancements to the simulator, the simulation of other applications, and the search for emergent behaviors in the network traffic patterns.

Section 7 concludes the body of the thesis. The experimental results are summarized, and we suggest that these results are likely to be of relevance when designing an APE-based system for the efficient execution of associative-operation-intensive AI applications similar to Morph. Some areas for further work and exploration are also presented.

Appendix A describes the Morph application in detail and describes the process that was used to develop the execution traces used in the APES experiments.

[1] As in Webster's [Webster, 1986] definition of philosophy as "a search for truth through logical reasoning rather than factual observation".

2. AI research: trends and tools

2.1 Why an AI architecture?

Artificial intelligence (AI) applications have computational requirements quite different from those of typical numerical or simulation applications. Historically, parallel supercomputers have been designed and optimized to run codes that are computation-intensive such as image analysis, weather prediction, and numerical analysis [Almasi and Gottlieb, 1989]. The patterns of interprocessor communication in these applications are usually well-defined and regular. By contrast, AI codes, especially those implementing a *connectionist* paradigm, tend to do little number-crunching and a large amount of irregular communication [Fahlman, 1979].

AI applications have been developed on all sorts of machines, and a number of specialized architectures exist [Wah and Li, 1989, Feigenbaum and McCorduck, 1983]. At present, however, there are few 'general-purpose' AI machines widely available. The Lisp machines marketed by Texas Instruments and Symbolics come to mind, as do massively parallel systems such as Thinking Machines' CM-2 [Thi, 1990] and ICL's Distributed Array Processor [Parkinson and Litt, 1990]. They support limited programming models, and have demonstrated mixed results in some very specific AI areas such as neural and semantic networks [Moldovan *et al.*, 1990, Singer, 1990]. An ideal machine would support exploratory AI programming at an instruction-set level while providing the flexibility of a 'conventional' programming environment.

2.2 AI research tools — a (slanted) perspective

Lisp — ready for the 21st century?

AI researchers tend to suffer particularly from the universal problem of "When all you have is a hammer, everything looks like a nail." The majority of the program code that defines the field of AI has been written in Lisp [McCarthy and others, 1962] or its variants; a number of Lisp machines have been available to support Lisp primitives at the instruction-set level. A standard Lisp program, however, regardless of its elegance in exploiting the list-processing programming model, suffers from the relatively high overhead and low degree of parallelizability of Lisp [Deering, 1984]. Many applications have thus been coded in Lisp, recoded, and recoded again in efforts to extract the best possible performance.

Parallel Lisps do not seem to have had a dramatic impact on the structure or performance of AI programs. The data-parallel *Lisp [Hillis, 1985] encumbers the programmer with language limitations and its performance is dominated by the mediocre throughput of the CM-2's global router network [Cook and L. B. Holder, 1990]. Other approaches, such as Multilisp [Jr., 1987], extend the language with awkward syntactic structures supporting only a coarse-grained parallelism. Neither style of approach seems to have significantly affected the design of Lisp programs themselves. While clearly an elegant syntax for the description of many algorithms, the Lisp programming model appears to be fundamentally serial in nature[1]. While some primitives, such as `mapcar`, may lend themselves well to implicit parallelism, it would appear that a truly parallel Lisp will require explicit language extensions and/or extremely clever compilers.

[1] The primitive object in Lisp is a linked list of cons-cells, not an obviously parallelizable structure.

Programs and paradigms

Perhaps the most popular AI applications today are *production systems* or so-called 'expert systems.' These have their basis in work begun by Newell and Simon in the late 1950's (described in [Newell and Simon, 1972]) and have evolved into a fairly useful (but far from general-purpose) computing paradigm. Current production systems are in use in applications as diverse as computer system configuration (the R1 system at DEC) and medical diagnosis (MYCIN and its descendants). In general, these systems are able to work faster and possibly better [Gupta and Forgy, 1989] than humans in *extremely limited domains*; they are quite incapable of generalizing to new situations [Shapiro, 1990]. Efforts have been made to parallelize the systems, and fair speedups have been obtained on specialized[2] hardware [Gupta, 1984].

Prolog [Clocksin and Mellish, 1981] emerged in the early 1980's as a candidate for an easy-to-understand, easy-to-parallelize language for AI applications. Further research in the area of parallel logic-based languages has been able to demonstrate good speedup in certain applications, such as the PARTHENON theorem prover reported in [Bose *et al.*, 1992]. Some interesting parallel Prolog implementations have appeared, such as the one presented in [Ambriola *et al.*, 1990]. The applicability of purely logic-based approaches to general AI research, however, remains to be seen [Shapiro, 1990].

The notion of 'semantic memory' was introduced in 1968 with the publication of Quillian's dissertation [Quillian, 1968]. This work was in response to the problem of modeling semantic organization in the mind, something of an obsession with early AI researchers. What Quillian observed was that memory is not so much a matter of storing data and retrieving them as it is one of traversing the many relations among stored data. The idea of treating memory as a process of active association involving numerous potentially simultaneous operations on many data represented an evolutionary change. Previous (and many subsequent) applications relied on comparing tags in the data themselves to find similarities and differences, but in general the focus was on the data themselves rather than on the relations among them.

Some relatively simple problems were run with modest results, but various shortcomings of the model caused it to fall into relative obscurity for several years. A decade later, Fahlman presented the NETL system [Fahlman, 1979], extending the Quillian model and generalizing it into more concrete terms. NETL effectively described an architectural abstraction of semantic memory that could be evaluated in concrete terms. The model of data transformation as the propagation of markers (effectively single bit flags) among a network of nodes representing some semantic attribute or feature of the problem domain opened up the field of *semantic networks*.

Semantic networks were conceived as a realizable hardware model that could actually be built with existing technology rather than just as an abstract mental experiment. Fahlman believed that at least a million processing elements would be necessary before his machine could produce any 'interesting' results [Fahlman, 1980]. This was well beyond the limits of technology, at least with respect to funding at the university research level. Other work in semantic net architectures [Moldovan *et al.*, 1990, Hendler, 1988] has continued, popularized in the book *Parallel Distributed Processing* [McClelland *et al.*, 1986].

[2] The RETE algorithm [Forgy, 1982], physically embodied in the DADO and NON-VON machines reported in [Stolfo, 1984] and [Shaw, 1985], provided speedup based on the significant degree of parallelism implicit in the serial algorithm itself.

Fueled by a resurgence of interest, connectionism emerged as the next direction for AI research. Both cognitive modeling and symbolic processing approaches, as well as various hybrids, have come to fall under the connectionist rubric. One branch of connectionism has taken up the cause of so-called *neural networks*, creating mathematical models of neuron-like entities in an attempt to re-create their functionality. This approach has shown reasonable success in some specific domains such as the modeling of certain areas of precognitive function including early vision processing [Ballard *et al.*, 1983]. These areas, however, represent a highly specialized functions; it appears unlikely that any sort of *generalized* intelligence will be achieved using current neural network-style approaches.

In [Hoffmann, 1990], Hoffmann uses the methods of algorithmic information theory to show that the application of neural networks to learning problems does not (and cannot) change the amount of *work* that must be done to learn a given domain. Whether the work is in the form of repeated training iterations or in the clever design of the system's architecture, the same information-theoretic lower bound exists on the amount of energy that must be expended to achieve a given level of performance.

We suggest that this structuralist approach, while valid in information-theoretic terms, fails to capture the effect of representation and chunking on the learning problem; consider a shape recognition task whose primitives are pixels with one that has as its underlying representation known objects such as lines and circles. The former must do all the work of learning about what a 'circle' is before it can begin to categorize objects into the 'circle' class. The latter, however, benefiting from previous knowledge encoded into its representation, can use its time learning higher-level relations. While the theoretic limitation is still true, in fact the chunking of knowledge at the representation level radically alters the meaning of the limitation.

Another branch of connectionism has attempted to merge the traditional 'symbolist' version of AI (canonified in Simon's Physical Symbol System hypothesis[3]) with the spreading activation model spawned by Fahlman's work. Minsky has proposed a sort of connectionist schema of intelligent systems in *The Society of Mind* [Minsky, 1986]. His assertion is that 'intelligence' is the result of many relatively independent agents acting in concert, each an expert in some particular domain and none necessarily in charge. This approach can be visualized as a cooperative heterarchy; a large set of 'purposeful' entities, each with its own area of specialization, are interconnected so as to engage in a cooperative problem-solving attack. The sum total of their efforts is greater than the abilities of any individual agent, yet all agents contribute to the result. This model may be the closest yet to abstracting relatively generalized cognitive functions into a computable methodology.

Another approach to making state-space search more efficient is Adaptive-Predictive Search (APS) [Levinson *et al.*, 1992]. APS concerns itself with accumulating a database of graphlike patterns based on combinations of predicates over the state space in an effort to capture domain knowledge. Each pattern in the database has a weight value associated with it that indicates the expected degree of that pattern's contribution to the acquisition of a goal state. The patterns are accessed by an associative search mechanism that attempts to relate patterns by their similarity according to a subsumption relation such as *more-general-than*. The intention is for patterns similar to one another, and therefore implicated

[3] The hypothesis states that physical symbol systems, i.e. representations of physical systems, possess "... necessary and sufficient means for general intelligent action". If we allow the possibility that any physical system can, in theory, be represented by a symbol system of arbitrary complexity, we have the basis for a belief in 'strong' AI.

in related chunks of knowledge, to be easily accessible to one another. Generalization on the feature space is an implicit part of this model, permitting the potentially fruitful reuse of patterns in subsequent searches over the same feature space.

The associative database in an APS system represents yet another style of connectionist approach; the collection of patterns is in effect a network of interacting objects which represents the 'knowledge' that the system has accumulated. APS is notable in its explicit use of associative primitives to access objects in the database, the lack of implicit structure in its knowledge base, and its espousal of experience-based learning in the form of the interaction of past knowledge with the state space.

We suggest that connectionist AI is an area currently undergoing a tremendous amount of exploration, and that its unique computational requirements are not well served by current architectures.

2.3 Some comments on the nature of AI research

The following are several ideas about working on AI that emerge from this author's survey of AI research. They are presented in an effort to convey some of the flavor of the exploratory and rather open-ended nature of the field.

Solutions to toy problems do not necessarily lead to real solutions. Building a blocks world and manipulating it, as Winograd's SHRDLU [Winograd, 1972] did, is simply no more than a blocks world. Developing a neural net that solves a small learning problem says little about how efficient a bigger version will be [McClelland *et al.*, 1986]. Morph [Levinson, 1991], a program able to learn the rudiments of chess, is unable to transfer its knowledge to checkers. In each case, generality is lost to a combination of tailoring the solution to the problem at hand rather than attacking the greater class of such problems, and of combinatorial explosion as domains grow larger.

Working on real problems demands real resources. Almost all AI applications demand lots of memory and lots of computing cycles for pattern-matching and set-associative operations. Much of AI amounts to search in a very large state space [Rich and Knight, 1991]. As clever as the search algorithms are, the bigger the state space the more resources are needed. It is sometimes possible, however, to leverage resources by exploiting parallelism both at the data level, as in the text-retrieval applications on the Connection Machine [Stanfill and Kahle, 1986] which have shown very good performance on complex searches, and at the algorithm level as in the parallel Rete match [Forgy, 1982] which does a static dataflow analysis and partitions the problem among available processors for a respectable speedup.

Much implementation effort has gone into shoehorning inherently parallel algorithms into Von Neumann-style platforms. A case in point is the Morph program; a tremendous amount of its development effort has been aimed at tuning the code to achieve decent performance [Gould and Levinson, 1991]. In effect, the underlying algorithms have been tweaked until they run reasonably on the available hardware. The refinements to the algorithms are thus driven less by abstract issues of theory than by hardware constraints.

Whether more appropriate hardware will permit the underlying theory to be better explored, or if such explorations will bear fruit, remains to be seen. After all, necessity is the mother of invention...

Some AI problems *require* an exploratory programming style. Marr distinguishes [Marr, 1990] between what he calls *type 1* and *type 2* theories in relation to AI problems. Type 1 theories are those that provide a 'clean', decomposable (read *rational*) solution to a particular problem. A computational theory of type 1 not only solves a specific problem, but does so by exposing some *fundamental* underlying mechanism. That the mechanism is fundamental implies that an algorithm built around it will be solid and correct, if not necessarily easy to implement. Numerous problems in the area of vision, for example, have lent themselves to type 1 solutions.

Type 2 theories, on the other hand, are inherently non-fundamental. They are composed of many subprocesses, and no single underlying theme unites them; the theory is described solely by the interaction of its components. Type 2 theories, in Marr's formulation, have dominated the AI field. Presumably these are the 'computational theories' that are communicated solely via program listings, since the implementation *is* the theory. Marr makes the point that there is not necessarily any easy way to determine whether a type 1 theory exists for a particular problem, and that the surfeit of type 2 solutions can hide an underlying type 1 theory.

If a particular problem has no type 1 solution (and there seems no reason to believe that all high-level cognitive reasoning problems do), the only way to develop a suitable type 2 solution is by exploratory programming, trying different approaches until a good one is found. By the same token, if a type 1 theory *does* exist, it may well be found after enough type 2 approaches have been tried.

This author believes that a significant cause underlying the preponderance of type 2 solutions in AI research is that the programming tools and environments that have been available fail to offer sufficient resources to explore the space of possible solutions adequately enough to discover underlying type 1 solutions. Once a type 2 solution is found, it lingers until another comes along. The process of exploration is not as well-exercised as it might be. Given good throughput and a programming model that supports search and matching, it seems reasonable to assume that more type 2 solutions will be tried, presumably leading toward more fundamental solutions.

2.4 Representation, search, and pattern associativity

AI and state-space search

Many different application domains have been characterized by the term 'Artificial Intelligence;' these run the gamut from playing simple games to understanding human languages and diagnosing illnesses. While a tremendous amount of effort and creativity has been devoted to developing different AI programs, the principles underlying almost all of them are similar: the classic AI problem is a search for some (nonempty) set of goal states over the state space consisting of all the possible configurations of the system under consideration. This implies several fundamental entities — a *representation*, which formalizes the state space, a *goal state* or *goal predicate* which defines the termination condition of the search, and a set of *operators* that transform states. Additionally, an

omniscient entity or *trainer* can provide feedback on the current distance to the goal state(s) as the search progresses.

The problem of searching an exponentially large state space has been attacked on many fronts. If *no* feedback (in the form of a trainer) is available from the environment until the goal predicate is satisfied, then little can be done besides brute-force search. Numerous *informed* search methods exist, however, for exploiting the information provided by some environmental feedback. Well-explored techniques such as A* [Hart *et al.*, 1968, Hart *et al.*, 1972] are proven to converge and are able to find optimal solutions.

