CCLISP™ on the iPSC™ Concurrent Computer

David Billstrom and Joseph Brandenburg
Intel Scientific Computers
Beaverton, Oregon, 97006 USA

John Teeter
Gold Hill Computers
Cambridge, Massachusetts, 02139 USA

Abstract

Concurrent Common LISP™ (CCLISP) is the LISP environment for the iPSC™ system, the Intel Personal SuperComputer. CCLISP adds message-passing communication and other constructs to the Common LISP environment on each processor node. The iPSC system is configured with Intel 80286 processor nodes, in systems ranging from 8 to 128 nodes. Performance on a per node basis roughly equivalent to AI workstation LISP performance is discussed, as are the implementation details of the CCLISP language constructs.

1 Introduction

Concurrent Common LISP (CCLISP) is a LISP environment extended for concurrency for the Intel Personal SuperComputer (iPSC) System. This software environment enables the researcher to implement concurrent symbolic programs on the iPSC System in a familiar language: Common LISP. The iPSC System is based on the *hypercube* interconnect topology pioneered by architects at California Institute of Technology [Seitz, 85]. The CCLISP environment is LISP listners at each processing node, communicating with each other via *message streams.*

The iPSC System, first available in 1985, utilizes VLSI technology in each of the processing nodes. Processing nodes consist of an Intel 80286 processor, 512 Kbytes random access main memory, and ethernet-based communication processors. Processing nodes are packaged in a standalone cabinet, along with optional memory expansion cards and vector processing cards. Additional memory enables 4.5 Mbytes of memory per node, and vector processing nodes offer better than 6 MFLOPS numeric performance per node. An Intermediate Host, currently a 286-based multiuser UNIX-based system, serves as network gateway, system administration console, and disk file system.

The availability of a concurrent LISP for a concurrent computer is attractive because existing artificial intelligence tools and applications may be ported from conventional computers and workstations. Several applications have already been moved to CCLISP, demonstrating substantial speedup by the use of many processors executing concurrently.

The speedup demonstrated by these first applications provides the motivation for using concurrent computers for symbolic problems. Many symbolic problems are too large for currently available uniprocessor computers to solve in a reasonable time, or at all [Stolfo *et al.* 83], [Hillyer and Shaw, 86]. Computers such as the iPSC system are used to develop algorithms, which in turn will be used to implement applications on future very large scale concurrent computers.

1.1 Architecture

The Intel concurrent computer is based upon a message-passing architecture. Asynchronous processor nodes execute their own programs, sharing data by passing messages. Although several alternative architectures provide ways of sharing data between processes, most are disadvantaged as the collection of processing nodes increases to hundreds or thousands of nodes. Since future delivery systems are planned to utilize such large numbers of nodes, the architecture of the iPSC system easily accommodates thousands of nodes.

Message passing avoids shared memory architectural solutions, which require expensive and complex data buses or switches for large numbers of nodes [Lee, 85], [Pfister, 85]. Data is exchanged between processors *not* by accessing common memory, with semaphores or monitors for synchronization, but by requesting and sending data objects among the processors via messages. The architecture was attractive because of the low component cost, and also because the system scales to large sizes. The message-passing architecture is also attractive because it translates relatively easily into language semantics and software protocols. Parallel architectures were built upon the pretext of message-passing, such as ACTORs from MIT, even before such architectures existed in hardware. ACTOR languages have been implemented on top of workstations, maintaining message passing [Agha and Hewitt, 85]. (The MIT Artificial Intelligence Lab is in progress bringing ACTORs to the iPSC system).

2. Constructs

Each processing node has a complete and separate Common LISP environment, with interpreter, editor, compiler, and debugging facilities. The programmer can open a window to any Concurrent Common LISP environment, on any node, from a workstation. File access is provided from each node CCLISP environment to connected workstations and to the system manager. Currently, the programmer edits and prototypes LISP code on the workstation, and then moves the source code down to a CCLISP node on the cube for testing, debugging, and compilation. Both compiled and interpreted code may be moved to other nodes as desired.

