Intel iPSC

4422 10:30 Friday

INTEL SCIENTIFIC COMPUTERS
INTEL INTERNATIONAL LTD.

Pipers Way, Swindon SN3 1RJ
Telephone: (0793) 696000 Telex: 444447/8

21st August 1986

Professor R Ibbett
Department of Computer Science
University of Edinburgh
The Kings Buildings
Mayfield Road
EDINBURGH
EH9 3JZ

Dear Professor Ibbett

I am writing to formally introduce you to the Intel iPSC range of concurrent and vector-concurrent supercomputers. These machines comprise the most widely used parallel processing system commercially supported, with almost 40 installations throughout the USA and Europe, and a user group with 150 members.

In their various forms these machines support applications in the fields of mathematics, computational chemistry and physics, and symbolic processing, computer science, geophysical sciences, meteorolgy etc. The following list of tools are currently available or in development, sponsored by Intel:-

TYPE	NAME	AVAILABILITY
Concurrent LISP	CCLISP (Compiler/Interpreter)	Now
Concurrent Prolog	FCP	Experimental version Now Commercial version 1987
Parallel Debugger	PDB	December 1986
C Compiler	C	Now
Fortran 77	RM Fortran	Now
Vectoriser	VAST-2	January 1987
Simulator for Unix m/cs	iPSC-SIM	Now
Simulator for VMS based VAX's	iPSC-SIM	September 1986
Dense matrix solver	LINPACK	Now
Eigenvalves/eigenvectors	EISPACK	Now
Sparse matrix solver	SPARSCUBE	December 1986
Extended matrix/vector	Extended BLAS	Early 1987
Differential equation Solver	FISHCUBE	Early 1987
Scientific Utilities	NAG	December 1986
Fortran Paralleliser	FPP	1987/1988
Distributed Ada	ADA	Late 1987
Fluid Dynamics	Lax-Wendroff	December 1986
Peptide Energy	ECEPP83	Now
Molecular Modelling	Guassian 86	October 1986
Molecular Modelling	MM2	Mid 1987
Molecular Modelling	MOPAC	Mid 1987
Image Synthesis	Genisys	December 1986
Oil Resivoir Modelling	VIP	Phased, September 1986 - Mid 1987
Semiconductor Modelling	Pisces	Early 1987

In addition to our commitment to ensure exceptional applications and programming support we recognise that price/performance is also important. We currently offer machines for numerically intensive applications at under £500/MFLOP. The 16 node iPSC-VX/d4, offering peak 32-bit performance of 320MFLOPS (106MFLOPS in 64-bit precision) is available until the end of 1986 to educational establishments for £132,000 (ex VAT) including 24MBytes memory, all system software, Fortran, Vectoriser and C. The iPSC-d4M comprising 16 scalar nodes, 72MBytes memory, Fortran, C and CCLISP for symbolic applications is similarly available for £88,500. Both systems include a Unix based Cube Manager with backing store and industry standard Multibus 1 for communication or perpheral devices. We would also be pleased to quote for larger or combined systems (some vector nodes, some large memory AI nodes) where required.

Should you wish to evaluate this machine in more detail, please do not hesitate to contact me on 0793 696578 which is my direct line.