One weakness of conventional informed search methods is that they tend to be relatively brittle; they all rely on some generally fixed evaluation function that represents an estimate of the current state's distance from the goal. A small discrepancy in the evaluation of a particular state can lead the algorithms far down unfruitful paths before finding their way back to the right one. Evaluation functions tend to be used to abstract as much of the representation as possible into a goodness value as quickly as possible. Since the evaluation function must be called for each state visited during the search, its computational cost must be minimized. This implies that it probably cannot capture much of the structure available in the representation, even though it is precisely this structure that can best inform the search!

Adaptive/Predictive Search and the Pattern-Weight formulation

An alternative to the model of explicit state-space search is the *Pattern-Weight* (PW) representation used in the Adaptive-Predictive Search (APS) approach described in section 2.2. APS attempts to abstract relevant features of the state space based on experience. This would provide a potentially more efficient search and allow for a better response in a reactive environment (i.e. a situation in which the goal predicates may change).

A PW consists of a graphlike pattern, which is a predicate over features of the state space, and an associated weight that indicates the expected degree of that pattern's contribution to the acquisition of a goal state, or the system's degree of 'belief' in the pattern. Weights are updated periodically using a form of temporal-difference learning [Sutton, 1987], in which periodic feedback from the environment causes reevaluation of the sequence of patterns that led up to the present state.

A pattern is effectively a set of interacting *features* which together represent some partitioning of the state space. Since patterns generally represent *partial* states, they in effect *generalize* the state space, providing a higher level of granularity of knowledge (or 'chunking') than that available to conventional search methods. The PW formulation attempts to extract whatever structure information is available in the given representation and to exploit it by creating patterns that represent 'interesting' sets of features (i.e. those that lead toward optimal satisfaction of the goal predicates).

APS and the PW formalism provides a compact and efficient alternative to conventional state-space search; in effect the interaction of PWs and the evaluation function result in an abbreviated heuristic search. APS attempts to exploit whatever structure is available in the underlying state-space representation.

Several applications implementing APS have been implemented [Levinson *et al.*, 1992, Levinson, 1993]. These systems all comprise the key elements of APS:

- Pattern representation: a methodology for the mapping of domain features to PWs; some style of semantic network is applicable. Conceptual graphs (CGs) [Sowa, 1992] are a good candidate for a pattern representation formalism. They are compact and semantically rich, and very efficient algorithms exist for their manipulation. Because they are based on semantic rather than syntactic rules, CGs can expose underlying structure in the feature space.
- Associative database (ADB) and pattern retrieval method: a scheme to organize and manipulate patterns according to a subsumption relation such as *more-general-than* (as described in [Ellis, 1992]). Patterns are to be classified into a partial order according to their relationships with other patterns in the database.
- Search method: strategy to select those patterns from the ADB most likely to lead to the desired result. Implemented as a hill-climbing energy-minimization search, patterns are selected using an evaluation function to provide a measure of the degree of 'goodness' or applicability of a particular pattern.
- Learning algorithm: a method for the modification of patterns' weights based on the result of their application. The experience-based learning (EBL) method described in [Gould and Levinson, 1991] suggests the use of a simple form of temporal-difference (TD) learning [Sutton, 1987], which assigns positive or negative credit to patterns contributing to a particular experimental outcome.
- Pattern generation method: a facility that creates new patterns both by integrating new observations into the ADB and by recombining existing patterns into more powerful or general configurations.

Its reliance on an associative database means that the performance of an APS system depends heavily on implementation. Because the ADB represents only a partial order on feature space, it cannot be efficiently implemented by conventional key-index methodologies. The graph matching operations at the core of the required subsumption operator are typically NP-complete and thus are typically not amenable to brute-force methods.

We suggest that providing instruction-level primitives to support the associative operations required to maintain an ADB is an important step toward the design of a platform that encourages experimentation in the APS domain. This is assuming, of course, that AI researchers would prefer to build systems that learn rather than wander the NP space of graph-matching algorithms.

3. Designing an associative processing environment

3.1 An overview of associative processing

We can distinguish *associative* from 'conventional' processing by defining associative processing as "the manipulation of data based on their content and equivalence classes[1]" [Hughey and Roberts, 1993]. This is in sharp contrast to the conventional computational model originally described by von Neumann [Burks *et al.*, 1947], whose explicit intention was the movement of individual data items among specific locations using unique addresses. Given these definitions, we extend the conventional notions of data manipulation to include the accessing of data *by content* rather than by location. Any data movement operation may involve set membership or other pattern matching operations, and any 'simple' operation may affect the global state of a machine rather than a specific location.

There is a fundamental assumption that can be made regarding the primitive operations that are required for the support of such an approach: **pattern matching and set operations must be handled as efficiently as possible and preferably at the instruction-set level**.

In the case of a system distributed over multiple processing elements and address spaces, it is necessary to *assume* that data accesses are at least occasionally, and in the normal case perhaps often, nonlocal. Nonlocality of access implies that traffic in the interprocessor connection network may be very bursty and may have intense transitory hotspots [Kumar and Pfister, 1986]. In addition, referential nonlocality is a natural consequence of the generally irregular and nondeterministic behavior of large AI programs. Design choices based on this model of data access must be made as early as possible.

Massive parallelism is appropriate to the needs of connectionist system; multiprocessing is a natural consequence of their structure, and the mapping to many-processor systems [Fahlman, 1979] is easy. In the case of marker propagation systems in particular, very large amounts of interprocessor traffic occur during processing [Miranker and Andrews, 1990]; it is worth noting that the size of marker messages is typically very small. Connections among the processors must be as fast and general as possible, and the design process must take these factors into consideration.

The Connection Machine CM-2 [Hillis, 1985] is representative of the one style of attack on the problem. Using a fine-grained single-instruction, multiple-data (SIMD) approach to parallelism, it was originally intended to be very good at semantic network applications [Hillis, 1981]. In actual fact, the production model turned out to be quite inefficient in that domain due to its rather weak performance in global routing[2]. Well optimized local communication yields good performance on regularly partitioned applications. Irregular problems, however, such as neural network simulation, have yielded relatively poor results [Singer, 1990].

---

[1] The equivalence classes in the case of an AI application would represent partitions on the feature space.

[2] This author's experience implementing Sparse Distributed Memory on the CM-2 [Noshpitz, 1991] confirms that significant effort must be expended minimizing global communication in order to achieve reasonable performance.

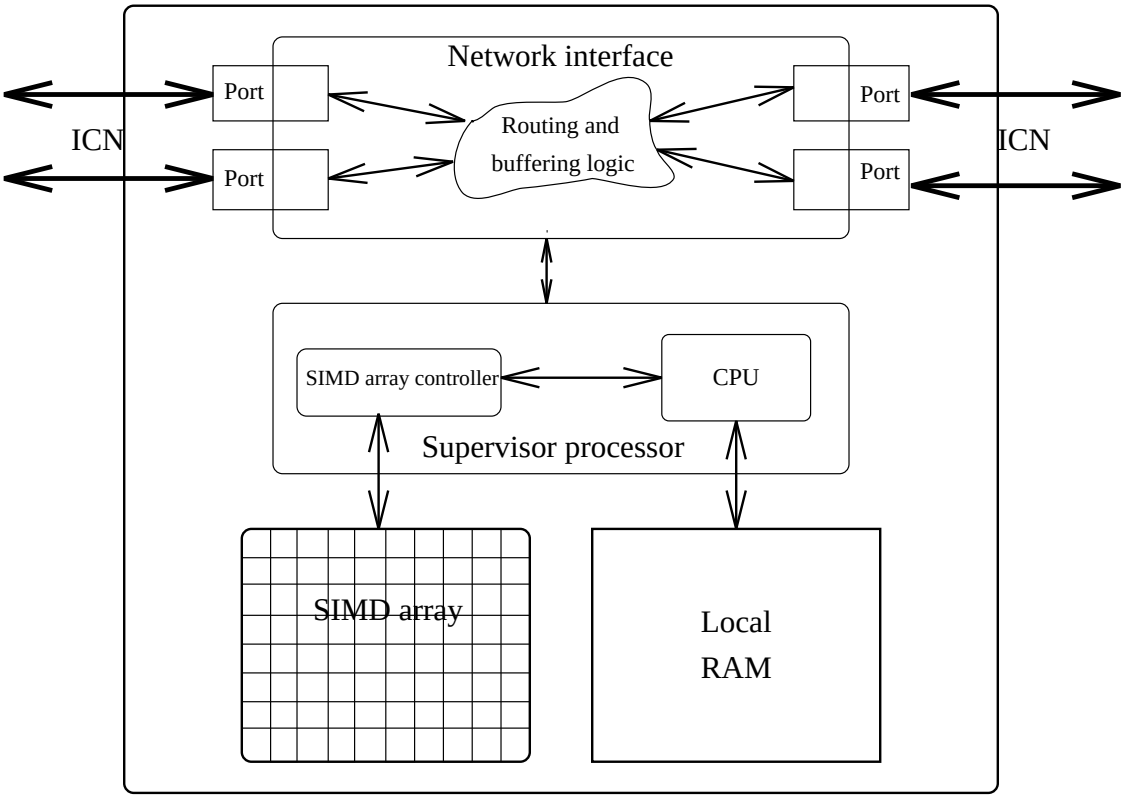

Figure 3.1: Block diagram of an associative computing node.

3.2 An associative processing architecture

The next level of abstraction for parallel computing in the SIMD style is to use the SIMD arrays as coprocessors, providing a high-powered supervising processor to control the arrays and do serial operations. We propose such an arrangement in the Associative Processing Environment (APE), a closely-coupled multicomputing environment based on message-passing among a large network of powerful computing elements each equipped with a SIMD processor array. Such a system would have the ability to execute associative operations as primitives in a data-parallel programming environment. The APE abstraction is based in large part on Roberts' MISC machine [Hughey and Roberts, 1993], which utilizes these ideas in a multicomputer with support for associative-match primitives.

An APE computing node consists of a conventional control processor, or *supervisor*, with a generous amount of local RAM, tightly coupled to a SIMD array of small processing elements. The array serves as a content-addressable memory (CAM) able to efficiently perform pattern-matching and set operations. The general structure of a processing node is illustrated in figure 3.1. A fairly dense machine could be put into a small cabinet, minimizing interprocessor wire lengths and allowing more resources to be devoted to 'fattening' the interconnects to speed communications.

A full APE system would consist of some number of computing nodes joined by a fast interconnection network. The network should provide efficient global routing and be as resilient as possible in the face of hotspots and faults. It is probable that a dynamic load-balancing scheme would be implemented to keep hotspot contention as low as possible; one

area in which the APE simulator is expected to be used is in evaluating the cost/benefit tradeoffs of load balancing. Marker-propagation messages tend to be short and frequent, and the migration of data structures at the process level generally involves periodic bulk transfers [Noshpitz, 1990]. A design goal, then, is for both short and large message traffic to be handled efficiently.

The Control Processor

The impact of embodying the associative processing paradigm at the processor level is considerable. Primitive operations may have high latency since associative match could take several orders of magnitude longer than local memory references. Some mechanism beyond simple pipelining is needed to mask this latency, which may be highly nonuniform. In Roberts' description of the MISC machine, the processors are similar to hybrid dataflow [Ianucci, 1988] machines, switching among several execution threads to hide the latency involved in associative array or message-bound operations.

Associative match instructions may need to broadcast keys to other processors and receive match data back in an iterative fashion, implying that interprocessor communication (IPC) can be expected to occur more often than in a conventional message-passing system. The latencies due to this communication should also be masked. Since associative primitives involving the local SIMD array are expected to occupy a significant part of the actual execution time, the split-phase transaction model described in the hybrid dataflow literature [Arvind and Nikhil, 1987] seems appropriate.

The Associative Array

Each control processor is bound to a SIMD array of simple processors based on a CAM cell such as the PCAM [Roberts, 1990]. Such an array is capable of performing exact match, k-nearest-neighbor and other associative operations (as described in [Kanerva, 1988]) efficiently. Each of the processing elements in the SIMD array contains a small local memory as well as a grid-based message routing mechanism.

The abstraction supported by the SIMD array is that of an efficient associative-memory module; data are written and read like a (relatively slow) conventional memory with additional support for fuzzy matching or associative operations. Such an array can provide support for various implementations of associative operations, such as Kanerva's Sparse Distributed Memory [Kanerva, 1988] and the K, d-tree and similar structures described in [Omohundro, 1990].

Associative array operations are supported at the instruction-set level of the supervisor processor. The supervisor contains logic to broadcast microinstructions and data to the processors in the SIMD array and to collect the results of their computations, so that array operations are invisible outside of the processing module. The latency of the SIMD array is generally proportional to its size; a simple pattern-matching operation should take the same amount of time regardless of the number of hits. Best-match searches, however, which may recall some number of elements in sequence, can take a variable number of SIMD cycles. This is another point in favor of providing as many mechanisms as possible to hide latency in the supervisor.

The interconnection network

We believe that a closely coupled network of APE computing nodes requires a very efficient global interconnection network (ICN). An important design goal for the ICN is for the latency of a nonlocal access to be bounded and to be within a small order of magnitude of that of a local access. This requirement places serious demands on the network in the face of the irregularity of communication and hotspot behavior anticipated to arise in the target applications.

The choice of which topology to use is not altogether straightforward; it is possible for implementation and technology-related issues to overwhelm theoretical analysis in actual network performance. Fat-trees [Leiserson, 1985] and multiple bus [Dai *et al.*, 1991] systems appear to be good candidates, as do some members of the large class of k-ary n-cubes described in [Dally, 1990].

Many studies of network performance have been based on a model of uniform, randomly distributed message traffic [Patel, 1981, Bhuyan *et al.*, 1989]. This is in fact not a particularly likely scenario in the applications described as the targets for APE. In particular, data clustering and hotspots seem quite likely to occur in the context of a semantic net and particularly so in conjunction with dynamic load balancing.

While conventional numeric and simulation applications seem to tend towards uniform, random traffic, AI applications may not. The law of large numbers implies that a large group of processes interacting *randomly* would tend to evince a normal distribution. AI applications, however, are expected to represent features of a *structured* domain. Some abstraction of this structure is likely to be reflected in the patterns of communication among objects in a distributed AI system working in a 'real-world' domain.

3.3 Why simulate an APE?

Based on this discussion, it would appear that the best that could be done in developing the APE network topology might be an educated guess, trying for the most 'general' approach that seems implementable. In fact, given a simulation tool that is efficient and easy to use, the chances of the guess being a good one could improve dramatically.