2.1 Message Passing

As with other languages for the iPSC system (C and

FORTRAN), the hypercube network is not directly visible to the programmer. Instead, the system provides a completely connected graph of processors. Resident on each processor node is a lightweight operating system, called *NX* (Node eXecutive) with two kinds of services: multitasking and message communications. Multiple UNIX-like processes may execute, passing messages between themselves and processes on other nodes. Communication services appear to the programmer as system calls such as *recv, send, recvw,* and *sendw* . The programmer is responsible for assembling messages in local memory, and specifying the message buffer, buffer size, type of message, and target processor id when calling the system service. In CCLISP, rather than the original C, the message is somewhat simpler; an example of a message would be:

```
(defstruct node-message
  connection       ; fixnum, NX channel id
  host-addr        ; fixnum, src/dest node
  correspondent-id ; fixnum, process on host-addr
  type             ; fixnum, (always 0 for now)
  buffer           ; simple-vector with fill ptr
```

Host-addr, correspondent-id, and *buffer* are filled with information during the receipt of a message, with host-addr indicating the source node of the message; correspondent-id indicating the name of the sender of the message; and buffer filled with the incoming information. An example of a receive follows:

```
(sys:recv node-message)
```

The buffer is a simple LISP array, containing either fixnum or chars. Messages passed at this level are compatible with C and FORTRAN processes, and their message-passing routines, so this is a method of implementing so-called *hybrid* applications. These are applications with mixed processor nodes: LISP on extended memory nodes, C on standard nodes, and FORTRAN on vector processing nodes. For instance, LISP applications with graphical output have been implemented by passing messages to C processes outside the iPSC system on connected workstations, utilizing the existing C graphic libraries on those workstations [Brandenburg, 86].

2.2 Node Streams

Above the transport-level message passing services, each node has the ability to communicate with any of the other nodes through a facility called *node streams*. Node streams are similar to Common LISP *I/O streams*. A stream can be established between any two LISP processes whether on the same node, different nodes, or on the remotely connected AI workstations. A wide array of LISP functions enable the programmer to send small or large packets of data to other processes, over a node stream (See Table 1). Since node streams are similar to I/O streams, many Common LISP functions operate on both I/O streams and node streams.

```
(let (stream (make-node-stream 0 :a-test
                :direction :io))
  (print 'hello-world stream)
  (finish-output stream)
  (close stream))
```

This code establishes a stream from the node where the code is executed to the node specified as a parameter, node 0. Nodes on the iPSC are numbered from 0. As many as 64 nodes can be loaded with LISP. The parameter *:a-test* is the name of the established stream; and *:direction* specifies the nature of the movement: input, output, or input/output. The name of the stream can be specified by the companion make-node-stream function on the opposing node, and the names are checked for equality. In this example a string is printed on node 0.

The node message stream is a powerful construct because of the similarity with I/O streams of Common LISP. Although it is premature for standards in concurrent languages, it is encouraging that such a concurrent construct could be added to the language, while approximating existing constructs such as I/O streams.

The CCLISP node stream is also notable because it is the first higher-level abstraction for communication on the iPSC system. Previous implementations for assembler, C, and FORTRAN languages exclusively utilized libraries of communication services, providing transport-level functions. Each of these calls required parameters containing not only node number and message, but message buffer length, processor id, and message type. Sufficient for the style of scientific

:read-char.....	Inputs next character from stream, waits if none
:read-line.....	Inputs next line, as delimited
:unread-char.....	Places character back onto the front of the stream
:read-char-no-hang.....	Inputs next character from the stream
:peek.....	Inputs next character without removing it
:listen.....	Returns t if a character is available
:fill-array.....	Places next n characters available into array
:write-char.....	Outputs character into stream
:write-line.....	Outputs string to the stream
:dump-array.....	Outputs the contents of array to the stream
:finish-output.....	Attempts to insure that all output is complete
:flush-output.....	Initiates the send of all internally buffered data
:clear-output.....	Aborts outstanding output operations in progress
:name.....	Returns the name of stream
:host.....	Returns the name of the node to which connected
:element-type.....	Returns the type of the stream
:close.....	Closes the stream
:direction.....	Returns the direction of the stream
:messages-sent.....	Returns the number of messages sent
:messages-received.....	Returns the number of messages received
:which-operations.....	Returns list of operations supported by the stream
:close-all-node-streams.	Closes all of the streams