Yours sincerely

David Moody
European Sales and Support Manager

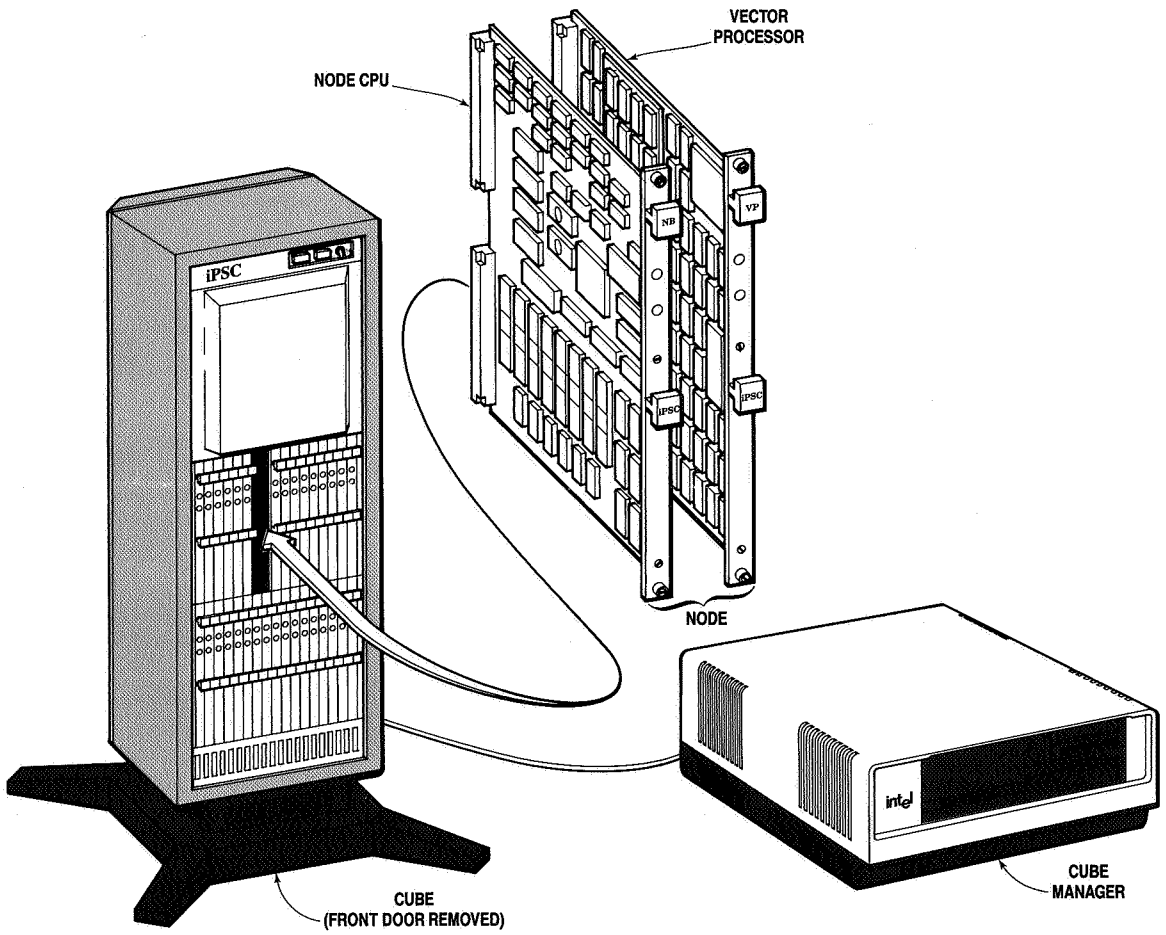
The iPSC-VX Vector Concurrent Supercomputer

The iPSC™-VX (vector extension) is a vector concurrent computer system, an enhanced member of the iPSC family of "personal supercomputers." The iPSC-VX offers true supercomputer performance for the individual scientist or research team, and provides the tools for developing efficient, high-performance applications in a large-scale parallel computing environment.

The cost of owning and maintaining a conventional supercomputer has made high-end scientific computing inaccessible to all but a privileged few. The iPSC-VX changes this. By using low cost VLSI to harness the complementary technologies of concurrent and vector processing, the iPSC-VX sets a new price-performance standard. The result is supercomputer performance at the price of a supermini.

The iPSC-VX family builds upon the basic architecture of the iPSC concurrent computer. By coupling a high-performance vector processor to each of the iPSC processing nodes, the vector enhancement results in dramatically improved computational performance for both vector and scalar operations. Optimized to meet the mathematical requirements of scientific computation, the iPSC-VX family is ideal for such applications as circuit simulation, structural analysis, fluid dynamics, and oil reservoir modeling.