By doing dataflow analysis on traces from an actual AI application such as the Morph project [Levinson, 1991] and mapping it onto the simulated APE system, we have characterized some of the essential qualities of expected communication behaviors and their sensitivity to topological variations.

Like most existing AI applications, Morph has been heavily optimized to run on single-processor Von-Neumann-style machines. Although its implementation is deeply rooted in a conventional workstation environment, Morph makes heavy use of the associative primitives that are expected to be at the core of an APE system.

Ultimately, it will be necessary to model application codes that do not yet exist; using assumptions based on known codes it should be possible to generate statistical models of message passing and access patterns with enough validity to exercise the simulator in ways which will prove useful to the system's design.

3.4 Design issues for distributed associative processing

We assume that at the heart of an application running in an APE is an associative database (ADB) distributed across the processing nodes. We expect that interprocessor communication (IPC) patterns will change as relations among objects in the ADB change. This represents a key difference between the assumptions underlying most 'scientific' applications and the connectionist style of AI. Many scientific problems are described as a regular tesselation over some computational space; communication requirements are thus predictable and regular, or at least deterministic [Almasi and Gottlieb, 1989].

The temporal and spatial structure of search in an ADB is not necessarily regular; in a system implementing adaptive-predictive search (APS) no *a priori* structure at all is imposed on the stored patterns or their relations. Objects in the ADB represent an abstraction of the application domain's feature space, and their structure is derived from the system's observations about the feature space rather than from explicit rules.

For some perhaps contrived problems this structure could be quite regular. In the case of more interesting problems it may be better to assume that the structure of the knowledge base as experience builds will resemble a chaotic system; the world is a rather chaotic place, with lots of low-level structure and order interacting at higher and higher levels of abstraction to generate more and more complex behaviors. It is expected that the structure of the database underlying a mechanical representation of a worldly system will not be regular, and probably will change (perhaps dramatically) as the system's experience evolves.

Depending on the complexity of the domain, it may be very difficult to determine ahead of time how IPC requirements will change. This emerges as a serious design issue when implementing a parallelized ADB because low latency is an important adjunct to the ability to maintain very large, and therefore interesting, databases. Since search time must scale with database size, ADB operations represent a potential performance bottleneck for large APS systems.

Locality of access and load balancing

We suggest that there is likely to be a good deal of locality among accesses to a distributed ADB. This locality takes the form of clustering patterns in parent-child relationships within the database hierarchy. Objects that are 'similar' in feature space are expected to be near one another in the ADB, since the ADB's classification scheme is defined as being related to distance in feature space. Therefore, access to 'similar' objects is likely to involve access to many of the same intermediate objects during search.

In order to maximize processor utilization and minimize IPC, we expect that some form of dynamic or demand-driven load balancing will need to be implemented. The general thrust of the load-balancing methodology is that those objects that are most often in communication (i.e. compared to one another) should be in closest proximity, either coresident in an APE node or a minimal number of network hops away. The criteria used to determine when to migrate or replicate an object are left to further research.

3.5 Chaotic dynamics in a large connectionist system

One key characteristic of many current AI codes, and presumably of any system attempting to behave in an 'intelligent' fashion, is non-determinism. Given that the world is

generally not a regular, orthogonal system, any system that maps the world internally to a significant degree must also map its irregularities. It is anticipated that once a computer system builds a complex enough world model, that model may begin to behave as a chaotic system [Gleick, 1988] rather than a strictly deterministic one.

In effect the connectionist paradigm, which holds that knowledge in a system is a holistic product of some set of relations among its elements, can be viewed as a mapping of features (i.e. facts about the domain under consideration) into a very high-dimensional space whose coordinate axes are a cross-product of all the degrees of freedom in the system. A particular knowledge item (or 'memory') is in effect a hyperplane passing through particular points of interest in the knowledge space.

In addition to the points of interest, however, the hyperplane is likely to pass through many other points that may not be relevant in themselves but which nevertheless are part of that particular assemblage of experience in the system. Therefore, the behavior of the system when moving among specific memories (i.e. locations in knowledge space) may defy simple prediction and could be expected to express chaotic qualities.

Points in the knowledge space can be mapped more-or-less arbitrarily to physical locations in the computer system. It then seems reasonable to assume that the inclusion of apparently unrelated points as described above is likely to cause nonlocal connections to be made among apparently unrelated objects. Although not bearing any direct semantic relation, these items nevertheless share points along the hyperplane defining what is in effect a query into the knowledge database. Assuming that data are clustered in particular processors according to some arbitrary structural considerations, it is possible that the observed communication patterns will end up appearing to be chaotic as the state of the system evolves through time.

So how does the potential for chaotic communications behavior affect the design of a computer system capable of representing and manipulating such a system? We suggest that a high-throughput global communications facility is essential. The underlying structure of the access pattern is likely to lead to hotspots; hotspots, as described in [Kumar and Pfister, 1986], can cause a network to exhibit complex nonlinear behaviors which are very difficult to handle gracefully. It may therefore be difficult to predict accurately any but the grossest levels of IPC behavior. Qualitative simulations serve as an effective adjunct to the design, providing understanding of the behavior of the system as it evolves through time. Given simple stochastic assumptions about program behavior, a straightforward simulation should be able to capture at least some of the flavor of the dynamical behavior of the system. It is with this intention that the APE simulator is designed.

4. APES, a network traffic simulator

4.1 Overview

The Associative Processing Environment Simulator (APES) presents a readily configurable interface to a generic simulation of the network traffic in a multicomputer of the style described in the previous section. The simulator is designed to develop a qualitative model of *patterns of activity* in the system given arbitrary topologies, routing methods and policies, and code profiles. The generic routing elements that are simulated are an abstraction of features to be found in most multicomputer systems. Simulation of traffic on arbitrary topologies using a variety of routing and buffering policies is supported. Message traffic may be generated according to random distributions or be derived from code profiles of actual programs.

APES simulates the message traffic in an arbitrarily connected packet-switched network. Each node in the network is considered to be a processing element (PE) and may contribute to message traffic independently of other PEs. The arcs in the network correspond to communication links which are assumed to have a fixed capacity such that a link can carry a single message packet at a time (the *width*, or number of wires, in a link is thus equivalent to the number of bits in a message packet).

A variety of buffering policies and non-adaptive routing functions are available. Statistics about traffic along each communication link and in each processing element throughout the time of the simulation are gathered. These can be analyzed for clustering and hotspot behavior as well as processor load balance. Statistics are collected for traffic in and out of each processing element (PE) as well as across each link in the network. Message latency statistics are also collected.

The simulator can be used both to analyze execution traces generated by existing applications and to experiment with instruction profiles for simulated applications. Traffic and latency statistics can be used to explore clustering and hotspot behavior as well as processor load balance. By providing tools to examine behavior of existing applications, the simulator enables the design of a system to be tuned to known problem approaches. In addition, novel approaches involving the exploitation of the unique facilities available in the APE can be simulated in a qualitative fashion in order to develop those approaches and discover architectural issues that may not be obvious otherwise.

4.2 Architecture of the simulator

The simulator consists of several interacting modules, each of which is an abstraction of some part of the system under study (with simplifications to enable qualitative evaluations to be made without incurring prohibitively high simulation overhead). The modules are linked together into a single executable file. The executable takes as input a file describing the network topology and possibly another file containing message events to be generated during the simulation run.

Measurements taken include traffic flow through each communication link and message latency. The resulting data are available both as a realtime graphical display of selected activity over time (for example, a moving graph of traffic in each link) and as a summarized statistical report. System state at each step of the simulation can be logged to a file.

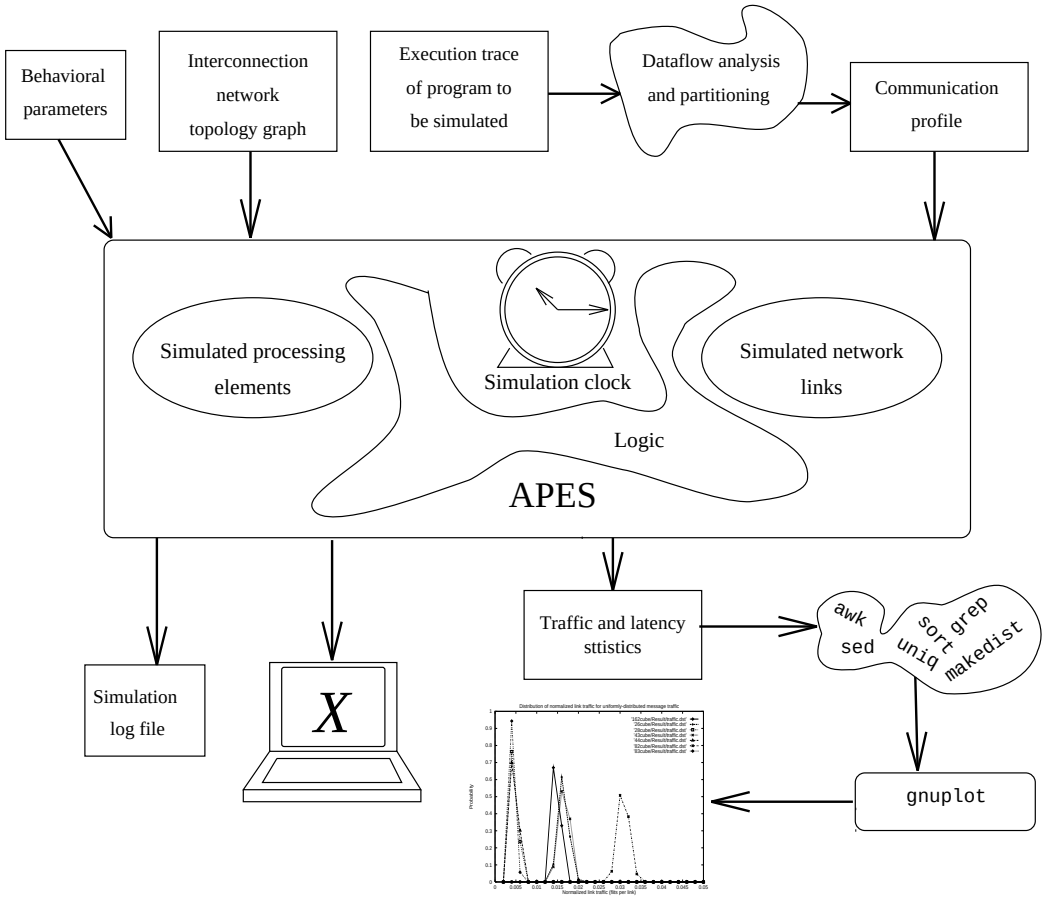

Figure 4.1: Block diagram of the APE simulator, its inputs, and outputs.

The overall structure of the simulator is illustrated in figure 4.1. The *topology graph* is a simple adjacency-list representation of the ICN to be simulated. The *execution trace* is produced by instrumenting the program whose behavior is to be simulated, and is analyzed to produce a *communication profile* for the program. *Behavioral parameters* include the choice of routing and buffering schemes, queue sizes, and operational details such as output format. The *simulation log* is a detailed trace of each step in the simulation. The *traffic and latency statistics* are described in section 4.7 and can be processed with a variety of analytical tools. The *console* can be used to view the progress of the simulation, choosing among several variables of interest such as link traffic and queue depth in each PE.

4.3 Features

Network topology

The interconnection network (ICN) to be simulated is expressed as an arbitrarily connected network of processing nodes and communication links. No explicit limitations are placed on connectivity, although the speed of simulation is polynomially related to the number of links in the network.

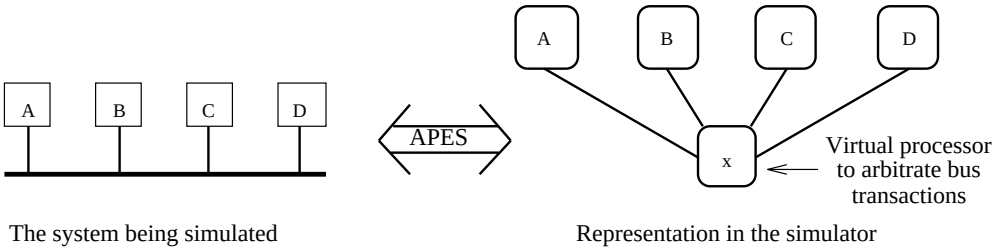

Figure 4.2: Representation of bus structures inside the simulator.

Buses and bus-like structures such as trees are supported explicitly. They are represented internally by virtual processors[1] which arbitrate among the links connecting the PEs on each bus (see figure 4.2). Hierarchical buses and other hybrid bus designs are also supported by this mechanism. For analytical purposes, the traffic on a bus is the sum of traffic in all the links connecting its constituent nodes to the virtual processor.

Message injection

Each node in the network has the potential to inject messages, or generate *message events*. A message event causes some number of packets to be injected into the network. These can be created stochastically according to a statistical distribution whose parameters are specified by the user or can be read directly from a user-supplied file. In the latter case, the file may originate from a parallelization of code traces from existing applications or from other modeling approaches.

Since injection of messages by a particular PE can be explicitly controlled, support exists for treelike topologies in which interior nodes act only as routers and all processing (and message injection) is done at the leaves. The virtual processors that support bus arbitration in an explicit bus simulation never inject messages of their own, but act only as forwarding agents.

Routing and buffering policies

There has historically been little distinction between buffering and routing in the architecture literature. Routing is a computational method for selecting the destination of a message packet, while buffering policy is concerned with the allocation of buffering resources within the router itself. APES provides mechanisms to implement several varieties of each.

A wide variety of approaches to routing and buffering are described in the literature; one partitioning is the distinction between adaptive and fixed routing techniques. Adaptive routing [Linder and Harden, 1991] may introduce some nondeterminism into the path a message will take with the benefit of providing better behavior in the face of network blocking. The cost, of course, is in computational complexity. Fixed routing schemes can be vulnerable to blocking (increasing latency) and deadlock, but are relatively efficient to implement. Another approach is the use of virtual channels [Dally, 1992] to split available bandwidth into independently buffered data streams at the cost of increasing the expense of buffers and routing logic; the cost is balanced by provably deadlock-free operation (again at the cost of some computation to route appropriately).

[1] These processors are not part of the explicit topology map provided by the user, and never inject messages. They act only to arbitrate among pending bus requests and pass messages along the bus.

The philosophy implemented in APES is to provide the simplest and most general degree of functionality possible. Therefore, only fixed routing methods are supported. These include dimension routing (described in [Hillis, 1985] and in a more general form in [Felperin *et al.*, 1991]) which routes in fixed order along each dimension of a cube-like structure[2], and an explicit routing based on the equivalent of a global routing table. The latter may require a computationally expensive search of the ICN to determine the best paths, but this is done only once at the start of simulation. For the future, a simple programmatic interface permits new routing schemes to be easily integrated.