Table 1: The functions available for use with node streams.

programming in C and FORTRAN, the message stream construct offers a level of abstraction in communication not previously seen on the iPSC system. Further, the message streams are used to communicate with processes not within the hypercube concurrent computer, such as LISP environments on network connected AI workstations.

2.3 FASL Streams

FASL streams are a special version of node streams allowing the transfer of CCLISP objects between nodes. This communication is speedier than regular node streams, at some sacrifice of functionality -- the functions supplied are enumerated in Table 2. The intent is to supply the programmer with a construct to move larger structures and compiled objects more rapidly. Compiled objects cannot be moved on regular node streams; all of the formats and protocols compatible with the CCLISP compiler may be moved by Fasl stream. In this example, the Fasl stream is used to move a compiled object:

```
(let (fstream (open-fasl-node-stream 0 :fasl-test
                :direction))
   (dump-object myfun fstream)
   (close fstream))
```

The ability to move compiled objects will be key to developing load balancing schemes dependent upon code movement.

:peek.....	Inputs next byte without removing it from stream
:listen.....	Returns t if byte available
:read-object.....	Returns next object available
:dump-object.....	Outputs the specified object to the stream
:dump-fasl-operator.....	Outputs the specified fasl-operator to the stream
:close.....	Closes the stream
:host.....	Returns the node number to which it is connected
:name.....	Returns the name used when stream made
:which-operations.....	Returns a list of the operations supported

Table 2: The functions operating on FASL streams.

2.4 Remote Evaluation

In addition to message streams, there is a powerful construct called *remote evaluation*. Remote evaluation offers the programmer a way to pass a Common LISP expression from one LISP environment to another LISP environment for evaluation. The programmer specifies a Common LISP form, and a target node number, and chooses either a synchronous or asynchronous remote evaluation of the form. The effect of the remote evaluation is to interrupt the target node (or connected AI workstation) and cause a read-eval-print loop to execute on the passed form.

In the case of synchronous execution, called *eval-remotely*, the sending process blocks and waits for the results of the evaluation before continuing to execute it's own program code. Consider the example of a simulation application: the basic "cause and effect" paradigm may be divided across processor nodes easily.

```
(if (cause)
   (eval-remotely 3 `(effect ,myparameter)))
```

Here, a cause results in an effect on a different node, node 3 in this case. There is no user code on the target node needed to "listen" for an activate request. However, the programmer must remember that the LISP form will be executed in the target environment, and there is no automatic provision for maintaining synchronous environments.

In the case of asynchronous execution, the sending process does not block and wait, but continues execution of its own code, while the specified destination process evaluates the passed expression. If the sending process needs the return value, the target node can use a message node-stream, or remote evaluate the value back to the original sending process. This *simple-eval-remotely* is the more commonly used construct, since it follows the paradigm of asynchronous processing, necessary for maximum utilization of a concurrent computer: processors do not stand idle dependent upon other processors.

```
(dotimes (node max-node)
   (simple-eval-remotely (1+ node) '(myfunction)))
```

In this example, the previously specified function *myfunction* is executed on each of the processor nodes in the system, in a serial manner. Since no value is returned, simple-eval-remotely is utilized for the side-effects it causes in the target node.

Interrupts from multiple *simple-eval-remotely* calls at the same target node can undesirably disrupt the execution of code, so an evaluation construct *without-interrupts* is available to turn interrupts off. Used carefully, for short periods of time, this allows multiple sources of interrupts to occur within one environment. For instance, the P and V primitives of semaphores could be implemented [Dijkstra, 68].