An iPSC-VX system consists of 16, 32, or 64 computational nodes. Each node consists of an independent microcomputer with its own CPU, communications control, local memory, and dedicated vector processor. In keeping with the basic iPSC architecture, the processing nodes in a system are interconnected using a hypercube topology where connected nodes are supported with reliable point-to-point message delivery service. The multiple nodes which comprise a system are collectively known as the "Cube" and may be housed in one, two, or four computational units. The Cube Manager, an Intel System 310 supermicrocomputer, provides a gateway to the system and serves as a convenient software development station.



iPSC-VX System

Order Number: 280311-001

intel®

iPSC-VX FEATURES

Vector Concurrent Architecture	Uses low-cost VLSI to capture the benefits of both vector and concurrent processing to ensure the lowest possible cost per calculation.
Supercomputer Performance	A high-performance vector processor is tightly coupled to each of the system's computational nodes, boosting (64-bit) peak performance to 424 MFLOPS on a 64-node iPSC-VX/d6.
Large Memory	1.5 MBytes per processing node meets the memory requirements for efficient computation. Total system capacity ranges from 24 MBytes to 96 MBytes, depending on system size.
Field Upgradable and Expandable	A 16-node iPSC-VX is field upgradable to 32 or 64 nodes. Standard iPSC systems can also be upgraded to any of the iPSC-VX system configurations.
UNIX-Based Development Environment	An enhanced System 310 Cube Manager with its XENIX 3.0 operating system is the primary host for supporting system diagnostics and program development. Its standard configuration includes 3 MBytes of system RAM, a 140 MByte hard disk, a floppy disk, and a 45 MByte cartridge tape drive.
Professional FORTRAN	Opimizing FORTRAN compiler from Ryan McFarland offers complete GSA certified implementation of the FORTRAN-77 standard plus popular VAX and VS extensions.
Vector Programming Support	Standard FORTRAN programs can be automatically converted to vector routines using the VXP vectorizing preprocessor. Explicit vector programming is also supported through a powerful library of matrix, vector and scalar math functions.
Network Support	Intel's MULTIBUS®-based System 310 provides a convenient gateway to users of the Cube from other systems and networks via TCP/IP protocols.
Performance Monitoring Display	Convenient front-panel monitor can be used to display processing, communications, and computational activities for each node. Provides the user with instant feedback on system runtime behavior, load balancing, and fault diagnosis.
Lab Environmental Design	Compact packaging and a quiet air-cooled design allows convenient placement in laboratory environments.
Low Mean Time-To-Repair	Systems are provided with spare vector and/or node processor boards to reduce mean time-to-repair.

SYSTEM SPECIFICATIONS

Cube	iPSC-VX/d4	iPSC-VX/d5	iPSC-VX/d6
Number of Nodes	16	32	64
Peak System Performance (MFLOPS)			
64-bit	106	212	424
32-bit	320	640	1280
Total System Memory (MBytes)	24	48	96

For more information call or write:
Intel Scientific Computers
15201 NW Greenbrier Parkway
Beaverton, Oregon 97006
(503) 629-7629

SUPERCOMPUTER SYSTEMS DIVISION
INTEL CORPORATION (UK) LTD.

Pipers Way, Swindon SN3 1RJ, U.K.
Telephone: +44-793-696000 Telex: 444447/8
Direct Fax: +44-793-488353
Fax: +44-793-641440

An update on the Intel iPSC®/860 Supercomputer

A year has passed since we announced the Intel iPSC/860 Supercomputer. Production shipments began in the Spring of 1990, the first European machines being installed in Britain and Germany. Subsequently customers in France, Italy and Sweden were added to the list of over 45 iPSC/860 sites.

So who uses the iPSC/860 and are they able to achieve performance levels which are truly of "supercomputer" class?

- Oak Ridge National Laboratory have achieved 1792 MFLOPS sustained performance on the KKR-CPA code in their superconductivity research (compared with 203 MFLOPS on a single processor Cray YMP). [1]

- Using our own "Prosolver-DES" out-of-core solver (previously called LOOCS) a major US corporation is routinely handling systems of equations of order 25000 and achieving 1612 MFLOPS sustained rate on just 64 processors - even while paging blocks from the concurrent I/O system! [2]

- Researchers at the University of Indiana have achieved over 500 MFLOPS on a 32 processor iPSC/860 system in their study of Quantum Chromodynamics, one of the Grand Challenges of computational science. [3]

Other users including Boeing Computer Services, SERC Daresbury Laboratory, Technical University of Munich, Honeywell, Prudential-Bache, San Diego Supercomputer Centre, ARCO, NASA, GSF, University of Lund and IMFL Marseille all have their own stories to tell.