The available buffering policies include wormhole [Seitz and others, 1985], virtual-cut-through [Kermani and Kleinrock, 1979], and store-and-forward [Tanenbaum, 1981]. These span a wide range of implementation possibilities. The observable areas in which they differ are in latency and blocking behaviors. Blocking behavior is a function of contention for network links. The question is whether a single blocked packet can rapidly propagate blocking through the network or if blocking behavior tends to be local. This depends in effect on the granularity of the buffering scheme; the fine-grained approach of wormhole buffering implies that blocking the head of a message could cause each router along the message's path to block immediately while the coarse granularity of store-and-forward buffering does not necessarily cause any routers but the current one to block.

Figure 4.3 illustrates the relative latencies of wormhole and store-and-forward buffering. It assumes that three routers are involved in the message's transit. Since only a single flit[3] is buffered at a time, the head flit in the case of wormhole buffering arrives at its destination long before its counterpart in the store-and-forward case. The latter must wait for all flits to reach the intermediate router before beginning to forward, engendering greater message latency.

The wormhole approach involves minimal buffering at each node; a packet gets passed along its path if the path is available, otherwise it blocks. This can cascade, blocking any packets waiting to travel along the same path behind the blocked one. On the other hand, a message can begin arriving at its destination as soon as its head flit has propagated through the network.

Store-and-forward buffering, by contrast, buffers all the packets in a message at each router along the path before propagating them. This causes the entire message to block at an intermediate router until the next link is available. The additional latency engendered by this approach can be offset by less catastrophic blocking behavior, since fewer routers are likely to become blocked by a single link being unavailable.

A third alternative, virtual-cut-through, is a hybrid of the two; it extends wormhole routing to allow several flits to be buffered at a blocked router. This mitigates the undesirable blocking behavior of wormhole buffering at the expense of added buffer space. If message lengths are variable and a buffer is not constrained to contain an entire message (but rather a small, fixed number of flits), this approach appears to represent a significant improvement over wormhole's blocking patterns while bounding the additional buffering overhead.

These policies comprise several continua of design features:

---

[2] Note that many common ICNs can be viewed as cube-like structures; see section 5.2.2

[3] *Flit* stands for **fl**ow control dig**it**, a small data packet. The words *flit* and *packet* are used interchangeably here.

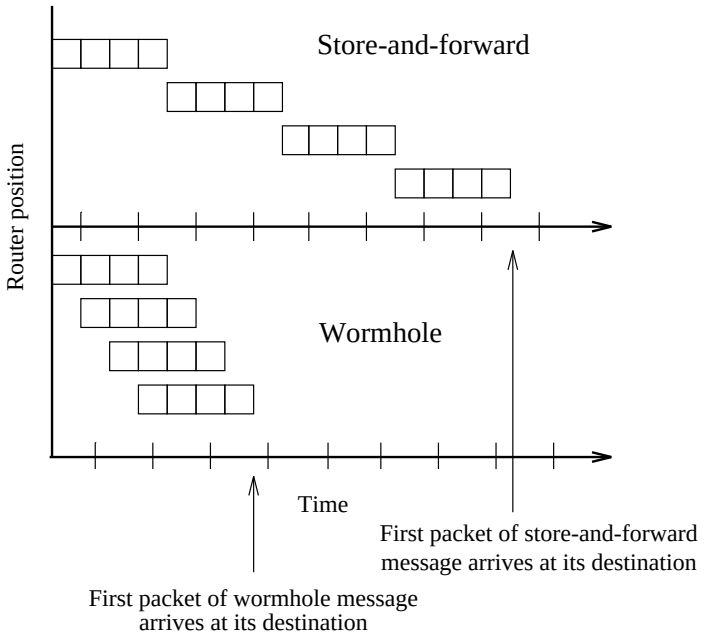

Figure 4.3: Comparison of latency for store-and-forward and wormhole buffering.

- **Implementation cost** as defined by required buffer size: the wormhole approach requires only a flit-sized buffer at each router, whereas store-and-forward demands a buffer at least as large as the longest possible message.
- **Message latency**: under wormhole-style buffering, the first flit of a message may be available at the destination before the last flit has left the origin. In the case of store-and-forward, the head of a message does not reach the destination until all flits have completely propagated through the network.
- **Blocking behavior**: wormhole buffering can easily cause the entire network to block if a circular wait condition arises; this is combated by clever routing techniques including adaptive routing and virtual channels [Dally, 1992]. While effective, these techniques increase the cost of routing logic and buffering resources. Store-and-forward buffering is less vulnerable to deadlock, assuming buffer sizes are sufficient at each router.

Synchronization

All actions in the simulator are coordinated by the master simulation clock. Each tick of the master clock represents some arbitrary time slice that is assumed to be equivalent to the time a packet takes to propagate across a link. We assume that link latency is smaller than router latency; in effect it is subsumed into router latency[4]. As an example, taking the approximations that light travels one foot per nanosecond and that signals propagate across a link at $0.1c$, a one-foot link can be expected to have a propagation delay of 10 ns. Assuming that a router can propagate a message in 10 instruction cycles (perhaps optimistic), and

[4] APES provides a facility to delay the arrival of a packet at the other end of a link for some number of simulation cycles. This is equivalent to making link latency some multiple of router latency, perhaps most useful for simulating widely distributed systems.

assuming that routers operate on a clock of 50 Mhz, or 20 ns, we expect a router to take 200 ns to route a flit. By the time the next flit is routed, therefore, the first flit will have long since reached the other end of the link. The more general assumption is that routing time is greater than link propagation time.

A *simulation cycle* is started by incrementing the clock tick count and then calling simulation routines for each of the subsystems, updating the state of all elements in the simulated system. The state of each part of the system is computed based on its behavioral rules and current state. Once all elements are updated to their new state, the display may be refreshed and the new global state may be written out to the log file.

4.4 Functional elements

Figure 4.4 illustrates the relationships among the various elements that comprise a simulated processing node. The *routing/buffering logic* implements whatever schemes were selected at runtime. The *master simulation clock* synchronizes all operations. A *router port* exists for each link in contact with the PE; the number of ports could differ among PEs in an irregular network. Each router port has a *send and a receive queue* to handle traffic in and out of the associated communication link. The queue sizes are fixed at runtime, and are varied with the choice of buffering method. Communication links are implemented as simple FIFO buffers (although we assume that a link can hold only one 'packet' of data at a time). *Message injection* is controlled by either a communication profile derived from trace analysis or by a random variable.

Processing elements

The basic object in any multicomputer system is the **processing element** (PE). In the context of the simulator, a PE is defined as an entity that generates and routes message traffic. In any particular simulation cycle, a PE may be idle or may generate one or more messages to be propagated through the ICN.

The PE abstraction is effectively a finite-state machine whose transitions are governed by some internal state — in trace analysis mode, the presence or absence of a pending message event, and in stochastic mode the result of a weighted coin toss.

In the former case, each PE can be independently programmed with respect to the ordering of message events. The events are derived from a user-supplied file of message events derived from an application profile.

For stochastic system modeling, a set of global parameters governs the probability of each PE generating a message during a particular simulation cycle. The probability (drawn from a uniformly distributed random variable) is a user-supplied option. The length (in packets) of each message generated can be constant or similarly controlled.

Routers

Processing elements communicate with one another via **routers**, which interface to the ICN itself. Each PE contains a router, and each router contains a **port** for each communication link impinging on the PE. Each port is comprised of a send and receive queue whose lengths are global parameters set by the user. The queues themselves contain logic to determine full/empty status. Routing and buffering decisions are made by logic

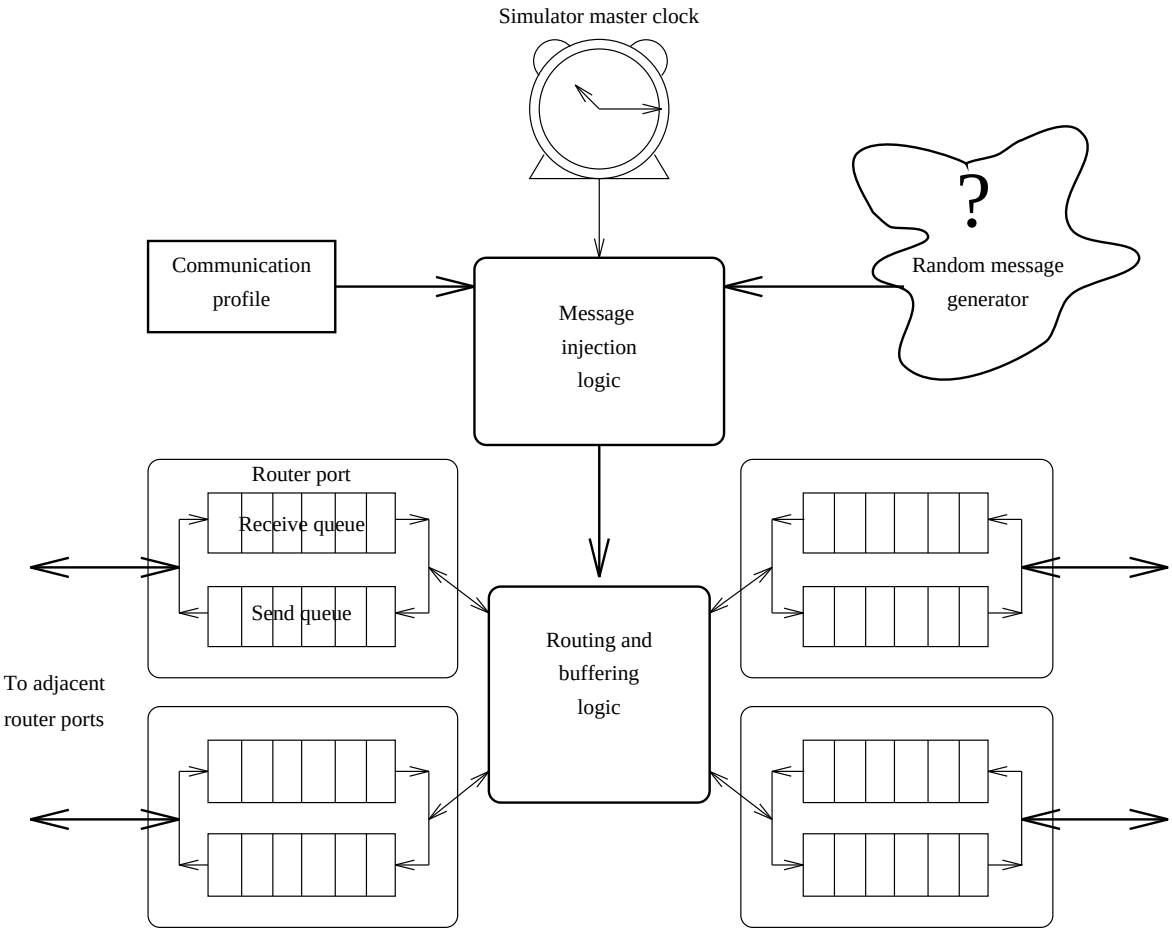

Figure 4.4: The abstraction of a simulated processing node.

within the router. This logic arbitrates among conflicting requests for send and receive queues in the ports and determines where to forward messages according to the routing algorithm.

Routing decisions are made according to either a global routing table or a dimension routing algorithm. The latter case, though less general, does guarantee freedom from deadlock under appropriate buffering conditions. A provably deadlock-free routing algorithm is useful since adaptive routing is not currently supported. The global routing table can embody an arbitrary routing map; the simulator provides a facility to create a best-first routing map by searching the network graph for shortest paths.

Network links

Data are transferred among processing elements via the **network links**. Links provide point-to-point connectivity between router ports; the router ports at each end of a link take care of queuing messages for the link and directing them on their path.

All links are assumed to have the same latency and capacity. The capacity is defined as one flit's worth of data, where a flit is a packet of finite, arbitrary length. In effect, each

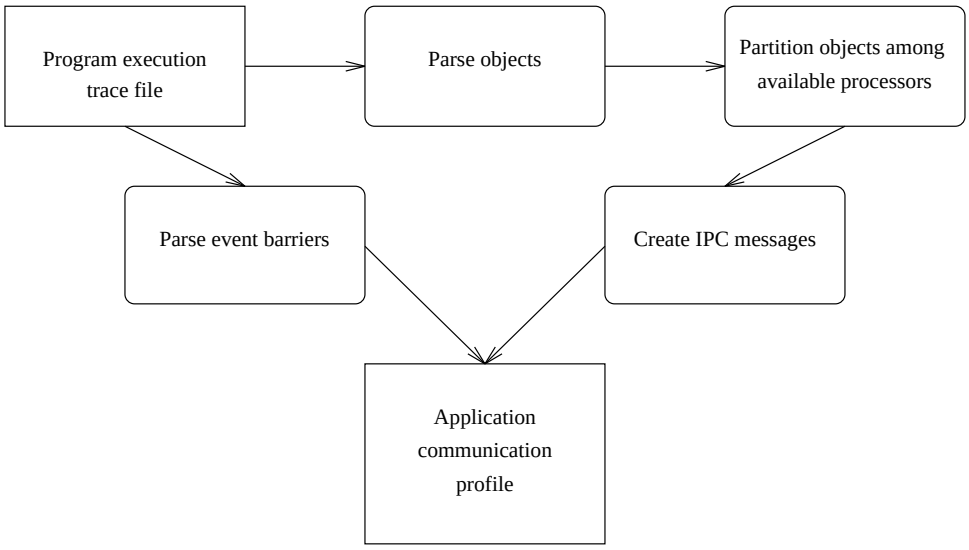

Figure 4.5: The process of generating a communication profile from an execution trace.

link passes an entire flit from end to end in one simulator cycle (see the discussion of timing assumptions in section 4.3).

The latency of a link in the context of a simulation is defined as the number of simulation cycles a packet takes to be delivered end-to-end. This value is assumed to be one for all links, although support for a user-definable *effective length* is available on a per-link basis. When this value is set greater than one, delivery is delayed for the specified number of cycles.

4.5 Mapping instruction traces to the simulated machine

When the simulator is being used to analyze execution traces, it takes as input *communication profiles*, specially formatted files of abstract simulator instructions generated by analysis of the actual execution traces. Communication profiles are generated by a series of separate analysis modules which abstract data movement instructions in the execution trace into simulated message traffic. This process is illustrated in figure 4.5.