Also, a broadcast of remote evaluations is offered by the construct *do-on-all-nodes*. This macro replaces the code in the previous example:

```
(dotimes (node max-node)
   (simple-eval-remotely (1+ node)'(myfunction)))
```

to

```
(do-on-all-nodes '(myfunction))
```

Do-on-all-nodes follows a ring architecture on the iPSC system, an artifact of the system scheme for numbering nodes. A spanning tree algorithm could also be utilized by the user to broadcast to every node in the system in Log N iterations of remote evaluation [Brandenburg and Scott, 86]. A common use of *do-on-all-nodes* is the need to load the same CCLISP executable program on each node of the system, in order to solve a large concurrent problem:

```
(do-on-all-nodes `(load "my-file"))
```

Both forms of remote evaluation can be used to pass Common LISP forms outside the iPSC system to connected (and supported) AI workstations. This construct enables the iPSC system to be used as an remote evaluation server in a network of AI workstations -- an attractive scenario because a

current LISP application would remain on the workstation, particularly user interface portions, and the compute intensive portions of the application would be moved to the iPSC system. Tasks would be assigned from the workstation or multiple workstations to the iPSC system via remote evaluation. While in development, the entire application could be prototyped on the workstation, separately from iPSC code development, and then later linked with the insertion of the remote evaluation function call.

3 Development Environment

The intent of the development environment is to offer the user flexibility: choosing dynamically during the development cycle between a familiar, connected workstation and the concurrent LISP environment.

The model of user interaction with CCLISP consists of three parts: virtual terminals, keyboards, and file I/O. Every CCLISP node is connected to a virtual terminal, and the user can switch the physical terminal dynamically from window to window. The user attaches the physical keyboard to one of the virtual terminals, and may also switch that connection dynamically. The intent is to give the programmer instant access to any node of the iPSC system, and while independent processes execute on various nodes, virtual terminals remain connected in order to capture any output sent to the screen. The virtual terminal and keyboard user interface may be connected to a variety of physical devices, including connected AI workstations.

Each CCLISP node environment is also connected, via Common LISP I/O streams, to disk file services on the Intermediate Host, as well as connected AI workstations. All of the Common LISP I/O stream functions are available. Some support for the specific file system on the Intermediate Host (a UNIX file system), such as *cd* (change directory), is available from CCLISP. Other support for the Intermediate Host also exists, such as a single key escape to the operating system.

4 Implementation

The original CCLISP software was based upon Gold Hill Computer's 286 personal computer LISP product, GCLISP 286 Developer™. Offering a subset of full Common LISP, this interpreter with compiler provided strong performance from the original iPSC 286 node processor. The GCLISP environment is a subset of full Common LISP, with mark and sweep garbage collection, reasonable performance, and a large installed user base. The message-passing constructs already available in the iPSC node operating system were interfaced to CCLISP, and node streams and remote evaluation were built on the message-passing.

The CCLISP environment on each node, including the compiler, uses about 1.7 Mbytes of the 4.5 Mbytes available on the node. The user may elect to not load the compiler on every node of the iPSC, thus preserving an extra 0.6 Mbyte of memory for the user's own code and data.

The CCLISP node streams were added to LISP by exploiting the transport-level services already provided by the node processor operating system. The FASL Stream implementation was based on work at Carnegie Mellon for fast file formats in SPICE LISP. Briefly, at each end of the stream a simple stack machine is established. Byte operators are transmitted from the emitting end, along with data, and then interpreted at the receiving end and assembled into objects. The implementation code is available to users of the CCLISP system.

Remote evaluation was then implemented with FASL streams, and by an *eval server*, resident on each node LISP environment (as well as on connected AI workstations). The servers allow for the asynchronous handling of remote evaluation requests in the target environment. The evaluation of the requested form occurs in the current stack group, and environment, of the target node. Care must be taken to insure that node-specific CCLISP environments are maintained in a consistent manner. This includes package considerations as well as stack-group management.