We believe that installations like these have proved the iPSC/860 to be worthy of the name "Supercomputer". However, we are not resting on our laurels. The success of the iPSC/860 has helped us to secure a contract to provide the world's most powerful computer to fourteen prominent research institutions forming the Concurrent Supercomputing Consortium. This machine, containing over 500 i860™ processors will provide a peak performance of 32 GFLOPS. Whilst this system is a limited production model, it will enable us to refine our next generation of supercomputers.

The success of Intel Scientific Computers has led to the achievement of full divisional status within Intel and henceforth we will be known as the Supercomputer Systems Division of Intel Corporation. An important benefit of this will be the immediate strengthening of our European team so that we are better able to support you in reaching your goals in supercomputing.

Yours sincerely

David Moody/Mary Herron
European Managers

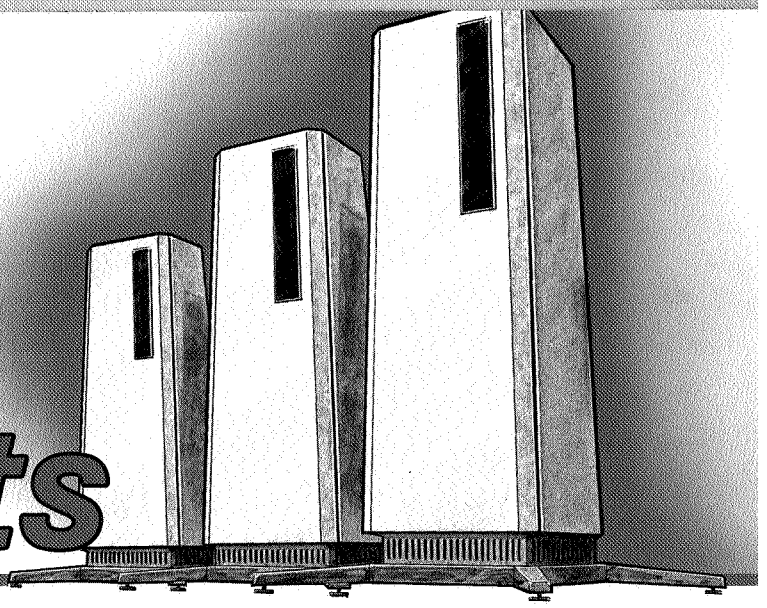
January 1991

References:

[1] MT Heath, GA Geist, JB Drake, "Early Experience with the Intel iPSC/860 at Oak Ridge National Laboratory", Draft Performance Report, Oak Ridge National Laboratory, Oak Ridge, TN, - 1990.

[2] DS Scott, E Castro-Leon, EJ Kushner, "Solving very large dense systems of Linear Equations on the iPSC/860" Proceedings of the DMCC5, Supercomputer Systems Division, Intel Corporation, Beaverton, OR.