The communication profile consists of a set of groups of potentially concurrent message events. Each group of events is delimited by synchronization barriers derived from a streamlined data dependency analysis of the application program. Since in the general case this is an NP-hard problem [Helmbold and McDowell, 1991], we make the simplifying assumption that a group of read accesses not separated by a write constitute a potentially concurrent segment. Although weaker than actual dependency analysis, we suggest that this approach abstracts enough of the application's object access behavior to constitute a valid model.

Initially, the target application must be instrumented so that all data accesses that are (or may be) the cause of a communication event are written out to a trace file. Each data object must have a unique label; this could be a real memory address or an arbitrary handle. Each potential event may also have a transfer size associated with it which eventually gets translated into some number of packets in the corresponding message. The trace file also

contains information regarding barrier points in the program for later creation of barriers in the communication profile.

The data movement operations described in the trace log are parsed into sets of potential message events according to the barrier points included in the trace or to user-supplied information derived from analysis of the code itself. An intermediate file is produced which contains barrier segments, each consisting of one or more communication events. During the generation of this file, a matrix of object relations (based on the addresses or handles supplied in the trace log) can be built; this matrix contains a row and column for each unique object, and the entry for each pair of objects indicates the number of contacts between the two objects in the trace log. This can be used for determining allocation of objects to processors in the next step.

The object handles or addresses are then resolved into processor IDs and written as messages to the communication profile. Allocation of objects to PEs can use information in the relation matrix to place in proximity those objects most likely to communicate. The problem of optimal allocation is at least NP-hard; a 'reasonable' allocation may be computed using cluster analysis or other statistical methods. Alternatively, a random allocation strategy can be used. In either case, the resulting file is the communication profile that is fed to the simulator. Each message event in the communication profile represents some data access in the program trace; each segment of events represents a set of potentially concurrent accesses.

4.6 Stochastic simulation

In the case of a stochastic simulation, the action of each processor at each time step as well as the destination of each message are governed by statistical distributions whose parameters are supplied by the user. The parameters depend on the architecture of the processors and on assumptions about distribution of data in the system over time. Representative distributions for various applications can be generated by conducting statistical analysis of the application program's behavior. This simulation strategy extends the design philosophy described in [Henessy and Patterson, 1989]. The idea is to analyze many variants of a particular idea without losing the underlying abstraction and without requiring extremely detailed simulation of each processor.

4.7 Simulation data

Massive amounts of data are generated during a simulation. Structures within APES itself keep track of every packet's progress; this is too fine a degree of granularity for the qualitative analysis described as the simulator's goal. Results are therefore presented as higher-level combinations of the available data. The most interesting of these is the question of how 'busy' communication links are during the simulation.

APES tabulates the contents of each link during each simulation step. A key metric to be examined is the behavior of the *load value* for each link, computed by counting the number of packets traversing the link within a user-supplied time window and dividing by the window size, yielding a normalized moving average of link activity. If W is the size of the window (in simulation steps), then the load value for the jth link at simulation step t is given by

$$L_t^j = \frac{1}{W} \sum_{i=t-W}^{t} T_i^j$$

where

$$T_t^i = \begin{cases} 1 & \text{if link } i \text{ is busy at time } t \\ 0 & \text{otherwise} \end{cases}$$

Note that L_t^j is normalized to the interval $[0 \ldots 1]$; as the window size is decreased the load average approaches unity, in accordance with the intuition that a narrower link is busier than a wide one. As window size is increased the load value decreases as would be expected if the link were fattened. The *peak* load value for each link, its greatest value over the course of a simulation run, is a measure of the worst-case behavior of the link.

The total number of packets crossing a link over the entire simulation run is another interesting performance metric. This is given by

$$T_{tot}^j = \sum_{i=0}^{S} T_i^j$$

where T_{tot}^j is the raw traffic for the j-th link, S is the total number of simulation steps, and T_i^j is as described above.

Message latency M is the number of simulation cycles a message takes to arrive at its destination. The simulator keeps track of $\overline{M}$, the mean latency, and M_{max}, the greatest latency of any message.

The mean and variance (over all links j) of L_i^j and T_i^j are calculated and displayed at the conclusion of a simulation run (or during the run, if interactive mode is in effect). For greatest flexibility in evaluation of the result, the output format is compatible with a suite of auxiliary statistical analysis tools. These tools are used to generate normalized distributions which can be viewed using a tool such as Gnuplot [Williams and Kelley, 1990].

In addition to statistical summaries, the simulator can output data in graphical format during the simulation run. Although of limited resolution due to the typically large number of objects that must be displayed, these runtime graphs can be very useful in pinpointing hotspots and visualizing traffic distribution over time.

5. Experiments

5.1 Overview

We describe a series of experiments conducted using the Associative Processing Environment Simulator (APES) and Morph communication profiles prepared as described in section 4.5. The objectives of these experiments were:

1. To characterize the interprocessor communication (IPC) behavior of a parallelized AI application on a variety of interconnection network (ICN) topologies.
2. To compare the resulting IPC traffic patterns with the behavior that would be predicted by using an assumption of uniformly-distributed message traffic.
3. To characterize the relationships between ICN topology and communication behavior for both the parallelized application and the uniform-traffic assumption.

The experimental procedure began with selection of the target ICNs and preparation of the Morph trace data. Next, the IPC traffic of each topology was simulated under both a communication pattern derived from the Morph trace and an assumption of uniformly-distributed message traffic. The resulting data on link traffic and message latency were tabulated and plotted.

In all cases the cumulative traffic through each link evinced an approximately normal distribution. Peak link load, a performance metric described in section 4.7, was also normally distributed in the case of uniform traffic. In the Morph-profile case we consistently found a subset of heavily loaded links that dramatically increased the variance in peak load. In combination with the approximately normal distribution of total link traffic, we suggest that this finding indicates the presence of transitory hotspots. We speculate that this hotspot behavior results from structure within the associative database (ADB) being mapped across processor boundaries in the network being simulated.

There appeared to be a clear relationship between the bisection width of a network and the mean value of normalized link traffic. A similar relationship was found between *aspect ratio* (see section 5.2.2 below) and mean peak link load for the case of uniformly distributed message traffic. There was also some relation between latency and aspect ratio in both traffic cases.

5.2 Procedure

5.2.1 Generating the communication profile

We instrumented Morph [Levinson *et al.*, 1992], a chess-playing program that uses adaptive/predictive search (APS) techniques to learn the game. At the heart of Morph is an associative database (ADB) of pattern objects derived from game positions. Accesses to the database were monitored over a number of execution runs, and the resulting trace data analyzed to create an ordered list of object access events. This list was then transformed into a series of partial orders representing groups of potentially parallelizable events. The ordered set of such groups was then partitioned according to implicit synchronization barriers derived from analysis of the program code.

The resulting list of sets of potentially parallel object access events represents the raw material of the communication profile to be simulated. The final step is the partitioning of ADB objects among the available processors. This was done on a random basis, assigning database objects to simulated PEs according to a uniformly-distributed variable. We chose random allocation because it reflects the behavior of an actual system in the absence of dynamic load balancing. Although an allocation scheme based on analysis of the database could have provided better performance, doing so would have meant making assumptions that were stronger than those that an actual APE system would be able to make.

The result of the partitioning is a 'script' of message events, or *communication profile*, representing communication among the PEs during program execution. The communication profile represents a standard data set that can be run on any desired topology containing the number of PEs selected in the partitioning step. Since the profile must be recreated if a different number of PEs is required, in these experiments each set of topologies with the same number of PEs shares the same profile.

Greater detail on the generation of the communication profile can be found in section 4.5 and in Appendix A.

5.2.2 The experimental topologies

The topologies used in the experiments were selected from the family of k-ary n-cubes [Dally, 1990]. This class of topology is of considerable interest because isomorphisms exist between k-ary n-cubes and a wide array of other topologies including multistage switching networks, meshes, tori, and hypercubes. Because of the networks' regularity, the switching elements required to implement routing can be designed compactly. In addition, efficient, straightforward deadlock-free routing algorithms are available.

One metric that can be used to classify k-ary n-cubes is *bisection width*, described in [Dally, 1990]. The bisection width $B(k,n)$ of a k-ary n-cube network is defined as the minimum number of links that cross an even partitioning of the network, that is, the smallest number of links that can be cut when the network is divided into two equal-sized parts. Consider such a network[1] embedded into a 2-dimensional plane such that $n/2$ of its n dimensions are assigned to each of the two planar dimensions. If there are $N = k^n$ processing nodes, there will be $\sqrt{N} = k^{n/2}$ rows and columns in the planar network. Suppose that we look at just the internode links in the highest dimension — these are the links crossing the midpoint of the network. There are $2k^{n/2-1}$ such links in each of the $\sqrt{N}$ rows (one link passing data in each direction per processor pair in the row), or $2\sqrt{N}k^{n/2-1}$ links total. The bisection width B is defined as

$$B(k,n) = 2\sqrt{N}k^{n/2-1} = \frac{2N}{k} = 2k^{n-1}$$

Bisection width is in effect a description of the network's capacity to move data. It is related to the network's dimension, but is a measure not just of *diameter* (equivalent to dimension for k-ary n-cubes), or how far a packet may have to travel, but of *capacity*, the number of packets tht can be in transit simultaneously. For a constant link capacity, higher-dimensional networks have a higher bisection width and require more and longer wires; long wires are slower than short ones and require more power to drive. Lower-dimension networks are cheaper but may not have the routing capacity of high-dimensional networks (having fewer dimensions means there may be more contention).

[1] We assume for the remainder of this discussion that k, the radix of the network, is even.

Topology	Radix k	Degree n	Size N	Bisection width $B(k,n)$	Aspect ratio $A(k,n)$
2,6 cube	2	6	64	64	1.0
4,3 cube	4	3	64	32	0.5
8,2 cube	8	2	64	16	0.25
16,2 cube	16	2	256	32	0.125
2,8 cube	2	8	256	256	1.0
4,4 cube	4	4	256	128	0.5
8,3 cube	8	3	512	128	0.25

Table 5.1: Experimental topologies and their attributes.

We introduce as a further characterization of an ICN's density the *aspect ratio*, defined as the ratio of bisection width to processor count. This is given by:

$$A(k,n) = \frac{B(k,n)}{N} = \frac{2k^{n-1}}{k^n} = \frac{2}{k}$$

Consider the partitioned network again; if the aspect ratio is low, many processors are wanting to use few available links to get to the other partition. If the ratio is high, the number of available links and the number of waiting processors is well-matched so that the network can be considered to be more 'efficient'.

We will present both bisection width and aspect ratio as metrics in the discussion of the experimental results. The particular topologies that were chosen for the experiments represent a spectrum of both these metrics. Table 5.1 identifies the characteristics of each of the experimental topologies.

5.2.3 Experimental parameters

Experimental runs were done for each topology using the Morph communication profile and a uniformly-distributed random traffic pattern chosen to approximate the profile's probability of message generation. Several communication profiles were created; the initial trace data were the same in all cases, but the mapping of objects to PEs, and therefore the patterns of traffic, were different for each number of PEs. Each topology containing the same number of PEs used the same profile. A total of three different random allocations were used in the experiments, one for each of the network sizes of 64, 256, and 512 PEs. The similarity of the peak load results (see Section 5.3.1) for each of the networks indicates that one random allocation is probably equivalent to another.

In the uniform-traffic studies, we assumed a constant probability of a PE generating a message in any particular simulation cycle. This probability was determined by a rough examination of the communication profile. If the profile contained m message events and ran to completion in c simulation cycles, we assumed that the probability of a PE injecting a message was given by

$$P \simeq \frac{m}{c}$$

We found the value of P to be near 3 % in each case that was examined, so this value was used in all the uniform-traffic experiments.

All messages were defined to be 4 packets in length. It would have been simpler to make them a single packet long, but we wanted to stress the network in an effort to discover issues relating to link contention over time. Another possibility (more realistic in terms of the data structures manipulated by Morph's ADB) would have been to allow message lengths to vary randomly on some range. Although interesting from a practical point of view, we felt that this would further muddy the observations of link traffic and latency due to the additional irregularity in network usage.

The routing method used was a straightforward dimension routing algorithm that guarantees freedom from deadlock. Wormhole buffering was chosen as the simpler of the two alternatives. Store-and-forward routing, in addition to its higher latency, required large buffers to be allocated at each node in order to avoid link starvation; the considerable overhead due to memory paging of the buffers led to unacceptably long simulation times for the larger networks.

5.2.4 Gathering and processing experimental data

Since we were interested in network utilization, we chose to examine statistics related to the measurement of traffic on each link. Simulation data were processed to obtain a record of total traffic and peak load for each communication link. The individual link data were analyzed to obtain mean and variance, and a discrete distribution function was computed. The distributions for each experimental run were then plotted and compared.

The key metric we examined was the *load value* for each link, a normalized moving average of link activity described in section 4.7. The maximum or *peak* load value for each link over the course of the simulation was kept as an index of the worst-case behavior of each link. We used a window size of $W = 75$ cycles in all the experiments; the resulting peak load values were well distributed over $[0 \ldots 1]$.

Since peak link load describes the greatest load on a link over some *interval*, if a particular link's peak load exceeds the mean peak load we infer that that a statistically unusual number of messages were passed over this link in a short time at some point during the simulation. If clusters of objects exist such that their members tend to be accessed in closer-than-average temporal proximity, we would expect the links spanned by the cluster to exhibit high peak load.

Because peak load may represent a very transitory phenomenon, we also examined the total traffic (i.e. number of packets) across each link in an effort to characterize the hotspot behavior. The raw traffic figures were normalized in each case to the total number of packets injected into the network over the course of the entire simulation; the resulting value represents the number of packets in a given link per packet injected into the network, intuitively the contribution of each link toward the total traffic in the network. We expected that the distribution of normalized link traffic would be approximately uniform since whatever locality of access existed in the original database should be evenly dispersed across the PEs due to the random partitioning of database objects to PEs.

A final area of network behavior that could be examined in order to characterize the effects of contention is message latency, defined as the number of simulation cycles a message takes to arrive at its destination. We found it impractical to collect complete latency data due to the large number ($\approx 10^5$) of messages in each profile. The data that were collected

Topology		Uniform		Trace	
Name	Aspect	$\overline{L_{peak}}$	$\sigma^2_{L_{peak}}$	$\overline{L_{peak}}$	$\sigma^2_{L_{peak}}$
2,8 cube	1.0	31.79	21	44.53	1236
2,6 cube	1.0	32.19	19	56.19	1086
4,4 cube	0.5	32.29	25	46.57	1264
4,3 cube	0.5	32.55	23	59.93	1015
8,3 cube	0.25	48.07	30	48.32	1422
8,2 cube	0.25	48.84	27	67.37	976
16,2 cube	0.125	72.03	37	61.43	1367

Table 5.2: Peak link load for uniform and Morph-derived traffic patterns.

include mean and maximum[2] latency for each simulation run. Although not sufficient to compute a distribution function, we suggest that these data may be of value in further describing IPC behavior.