Support for AI workstations for the user interface, file i/o services, and remote evaluation required compatible lower level services, such as TCP/IP ethernet connections. Special eval servers for each of these workstation environments, along with file servers and user interface connections, were developed. Each follows, or will follow, a public protocol for such support jointly developed by Intel, Gold Hill, and early users of the CCLISP/iPSC system [Intel, 87].

5 Performance

Concurrent computers demand at least two steps to measuring performance: first, processor node performance, and then applications demonstrating aggregate performance, the effect of all of the processing nodes. And, because benchmarks do not always consume the dynamic memory required of real-life applications, the total memory available per node is an important secondary component.

Performance of sequential LISP is popularly compared by use of Gabriel's LISP Benchmarks [Gabriel, 85]. Using a simple average of (most of) the Gabriel Benchmarks, the performance of the original CCLISP on a single node is almost equivalent to a low-end AI workstation, such as the Xerox 1108 Dandelion, as illustrated in the chart below.

Gabriel Benchmarks	CCLISP on a single iPSC node (compiled)	Xerox 1108 Dandelion	Symbolics 3600
stak	12.78	4.66	2.58
tak	4.21	1.67	0.60
ctak	8.91	63.20	7.65
takl	91.73	14.00	6.44
data deriv	26.16	33.30	5.24
derivative	26.70	23.80	5.12
destructive	9.45	17.58	3.03
div2 iterative	12.06	23.80	1.85
div2 recursive	18.39	24.80	2.89
boyer	67.49	74.60	11.89
browse	401.40	174.00	30.80
triangle	1034.34	856.00	151.70
traverse init	49.25	48.00	8.62
Average	135.61	104.57	18.34

Measuring the aggregate performance of a concurrent computer is more difficult. Primarily the problem is one of

size: large computers require large problems, especially in light of constant overhead. Only three of the Gabriel benchmarks entice the effort of parallelization. The longest running Gabriel Benchmark is the Triangle game, which consumes 14.44 seconds of cpu time on a CRAY-XMP, and 151.7 seconds on a Symbolics 3600. On a 16 node iPSC system, the benchmark is completed in 69.8 seconds, demonstrating 14.8 speedup for 16 processors. The benchmark completes in 37.5 seconds, for 27.6 speedup on a 32 node processor system. Perhaps more interesting than numerical results from recent timings of simple, small benchmarks are the approaches used to "parallelize" these benchmarks. The following sections each describe the changes made to run the simple benchmarks concurrently.

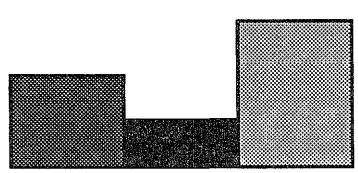
5.1 Gabriel Triangle

The benchmark finds all solutions to the "triangle game." The game consists of a triangular board with fifteen holes; a peg is placed in every hole except the middle. The player makes a move by jumping over a peg into a vacant hole and removing the jumped peg, as in checkers. The object of the game is to remove all of the pegs but one,. There are many possible sequences of moves, but only 1,550 sequences result in a single remaining peg. The Gabriel version of the algorithm finds 775 solutions; the other 775 solutions are symetrically identical (only one of the two initial moves is taken by the original benchmark algorithm).

The general problem is represented as a tree of possible moves; each node of the tree represents a decision about the next possible move. The problem heap technique [Moller-Nielsen and Straunstrup, 85],[Brandenburg, 86] was used as the basic distribution algorithm. This method uses a single node as a manager; the manager assigns each of the other nodes subgraphs from the search tree. The manager node solves the tree to the fourth level of 120 leaves. It then distributes each of the 120 subgraphs to the remaining fifteen worker nodes, assigning a subgraph to each worker node as the worker node becomes available from solving a subgraph.

Each node solves the subgraph using the traditional sequential algorithm, and then reports back to the manager the results, and requests another subgraph. The manager continues to distribute subgraphs and receive results until results for all subgraphs have been received from worker nodes. This exhaustive search incurs little overhead from communication between nodes, since nodes only communicate when they have completed computing each of 120 tasks. The algorithm is also attractive because it retains the basic search algorithm of the original benchmark, and thus minimizes the programming effort to implement the concurrent version.