5.3 Results

5.3.1 Peak link load

The results of the measurements on peak link load are summarized in table 5.2. We find that in each case the Morph communication profile engenders a large variance in peak load, indicating that some irregularity exists in the traffic pattern.

In the uniform-traffic cases, we note that the mean peak loads fall into three clusters, each corresponding to a different range of aspect ratio $A(k, n)$. We suggest that this correlates with our intuitive notion of aspect ratio as a measure of the ICN's efficiency, in that the topologies with the greatest value of A, such as the 2,6 cube and 4,3 cube, show generally lower peak usage than those with smaller A such as the 16,2 cube.

The dependence on A is not evident for the trace-generated data; presumably the irregularity in the traffic patterns in this case overshadows the behavior.

Plotting discrete distributions for each case as illustrated in figures 5.1 and 5.2, we see that the uniformly-distributed random traffic displays an approximately normal distribution of peak link load while the trace-derived traffic shows a similar distribution with the addition of a sharp peak at the maximum-load end. We take this as an indication that a subset of links has undergone one or more episodes of high usage, while the remainder have experienced approximately uniform traffic.

5.3.2 Cumulative link traffic

We explored measures of cumulative traffic across links in order to further characterize IPC behavior. The total number of packets passing over each link during a simulation run was normalized to the total number of packets injected. This gave a measure of the amount of work each link had to do in response to each packet injected into the network. Variances were also computed and normalized. These values are summarized in table 5.3. Note that

[2] Since we use wormhole routing the minimum latency is always just the number of flits in a message, in this case 4.

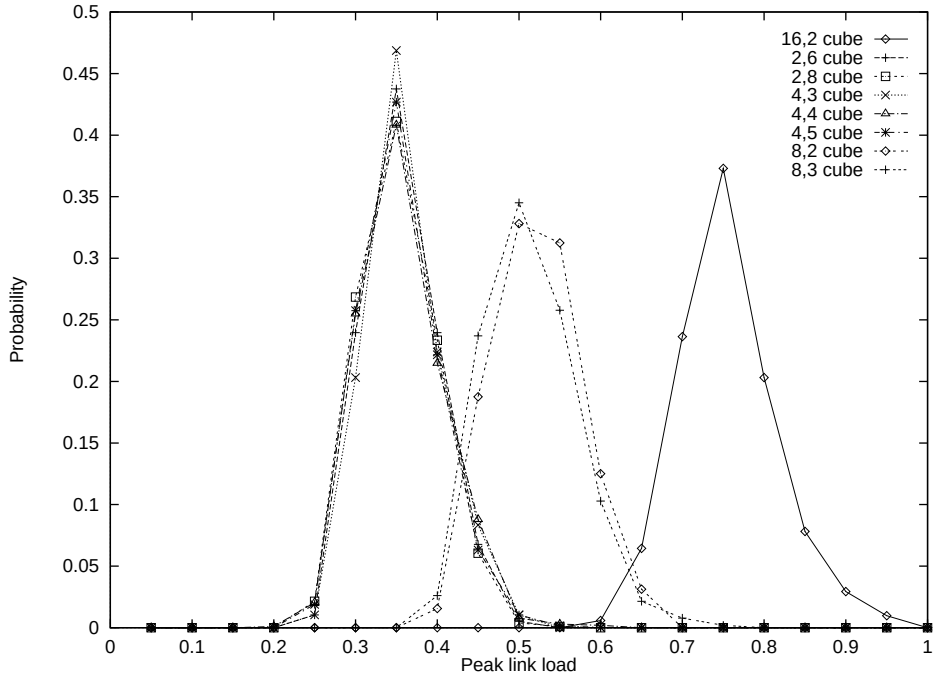

Figure 5.1: Probability distribution of peak link load from uniformly-distributed random traffic.

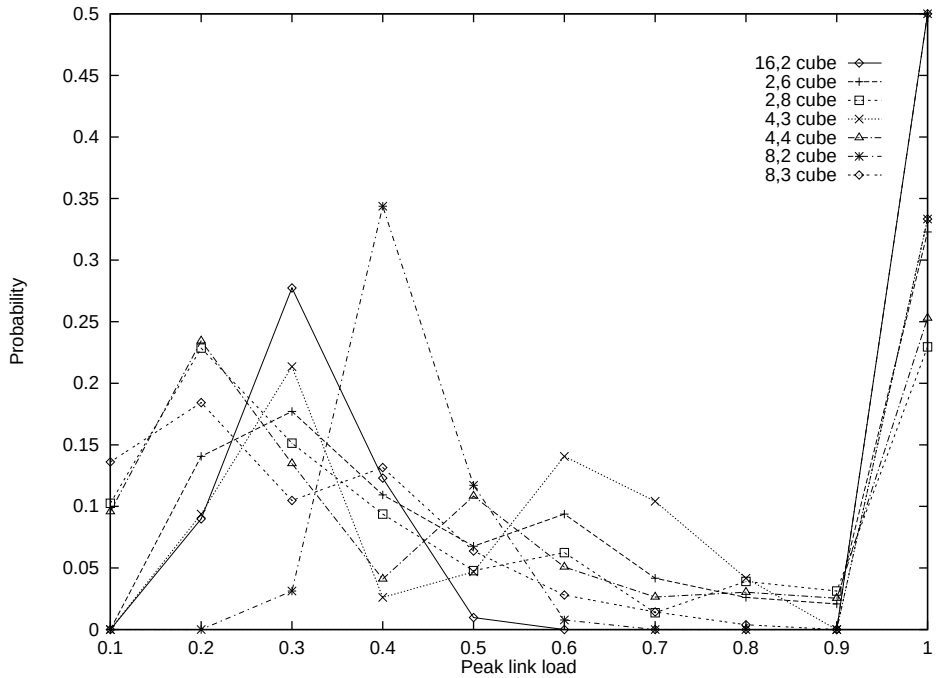

Figure 5.2: Probability distribution of peak link load for traffic derived from the Morph communication profile.

Topology		Uniform[a]		Trace	
Name	Width	$\overline{T_{norm}}$	$\sigma^2_{T_{norm}}$	$\overline{T_{norm}}$	$\sigma^2_{T_{norm}}$
8,2 cube	16	29.8	1.4	31.7	19.3
16,2 cube	32	13.8	0.2	15.7	27.9
4,3 cube	32	15.6	1.4	15.9	9.6
2,6 cube	64	15.5	1.2	15.9	10.5
4,4 cube	128	3.8	0.1	3.9	5.4
8,3 cube	128	3.7	0.2	3.9	6.6
2,8 cube	256	3.8	0.1	3.9	5.3

Table 5.3: Normalized cumulative link traffic.

[a]Note that all measurements in this table have been scaled by 10^{-3} for readability

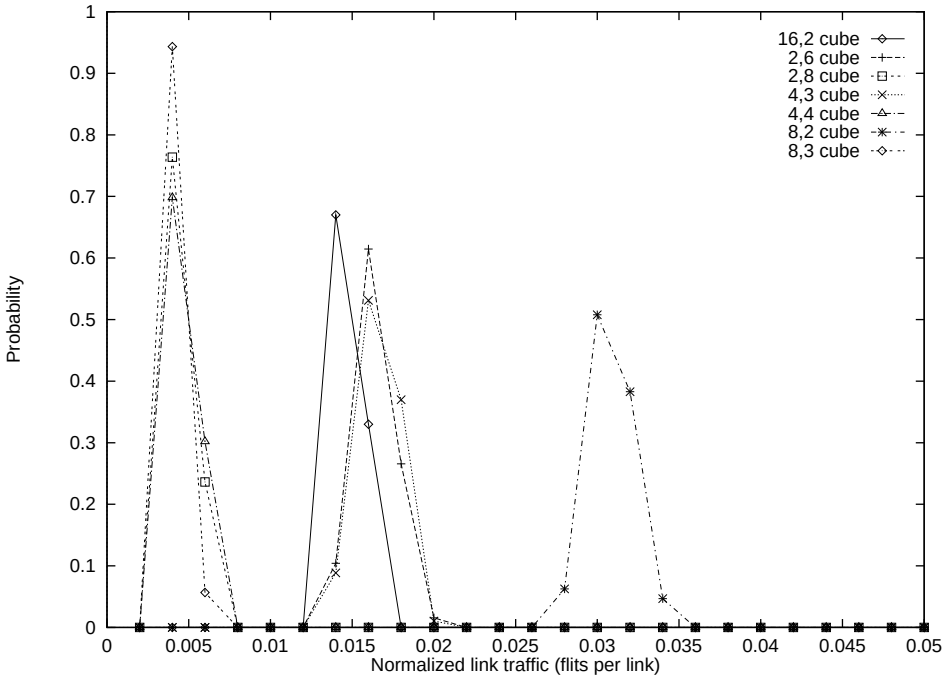

Figure 5.3: Probability distribution of normalized link traffic for uniformly-distributed random traffic.

in order to make the table more legible, all the actual normalized traffic figures were scaled by 10^3.

Figure 5.3 illustrates the discrete probability distribution of traffic for the uniform-traffic case. The distributions are narrow, indicating little variation among links, and are clustered into groups with common bisection width. We can conclude from the small variance that under uniform traffic conditions links are loaded evenly and contention is not a major factor in network usage.

Distributions for the trace-derived traffic, shown as figure 5.4, are quite similar to those for uniform traffic. Their greater variances are visible as wider ranges of values, consistent with the hypothesis of nonuniformity in the distribution of IPC traffic. The groupings according to $B(k, n)$ are virtually identical to those in the uniform-traffic case.

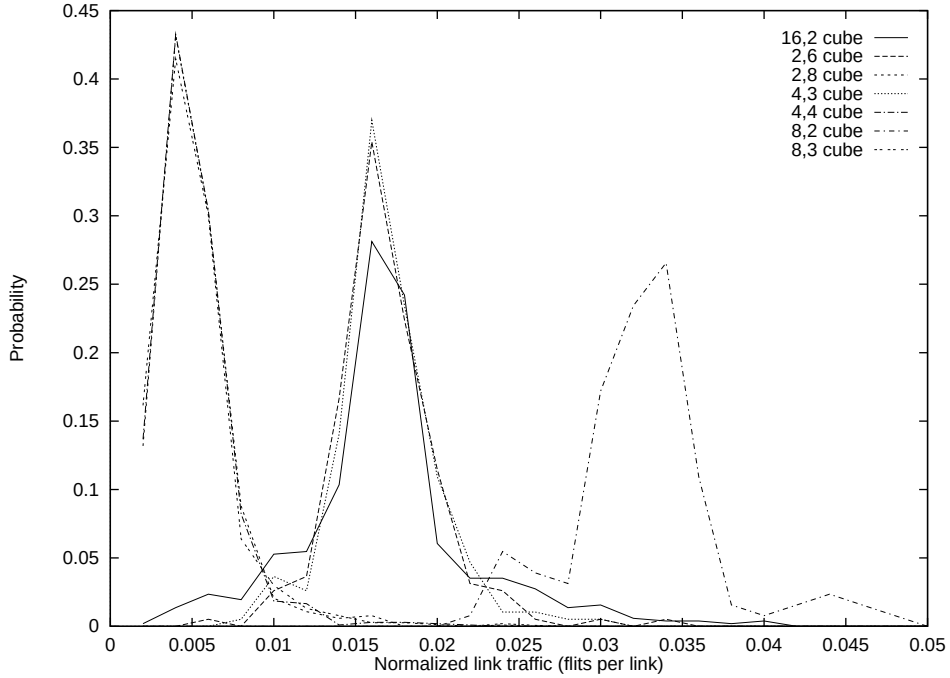

Figure 5.4: Probability distribution of normalized link traffic derived from the Morph communication profile.

The clustering of the means evident in both the uniform and trace-generated cases appears to be due to bisection width B; the 8,2 cube, with $B = 16$, stands apart from the 16,2 and 2,6 cubes, with $B = 32$ and $B = 64$ respectively. These in turn are apart from the 2,8 and 8,3 cubes ($B = 256$ and $B = 128$). This is in accordance with the intuition that for a constant rate of message generation, a 'denser' cross-section of the network requires fewer packets to traverse each link.

Because of the similarity between the normalized cumulative traffic values for uniform and trace-derived traffic, we conclude that the *time-averaged* behavior (i.e. *average* temporal locality) of link traffic in the trace-derived case is essentially the same as the uniform case. The variances, however, differ considerably. It appears that there is a greater degree of spatial irregularity in the trace-derived traffic pattern (i.e. message destinations are not distributed altogether randomly.)

We suggest that the similarity in average temporal locality indicates that the extreme peak loads seen in figure 5.2 represent relatively transitory hotspots, causing certain links to carry unusually high traffic for some period of time small enough that the cumulative effect was too small to skew the cumulative traffic very much.

5.3.3 Latency

Message latency data are shown in table 5.4. The column labelled $\frac{M_{max}}{\overline{M}}$ describes the ratio of maximum to mean latency, a measure of the magnitude of worst-case behavior in the network. The relationship of mean message latency to aspect ratio is shown for the trace-derived and uniform-traffic cases in figure 5.5. Note that the vertical axis is logarithmically scaled, and that data points have been interpolated between topologies sharing an aspect ratio.

Topology		Uniform			Trace		
Name	$A(k,n)$	$\overline{M}$	M_{max}	$\frac{M_{max}}{\overline{M}}$	$\overline{M}$	M_{max}	$\frac{M_{max}}{\overline{M}}$
16,2 cube	0.125	12.6	46	3.65	61.8	185	2.99
8,2 cube	0.25	7.5	23	3.07	51.5	184	3.57
8,3 cube	0.25	9.9	34	3.43	56.8	191	3.36
4,3 cube	0.5	6.3	18	2.86	43.6	180	4.13
4,4 cube	0.5	7.4	26	3.51	47.7	188	3.94
2,6 cube	1.0	6.2	17	2.74	44.1	188	4.26
2,8 cube	1.0	7.2	22	3.06	47.7	181	3.80

Table 5.4: Message latency (in simulation cycles)

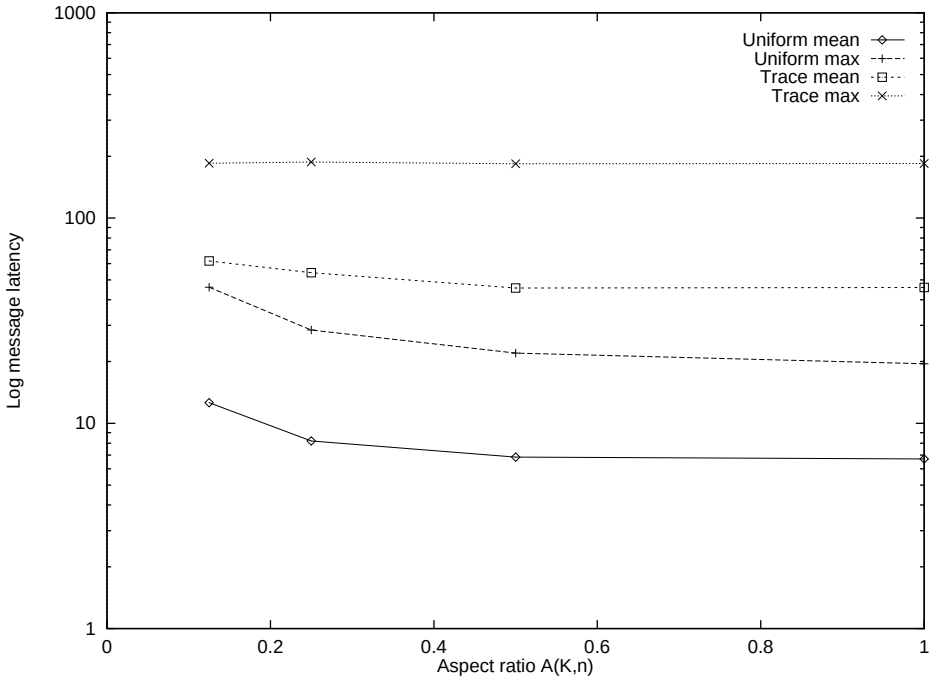

Figure 5.5: log Message latency versus aspect ratio.

Once again, we see similar behavior between the two traffic sources, with the trace-derived latencies being about an order of magnitude greater than those due to uniformly-distributed traffic. Both the mean and maximum latencies are presented in the plot; it is interesting to note that in both traffic cases, the maximum is a fairly constant factor greater than the mean latency. The constant factor differs between the two traffic sources: it is approximately 3.19 for uniform traffic and 3.72 for the trace-derived traffic. Presumably the difference is due to the greater variance in load for the latter case, leading to longer waits when messages block.

We assume that the greater message latency in the trace-derived case is due to excessive contention during hotspot activity. The role of the ICN topology is prominent in that topologies with lower aspect ratio, and therefore more contention for fewer available links, experience greater latency than the others. We take this as an indication that, ignoring hotspots, trace-derived message patterns are similar to those under the uniform assumption

(and therefore react to topology changes in a similar way). Contention due to hotspots increases average latency, but network topology is still the determining factor in the network's response to traffic.

5.4 Discussion

The following observations summarize the experimental results:

1. In the case of uniformly-distributed random message traffic, both peak load and link traffic show a normal distribution.
2. Peak link load is unevenly distributed for trace-derived message traffic; in each case, a portion of the distribution is approximately normal while the rest is clustered at the high-load end.
3. Link traffic in the case of trace-derived message does not show the cluster of extreme values seen in the peak load distribution. There is a greater variance than the uniform case, however, indicating that a good deal of nonuniformity is present in the distribution of messages.
4. Aspect ratio appears to be a factor in determining the effect of peak link load on a network under the assumption of uniformly-distributed traffic. The greater the ratio, the lower the peak load tends to be. This appears to validate our intuition that aspect ratio represents a measure of the network's efficiency or grace under stress.
5. Networks with greater bisection width display lower normalized link traffic than those with smaller B. The more links are available to shuttle packets over, the fewer packets need to travel over each link.
6. Message latency is much greater in the trace-derived than the uniform simulations, presumably due to a high degree of network contention.

A common method of evaluating ICNs is to assume a uniform random distribution of message traffic to model IPC behavior. This approach is demonstrably valid for many 'typical' parallel applications [Patel, 1981]. The experiments described here, however, indicate that the uniform communication model is not altogether valid in the case of the applications of interest in this research.

The discussion of hotspots in [Kumar and Pfister, 1986] suggests that the behavior of an ICN in the presence of transient overloads tends to be highly unstable; in current terminology we might consider the behavior to tend toward chaoticity. Given that chaotic behavior in such a network can be anticipated, design choices should be made as early as possible in order to maximize throughput and minimize unstable network behavior.

The results presented here indicate that hotspots may present a significant source of latency and network congestion for the class of applications under consideration. We suggest that traffic simulation is a useful adjunct to analytical methods based on assumptions of uniform traffic.

6. Further work

6.1 Enhancements to the simulator

Several areas of the simulator could be enhanced. There are issues of both efficiency and capability.

- Increased efficiency — The current implementation of the simulator devotes relatively little attention to issues of efficiency and performance. We estimate that optimizing the code would speed up simulations by an order of magnitude.
- Better visualizations — It would be instructive to see a rendered, animated image of the traffic in the network itself as the simulation is running; the search for emergent behaviors would be greatly facilitated by this feature.
- Individual distribution parameters for each node — more detailed probabilistic simulations could be done if each node could be programmed independently.
- Virtual-circuit buffering [Dally, 1992] — An adjunct to adaptive routing, support for this buffering scheme would greatly expand the range of system parameters.
- Adaptive routing — A number of adaptive routing algorithms with favorable behaviors are described in the literature [Glass and Ni, , Chien and Kim, 1992, Konstantinidou, 1990], particularly in conjunction with virtual-circuit buffering mechanisms.
- Parallel implementation — The simulator is a good candidate for porting to a massively parallel SIMD platform such as the Maspar MP-1 [Nickolls, 1990]. Since APES has very regular patterns of access in its operation, it would map well to a data-parallel programming model.

6.2 Parallelizing the applications

Additional work also remains in the methods of parallelization; although the scheme used in these experiments faithfully reflects the behavior of the serial Morph application, an ADB access method *designed* for parallel implementation could have different access patterns.

A variety of different current AI methods should be parallelized and simulated to develop a better model of the underlying issues they may share.

6.3 Assignment and load balancing

The problems of distributing objects among available processing elements and migrating them as the database changes are crucial to efficient operation of an APE system. The criteria used to evaluate the result of a given assignment or load-balancing policy are highly dependent on the system's architecture as well as the structure of the problem domain. A wide variety of assignment and load-balancing mechanisms and policies are possible, and experimentation under varying assumptions about problem domain and system architecture represents an effective way to evaluate many of them.

There are opportunities for exploration in terms of the allocation of objects to PEs. The random allocation method used here, although practical, fails to exploit the structure of the database; our reasoning is that a 'real' system would not have access to the database ahead

of time to decide how to allocate. Simple heuristics may exist, however, that would enable allocation to be done bottom-up in a manner more supportive of desirable IPC behavior (i.e. relatively uniform communication). Exploring simple allocation methods would be a worthwhile study that could easily be done using the facilities available in APES.

6.4 Emergent properties

Complex nonlinear systems generally exhibit high-level behaviors that are not obviously related to conditions measurable within the systems themselves [Gleick, 1988]; observation of these macrobehaviors can yield valuable insights into levels of underlying order in the system [Forrest, 1991].

The adaptive-predictive search (APS) paradigm as described in [Levinson *et al.*, 1992] is predicated on non-deterministic recombination of feature patterns in an effort to extract structure from the underlying representation. To the extent that this process represents a nonlinear transformation of problem domain into feature space it seems reasonable to expect that emergent properties of the system implementing APS itself represent yet another level of structure that is likely to be semantically rich.

The simulator could be used as a tool to develop an understanding about patterns of order within a complex system implementing APS. For example, consider the coupling between macrobehaviors of the system, such as patterns of interprocessor communication, and microbehaviors such as the degree of weight change in particular sets of graph objects. Although some relation clearly exists, it would be difficult to predict how a given change in feature space (i.e. in the system being learned about) might reflect on the state of the APS system itself. Strategies developed for observation of complex collective systems [Kephart *et al.*, 1991] could be applied to simulation results. Learning about how experimental 'intelligent' systems react to varying environmental conditions would be an invaluable adjunct to the eventual design of effective 'real-world' systems.

7. Conclusion

A computing environment supporting a rich set of associative-processing primitives is a requisite platform for experimental research in the area of artificial intelligence applications. The analytical techniques that have been applied to the design of conventional computing systems fail to completely capture the nonlinear behaviors of such applications.

The simulator described in this thesis provides a flexible and readily configurable platform for the evaluation of a wide variety of design alternatives for such a computing environment. The experiments that are described use APES to compare the communication behaviors of a parallelized artificial intelligence program with those arising from an assumption of uniformly distributed traffic. The simulation results indicate that although much of the program's behavior approximates that derived from the uniform traffic assumption, significant hotspots arise and affect network throughput and latency.

We conclude that patterns of activity based on semantic relatedness among objects in the associative database give rise to communication patterns that are not modeled effectively by a uniform distribution of message traffic. Some adjunct to uniform analysis is appropriate and we suggest that simulations along the lines of those presented here represent a step in the direction of a more comprehensive design methodology.

Appendix A. Modelling distributed APS using Morph

A.1 Overview

In the interest of developing a generalized model of the issues that might be important in the design of a distributed system capable of efficiently implementing adaptive-predictive search (APS), it is instructive to examine the behavior of one particular application that incorporates the APS metaphor. Morph [Levinson *et al.*, 1992] is a chess-playing program that has been used as a testbed for many of the ideas underlying the APS model.

We make the assumption that Morph's pattern database, which is in fact an associative database (ADB), is to be distributed across a network of computing nodes supporting an associative processing environment (APE). Each processing node is expected to maintain a subset of the entire database. When a particular node's search needs to access an object that is nonlocal, interprocessor communication (IPC) must occur. Because the APE nodes can execute associative-array operations efficiently, the latency due to network traffic among the processors may represent a significant performance bottleneck for the system.

In order to explore the behavior of IPC requirements, Morph is instrumented to produce traces over a series of execution runs. The trace data are analyzed to develop a description of access patterns on the ADB. A simple parallelization is done by translating the access patterns into a model of IPC for a distributed ADB.

Because Morph is a representative implementation of the APS approach, we suggest that evaluation of its simulated IPC behaviors is likely to reflect issues that are common to any distributed APS system.

How Morph works

Morph was designed to learn the game of chess by playing a large number of games against an expert opponent (the GnuChess program, originally developed by John Stanback) and extracting rules and strategy from the interaction of its knowledge database with the outcomes of the games that are played. Patterns are added to the database as Morph's experience grows, and the weights associated with patterns leading to winning situations are reinforced by the method of temporal-difference (TD) learning.

The general procedure for an iteration of Morph is as follows (note that several thousand of these iterations were required before Morph won its first game):

1. Play a game, selecting the most applicable move at each opportunity by matching patterns derived from the current board configuration against those in the pattern database.
2. At the conclusion of the game, the weight of each pattern that contributed to the moves that were selected is re-evaluated according to the outcome of the game. This is a simple application of TD learning; periodic feedback from the environment in the form of win/lose status is applied to the internal knowledge base in an effort to reward those data that contributed to a positive outcome.
3. As new board configurations are encountered, corresponding new patterns are added to the database.

A.2 Search in an associative database

The ADB interface is similar to that of a generic database; objects can be added, removed, and updated (accessed then rewritten). An ADB represents a partial order on the relation *more-general-than*, in contrast to a conventional database which represents a total order on some key. Search in an ADB is therefore a computationally demanding problem since no simple indexing scheme can capture the entire partial order.

To enable associative retrieval in the ADB, a graphlike representation such as a semantic network is used to represent patterns. This in effect reduces search to a series of subgraph isomorphism tests. Subgraph isomorphism is a well-explored area of computer science; although it represents an NP-complete problem, Ullmann's algorithm [Ullmann, 1976] appears to make it manageable for graphs of reasonable size, and Willet and Wilson describe an efficient parallel implementation in [Willet and Wilson, 1991].

A.3 Parallelizing Morph's ADB

The search method used in Morph, referred to as Method III in the APS literature [Levinson, 1991], has as its critical 'inner loop' a subgraph isomorphism test. The essence of the parallelization of APS proposed in this thesis is to use the very efficient pattern-matching operations available in the APE's associative arrays to do the graph comparisons, thereby speeding all ADB operations.

We assume that because of its size, the ADB must be spread across a number of APE computing nodes. An attempt to compare two objects residing in different processing elements would therefore engender an IPC transaction. The structure of the ADB is such that any pair of objects may share many common predecessors (i.e. have much in common) and successors (i.e. be generalizations of many of the same objects). This implies that a search for the greatest or most general common ancestor of two objects may cause numerous intermediate objects to be accessed and evaluated. We expect the high branching factor of the ADB to have considerable impact on IPC behavior, since it is improbable that the ADB's partial order hierarchy can be well distributed among the available processing nodes without prior knowledge of its structure.

A.4 Instrumenting Morph

We added code to Morph that wrote trace data to a log file for later analysis. The code modifications demanded by the instrumentation affected 14 of Morph's 78 program files, adding a total of approximately 120 lines of code. No perceptible effect was observed on Morph's performance.

All accesses to Morph's ADB are traced; any operation that accesses an object, such as pattern evaluation or weight update (a read-modify-write cycle), leaves a record in the trace. Each trace record therefore represents some contact with the database; some are used to create barrier synchronization points, and others cause the generation of potential IPC message events.

The primitive operation at the heart of the parallelization is graph comparison. Each time two ADB objects are compared, the ADB code searches the database for their common ancestors and/or descendants, causing a number of graph comparison operations to take place as the partial order hierarchy is traversed. Each of these comparisons represents

a potential IPC event. Whether a message is actually generated or not depends on the allocation of ADB objects to APE nodes; in the trace each ADB object has a unique handle that is later resolved into a PE ID.

A.5 Analyzing the traces

A considerable amount of Morph's implementation effort has gone into optimizing its performance in a conventional workstation environment. Deducing potentially parallel operations is therefore not an altogether straightforward task. The goal is to develop a profile of object accesses in the pattern database which maintains as much as possible of the behavior that a similar ADB distributed on a multiprocessor network would express.

Perhaps the most basic issue in attempting to parallelize a serial application is the establishment of synchronization barriers between sets of events. Suppose we label as an 'event' any operation incorporating access to a database object (i.e. insertion, comparison, and deletion as outlined above). The trace of a program execution is then just an ordered list of events. However, we cannot assume that all the events could have happened at the same time! Clearly there are dependencies among events such that event A (i.e. 'insert object X') must happen before event B (i.e. 'compare objects X and Y'). There are also events that have no dependencies on each other so that they *could* happen in any order.