Triangle Gabriel Benchmark

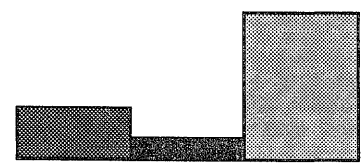


	iPSC Single node	iPSC 16 nodes	iPSC 32 nodes	Sym-bolics 3600
T (seconds)	1,023.2	69.8	37.5	151.70
Speedup		14.8	27.6	

5.2 Gabriel Browse

The Browse benchmark is a simple database search, similar to many AI matching problems. The database itself is an artificial representation of a real LISP database, consisting of very small objects. To convert the problem to a concurrent computer, the simple approach of dividing the database into 16 portions was used. Each CCLISP node carries a duplicate copy of the benchmark code, and its' own 1/16 portion of the database. There is no communication between processes, because the iterative nature of the benchmark requires none. And the results, compared with single node performance on the same problem, indicate superlinear speedup. This extra efficient use of 16 processors is not due to the architecture of the system, or even the problem, but the decreased load on the processor for garbage collection and other system resources in the LISP. As the data objects were reduced in volume to 1/16 the original size, system resource overhead was reduced accordingly.

Browse Gabriel Benchmark



	iPSC Single node	iPSC 16 nodes	iPSC 32 nodes	Sym-bolics 3600
T (seconds)	401.4	15.0	15.0	30.8
Speedup		26.8	26.8	
Pre-load T	398.5	9.4	5.2	
Speedup		42.4	76.6	

The Browse results clearly illustrate the difficulty of measuring symbolic computational performance on a large concurrent computer with small problems: the speedup for 32 nodes is 26.5, and for 16 nodes is 26.6. This lack of improvement with additional computational resources is due to the dominant factor of file access. This can be illustrated even further by a simple tuning of the code. By pre-loading the basic functions required in the benchmark -- lowering the non-computational overhead -- speedup can be improved to 76 for 32 nodes, and 42.6 for 16 nodes.

5.3 Gabriel Puzzle

The Puzzle benchmark was not successful demonstrating substantial speedup; the parallel version (with 16 processing nodes) found the solution only twice as fast as the sequential version. This is due to Puzzle seeking only one solution in the search tree, rather than all of the solutions -- as Triangle requires. When only one solution is sought, the decomposition of the problem into segments for each node of a concurrent computer becomes much more difficult. Depending upon where in the search tree the single solution may be found, the relative efficiency of a depth-first or breadth-first search varies greatly. In the case of the Puzzle benchmark, the solution was found among the first branches of the tree, down nine levels.

Single solution search tree algorithms for concurrent computers remains an important research issue.

6 Conclusions

The constructs of CCLISP were designed to allow relatively easy conversion of sequential LISP applications to a concurrent computer. As well as being conceptually simple but powerful, the constraints are sufficient to have allowed, in most cases, the first versions of applications to limp along in parallel within a week of starting the conversion. Indeed, since CCLISP has been in customer's hands, two software environments have been developed [Gasser *et al.*, 86], [Gasser and Braganza, 87], [Blanks, 86], others are on the way, and demonstration applications have been shown [Brandenburg, 86], [Intel, 86], [Gasser *et al.*, 87a], [Gasser *et al.*, 87b], [Yeung, 86]. All of the work thus far was converted from AI workstations such as the TI Explorer™ and the Symbolics 3600™. A handful of concurrent applications, including a community of expert systems, were running within a few months.

The performance of the iPSC system with CCLISP is wholly dependent upon algorithm choice and programmer effort, but the initial indications are very encouraging, with speedups of 14.8 for 16 processor systems on common symbolic processing problems of search. Node performance of CCLISP is respectable, almost equivalent to AI workstations such as the XEROX 1108 Dandelion.

The iPSC System with CCLISP has been available since September, 1986.

References

[Agha and Hewitt, 85] G. Agha & C. Hewitt. *Concurrent Programming Using ACTORS: Exploiting Large-Scale Parallelism.* MIT Press, AI Memo No. 865, October 1985.

[Blanks, 86] M. Blanks. Concurrent Cooperating Knowledge Bases. Presented at Aerospace Applications of Artificial Intelligence. Dayton, OH. October, 1986.

[Brandenburg, 86] J. Brandenburg. *A Concurrent Symbolic Program with Dynamic Load Balancing.* To appear in Proceedings of the Second Conference on Hypercube Multiprocessors. Oakridge, TN. September, 1986.

[Brandenburg and Scott, 86] J. Brandenburg and D. Scott. *Embeddings of Communication Trees and Grids into Hypercubes,* Intel Scientific Computers Document: Technical Report No.1, 1986.

[Dijkstra, 68] E.W. Dijkstra. *Cooperating Sequential Processes.* Programming Languages. (F. Genuys, editor), Academic Press (1968).

[Gabriel, 85] R. Gabriel. Performance and Evaluation of LISP Systems. MIT Press, 1985.

[Gasser and Braganza, 87] L. Gasser and C. Braganza. *MACE Multi-Agent Computing Environment, Version 6.0.* Technical Report CRI 87-16. Distributed Artificial Intelligence Group, CS Dept, University of Southern California. March, 1987.

[Gasser *et al.*, 86] L. Gasser, C. Braganza, N. Herman, and L. Liu. *MACE Multi-Agent Computing Environment, Reference Manual, Version 5.0.* Distributed Artificial Intelligence Group, CS Dept, University of Southern California. July, 1986.

[Gasser *et al.*, 87a] L. Gasser, C. Braganza, and N. Herman. *MACE, A Flexible Testbed for Distributed AI Research.* To appear in Distributed Artificial Intelligence, H. Hugns, Ed. Pitman, 1987.

[Gasser *et al.*, 87b] L. Gasser, C. Braganza, and N. Herman. *Implementing Distributed AI Systems Using MACE.* To appear in Proceedings of the Third IEEE Conference on AI Applications. Orlando, FA. February, 1987.

[Hillyer and Shaw, 86] Hillyer and Shaw. *Execution of OPS5 Production Systems on a Massively Parallel Machine.* Journal of Parallel and Distributed Computing, Vol.3,No.2, June 1986. pp. 236-268.

[Intel, 86] *A Preliminary Naval Battle Management Simulation.* Intel Scientific Computers Document: Artificial Intelligence Note 116. July, 1986.

[Intel, 87] *iPSC CCLISP host interface protocols.* Intel Scientific Computers Document, February, 1987.

[Lee, 85] R. Lee. *On "Hot Spot" Contention.* Computer Architecture News, Vol 13, No. 5, December 1985.

[Moller-Nielsen and Straunstrup, 85] P. Moller-Nielsen. and J. Straunstrup. *Problem Heap: A Paradigm for Multiprocessor Algorithms.* Aarhus University Technical Report DK-8000. Denmark. 1985.

[Pfister and Norton, 85] G.F. Pfister & V.A. Norton. *"Hot Spots" Contention and Combining in Multistage Interconnection Networks.* IEEE Transactions on Computers. Vol. C-34, No. 10, October 1985, pp. 943-948.

[Seitz, 85] C. Seitz. *The Cosmic Cube.* Communications of the ACM. 28-1 (1985), pp. 22-23.

[Stolfo *et al.*, 83] Stolfo, Miranker, and Shaw. *Architecture and Applications of DADO: A Large-scale Parallel Computer for AI.* Proceedings of the Eighth International Joint Conference on Artificial Intelligence, August, 1983. pp. 850-854.

[Yeung, 86] D. Yeung. *Using Contract Net on the iPSC.* Distributed Artificial Intelligence Group Research Note 20, CS Dept, University of Southern California. July, 1986.