In the general case, this sort of dependency analysis is at best an NP-hard problem [Helmbold and McDowell, 1991]. A concurrency graph must be generated which represents all possible program states during execution. Analysis of this graph then provides enough information to determine which events *could* have occurred concurrently. These partial orders of events are then organized into barrier sections such that all events in a barrier section must have been completed before those in the next section could be completed. Without prior knowledge of the program's structure this analysis appears to be exponential in character.

Fortunately, the structure of Morph provides us with built in barrier points in the form of its higher–level operations. Since Morph plays a game one move at a time, we assume that the ADB comparisons made during evaluation of a single move are concurrent. Each move in a Morph game represents a set of events (comparisons across the database to select the most applicable move) that could be happening in parallel. Since Morph is simply evaluating all patterns in its database that resemble the current board configuration, there are no constraints on which potential moves get evaluated first, since the ADB is not modified during this phase of operation. In actual fact, there may be some ordering in the comparisons due to the particular details of the matching algorithm, but we claim that as a first approximation it is reasonable to assume that all the comparisons due to a particular move are concurrent. Similarly, we assume that each set of comparisons leading to insertion or deletion of an ADB object is concurrent.

References

[Almasi and Gottlieb, 1989] George Almasi and Allan Gottlieb. *Highly Parallel Computing.* Benjamin/Cummins, Redwood City, CA, 1989.

[Ambriola *et al.*, 1990] V. Ambriola, P. Ciancarini, and M. Danelutto. Design and distributed implementation of the parallel logic language Shared Prolog. *SIGPLAN Notices*, 25(3):40–49, March 1990.

[Arvind and Nikhil, 1987] Arvind and R. S. Nikhil. Executing a program on the MIT-TTD architecture. In de Bakker, Nijman, and Treleaven, editors, *Parallel Archtiectures and Languages Europe*, volume 2, 1987.

[Ballard *et al.*, 1983] D. H. Ballard, G.E G. E. Hinton, and T. J. Sejnowski. Parallel visual computation. *Nature*, (5938):21–26, November 1983.

[Bhuyan *et al.*, 1989] Laxmi N. Bhuyan, Qing Yang, and Dharma Agrawal. Performance of multiprocessor interconnection networks. *Computer*, 22(2):25–37, February 1989.

[Bose *et al.*, 1992] Soumitra Bose, Edmund Clarke, et al. PARTHENON: A parellel theorem prover for non-horn clauses. *Journal of Automated Reasoning*, 8, 1992.

[Burks *et al.*, 1947] Arthur Burks, Herman Goldstine, and John von Neumann. Preliminary discussion of the logical design an electronic computing instrument. Technical report, Institute for Advanced Study, Princeton, NJ, September 1947.

[Chien and Kim, 1992] Andrew A. Chien and Jae H. Kim. Planar-adaptive routing: Low-cost adaptive networks for multiprocessors. In *19th International Symposium on Computer Architecture*, pages 268–277. Association for Computing Machinery, 1992.

[Clocksin and Mellish, 1981] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, Berlin, 1981.

[Cook and L. B. Holder, 1990] D. J. Cook and L.B L. B. Holder. Accelerated learning on the connection machine. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing 1990*, pages 448–454. IEEE Computer Society, 1990.

[Dai *et al.*, 1991] Wayne W. Dai, Yoji Kajitani, and Yorihiko Hirata. Multiple bus networks based on block designs. In *1991 IEEE International Symposium on Circuits and Systems*, pages 1009–1012 vol. 2. IEEE, 1991.

[Dally, 1990] William J. Dally. Performance analysis of k-ary n-cube interconnection networks. In Winston and Shellard, editors, *Artificial Intelligence at MIT*, chapter 21. MIT Press, 1990.

[Dally, 1992] William J. Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2), Mar 1992.

[Deering, 1984] Micheal F. Deering. Hardware and software techniques for efficient AI. In *Proceedings of the National Conference on Artificial Intelligence*, pages 73–78. American Association for Artificial Intelligence, 1984.

[Ellis, 1992] Gerard Ellis. Efficient rertrieval from hierarchies of objects using lattice operations. Technical report, Key Centre for Software Technology, University of Brisbane, Brisbane, QLD 4072 Australia, 1992.

[Fahlman, 1979] Scott E. Fahlman. *Netl: A System For Representing And Using Real-World Knowledge.* MIT Press, 1979.

[Fahlman, 1980] Scott E. Fahlman. Design sketch for a million-element netl machine. In *The First Annual Conference on Artificial Intelligence*. AAAI, August 1980.

[Feigenbaum and McCorduck, 1983] Edward Feigenbaum and Pamela McCorduck. *The Fifth Generation: artificial intelligence and Japan's computer challenge to the world*. Addison-Wesley, Reading, MA, 1983.

[Felperin *et al.*, 1991] Sergio A. Felperin, Luis Gravano, Gustavo D. Pifarré, and Jorge L.C. Sanz. Routing techniques for massively parallel communication. *Proceedings of the IEEE*, 79(4):488–503, April 1991.

[Forgy, 1982] Charles L. Forgy. Rete: a fast algorithm for the many pattern/many object pattern problem. *AI Journal*, 1982.

[Forrest, 1991] Stephanie Forrest. Introduction. In Stephanie Forrest, editor, *Emergent Computation*. MIT Press, 1991.

[Glass and Ni,] Christopher J. Glass and Lionel M. Ni. The turn model for adaptive routing. In *19th International Symposium on Computer Architecture*, pages 278–287. Association for Computing Machinery.

[Gleick, 1988] James Gleick. *Chaos: making a new science*. Penguin Books, New York, 1988.

[Gould and Levinson, 1991] Jeffrey Gould and Robert Levinson. Method integration for experience-based learning. Technical Report CRL-91-27, Univerity of California, Santa Cruz, August 1991.

[Gupta and Forgy, 1989] A. Gupta and Charles L. Forgy. Static and run-time characteristics of OPS5 production systems. *Journal Of Parallel And Distributed Computing*, 7(1), August 1989.

[Gupta, 1984] A. Gupta. Implementing OPS5 production systems on DADO. In *International Conference on Parallel Processing*. IEEE, 1984.

[Hart *et al.*, 1968] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the determination of minimum cost paths. *IEEE Transactions on SSC*, 4:100–107, 1968.

[Hart *et al.*, 1972] P. E. Hart, N. J. Nilsson, and B. Raphael. Corrections to a formal basis for the determination of minimum cost paths. *SIGART Newsletter*, 37:28–29, 1972.

[Helmbold and McDowell, 1991] David P. Helmbold and Charlie E. McDowell. Computing reachable states of parallel programs. *SIGPLAN Notices*, (12):76–84, December 1991.

[Hendler, 1988] James A. Hendler. *Integrating Marker-Passing and Problem-Solving*. Lawrence Erlbaum, Hillsdale, NJ, 1988.

[Henessy and Patterson, 1989] John Henessy and David Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, 1989.

[Hillis, 1981] W. Daniel Hillis. The Connection Machine (computer architecture for the new wave). Technical Report AI-646, MIT, September 1981.

[Hillis, 1985] W. Daniel Hillis. *The Connection Machine*. MIT Press, 1985.

[Hoffmann, 1990] A.G. Hoffmann. On computational limits of neural network architectures. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, pages 818–825. IEEE Computer Society, 1990.

[Hughey and Roberts, 1993] Richard Hughey and James D. Roberts. Architectural concepts for a parallel associative processor. Technical Report to appear, University of California, Santa Cruz, 1993.

[Ianucci, 1988] Robert A. Ianucci. Toward a dataflow / von Neumann hybrid architecture. In *Computer Architecture Symposium*. IEEE, 1988.

[Jr., 1987] R. H. Halstead Jr. Overview of Concert Multilisp: a multiprocessor symbolic computing system. *Computer Architecture News*, 15(1):5–14, March 1987.

[Kanerva, 1988] Pentti Kanerva. *Sparse Distributed Memory.* MIT Press, Cambridge, MA, 1988.

[Kephart *et al.*, 1991] Jeffrey Kephart, Tad Hogg, and Bernardo Huberman. Collective behavior of predictive agents. In Stephanie Forrest, editor, *Emergent Computation.* MIT Press, 1991.

[Kermani and Kleinrock, 1979] Parviz Kermani and Leonard Kleinrock. Virtual cut-through: a new computer communications switching technique. *Computer Networks*, 3:267–286, 1979.

[Konstantinidou, 1990] Smaragada Konstantinidou. Adaptive, minimal routing in hypercubes. In William J. Dally, editor, *6th MIT Conference on Advanced Research in VLSI*, pages 139–153, 1990.

[Kumar and Pfister, 1986] Manoj Kumar and Gregory F. Pfister. The onset of hot spot contention. In *International Conference on Parallel Processing*, pages 28–34. IEEE, 1986.

[Leiserson, 1985] Charles L. Leiserson. Fat trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, 1985.

[Levinson *et al.*, 1992] Robert Levinson, Brian Beach, Richard Snyder, Tal Dayan, and Kirack Sohn. Adaptive-predictive game-playing programs. *Journal of Experimental and Theoretical Artificial Intelligence*, 4:315–337, 1992.

[Levinson, 1991] Robert Levinson. Pattern associativity and the retrieval of semantic networks. Technical Report CRL-91-14, Univerity of California, Santa Cruz, August 1991.

[Levinson, 1993] Robert A. Levinson, November 1993. Personal communication.

[Linder and Harden, 1991] Daniel H. Linder and Jim C. Harden. An adaptive and fault-tolerant wormhole routing strategy for k-ary n-cubes. *IEEE Transactions on Computers*, 40(1):2–12, January 1991.

[Marr, 1990] David Marr. Artificial intelligence: a personal view. In Derek Partridge and Yorick Wills, editors, *The foundations of artificial intelligence: a sourcebook.* Cambridge University Press, Cambridge, 1990.

[McCarthy and others, 1962] John McCarthy et al. *LISP 1.5 Programmer's Manual.* Massachusetts Institute of Technology, Cambridge, MA, 1962.

[McClelland *et al.*, 1986] James McClelland, David Rumelhart, et al. *Parallel Distributed Processing.* MIT Press, 1986.

[Minsky, 1986] Marvin Minsky. *The Society of Mind.* Simon and Schuster, New York, 1986.

[Miranker and Andrews, 1990] Daniel Miranker and Archie Andrews. On balanced synchronous parallel for AI. Institute of Electrical and Electronic Engineers, 1990.

[Moldovan *et al.*, 1990] D. Moldovan, W. Lee, et al. Parallel knowledge processing on SNAP. In *International Conference on Parallel Processing.* IEEE, 1990.

[Newell and Simon, 1972] Allen Newell and Herbert A. Simon. *Human problem solving.* Prentice-Hall, Englewood Cliffs, NJ, 1972.

[Nickolls, 1990] John R. Nickolls. The design of the Maspar MP-1: A cost effective massively parallel computer. In *COMPCON Spring '90*, pages 25–28. IEEE Computer Society, February 1990.

[Noshpitz, 1990] Claude Noshpitz. Some design issues for process migration systems. CE220 Class Project at University of California, Santa Cruz, May 1990.

[Noshpitz, 1991] Claude Noshpitz. An efficient implementation of Kanerva's Sparse Distributed Memory on the Connection Machine. Independent Study at University of California, Santa Cruz, Mar 1991.

[Omohundro, 1990] Steven M. Omohundro. Geometric learning algorithms. *Physica D*, 42(1–3):307–321, June 1990.

[Parkinson and Litt, 1990] Dennis Parkinson and John Litt, editors. *Massively Parallel Computing with the DAP*. The MIT Press, 1990.

[Patel, 1981] Janak H. Patel. Performance of processor-memory interconnections for multiprocessors. *IEEE Transactions on Computers*, C-30(10):771–780, October 1981.

[Quillian, 1968] M. Ross Quillian. Semantic memory. In Marvin Minsky, editor, *Semantic Information Processing*, chapter 6. MIT Press, 1968.

[Rich and Knight, 1991] Elaine Rich and Kevin Knight. *Artificial Intelligence*. McGraw-Hill, second edition, 1991.

[Roberts, 1990] J. Donald Roberts. Proximity content-addressable memory: An efficient extension to k-nearest neighbors search. Technical Report UCSC-CRL-90-45, University of California, Santa Cruz, 1990.

[Seitz and others, 1985] Charles L. Seitz et al. The hypercube communications chip. Technical report, Department of Computer Science, California Institute of Technology, 1985.

[Shapiro, 1990] Stuart C. Shapiro, editor. *Encyclopedia of Artificial Intelligence*. John Wiley and Sons, New York, 1990.

[Shaw, 1985] David Elliot Shaw. NON-VON's applicability to three ai task areas. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 61–72. American Association for Artificial Intelligence, 1985.

[Singer, 1990] Alexander Singer. Implementations of artifial neural networks on the Connection Machine. *Parallel Computing*, 14, 1990.

[Sowa, 1992] John F. Sowa. Conceptual graphs as a universal knowledge representation. *Computers and Mathematics with Applications*, 23(2-5), January-March 1992.

[Stanfill and Kahle, 1986] Craig Stanfill and Brewster Kahle. Parallel free-text search on the Connection Machine system. *CACM*, 29(12), December 1986.

[Stolfo, 1984] Salvatore J. Stolfo. Five parallel algorithms for production system execution on the DADO machine. In *Proceedings of the National COnference on Artificial Intelligence*, pages 300–307. American Association for Artificial Intelligence, 1984.

[Sutton, 1987] Richard S. Sutton. Learning to predict by the methods of temporal differences. Technical Report TR87-509.1, GTE, 1987.

[Tanenbaum, 1981] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[Thi, 1990] Thinking Machines Corporation. *Connection Machine User's Guide*, 1990.

[Ullmann, 1976] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the Association for Computing Machinery*, 23(1):31–42, January 1976.

[Wah and Li, 1989] Benjamin Wah and Guo J. Li. A survey on the design of multiprocessing systems for artificial intelligence applications. *IEEE Transactions on Systems, Man, and Cybernetics*, 19(4), August 1989.

[Webster, 1986] Daniel Webster. Webster's 7th dictionary: Online edition, 1986.

[Willet and Wilson, 1991] Peter Willet and Terence Wilson. Atom-by-atom searching using massive parallelism. Implementation of the Ullman subgraph isomorphism algorithm on the Distributed Array Processor. *Journal of Chemical Information and Computer Sciences*, 31(2):225–233, 1991.

[Williams and Kelley, 1990] Thomas Williams and Colin Kelley. *GNUPLOT: an Interactive Plotting Program*, 1990.

[Winograd, 1972] Terry Winograd. *Understanding Natural Language*. Academic Press, New York, 1972.