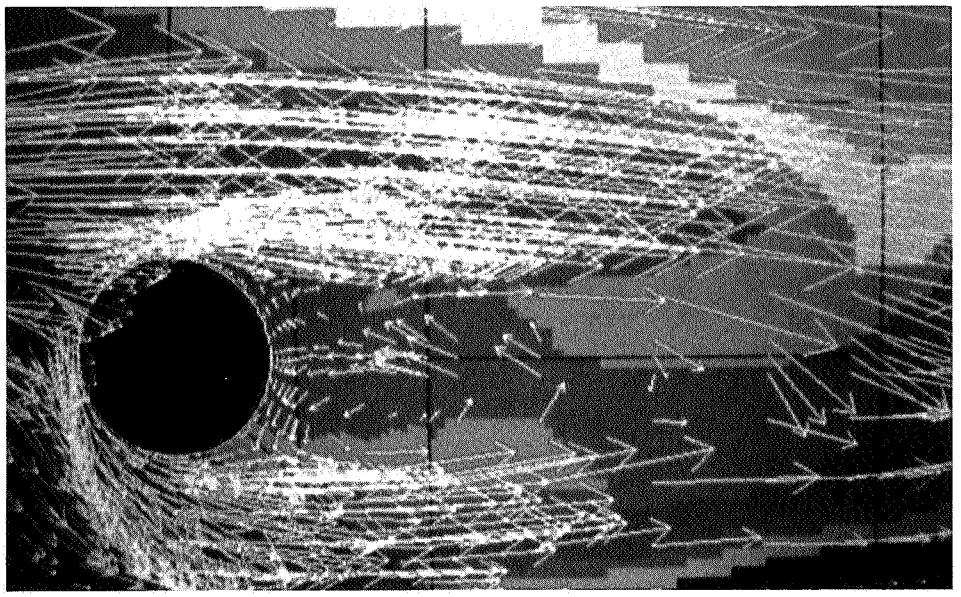
[3] S Gottleib, A Krasnitz of Indiana University, D Toussaint of University of Arizona, "Quantum Chromodynamics Performance on the iPSC/860", Preliminary Report, Oct 1990



iSCurrents

Published by Intel Scientific Computers *Fall/Winter 1987*

NEKTON Announced for iPSC®



This display shows the velocity and temperature fields in the vicinity of an Eddy-Promoter Heat Exchanger for an unsteady flow, as calculated by the NEKTON fluid dynamics and heat transfer numerical simulation package.

The first full-scale production software package for the iPSC® has just been announced by Nektonics Inc., of Bedford, MA. The *NEKTON* package is a state-of-the-art fluid dynamics and heat transfer simulation code that includes the most recent advances in numerical methods and algorithmic techniques. It features 2D and 3D modeling capabilities, high computational efficiency, and accurate modeling further into transitional regions than competitive packages.

The NEKTON package is already implemented and widely used on a variety of computational platforms, ranging from MicroVAX and Sun workstations to Cray supercomputers. The package has been used to study thermal management of electronic components, crystal growth, contaminant dispersion, and many other fluid flow and heat transfer processes.

Now, NEKTONICS has announced a beta release of NEKTON for the iPSC-VX Vector Concurrent Supercomputer, with production shipments to begin early in 1988. "The iPSC-VX architecture is ideally suited to the computational methods used by NEKTON," said Dr. Anthony Patera, chairman and co-founder of Nektonics. "The price-performance advantage of this system makes it possible to put a dedicated, supercomputer-class simulation engine into the hands of a small engineering team."

The first public announcement of the iPSC-VX version of NEKTON came at the August meeting of the American Society of Mechanical Engineers in New York City.

The performance of the iPSC-VX / NEKTON system makes it a highly cost-competitive alternative to traditional, sequential approaches. "NEKTON on the iPSC-VX system can dramatically improve engineering productivity," said Elliot Swan, Market Development Manager for Intel Scientific Computers. A NEKTON simulation that takes *four hours* on a VAX 8600 system takes just *three minutes* to complete on a comparably priced, 32-node iPSC-VX. "When you can do a half-day of simulation in the time it takes to get a cup of coffee, there will be significant changes in the way engineers do their work," added Swan.

NEKTON performance on the iPSC-VX is roughly equivalent to a CRAY X-MP costing 10 times as much. "With this level of computing power available *locally* and *dedicated to an engineering team,* simulations can be more frequent, more finely tuned, and consequently of higher quality," continued Swan. "And with the iPSC-VX as a shared engineering resource, simulations formerly done on

continued on page 2

In this Issue of iSC

iSCurrents

Volume 2 Fall/Winter 1987

iSCurrents is published quarterly (Spring, Summer, Fall, Winter) by:

Intel Scientific Computers
15201 N.W. Greenbrier Parkway
Beaverton, OR 97006
503-629-7629

Nektonics, Inc.:
Planning for Concurrency

The primary goal of Nektonics, Inc. is to provide the engineering and scientific communities with *leading edge floware* — proven, cost-effective tools for solving a wide range of problems in fluid dynamics and heat transfer. NEKTON is Nektonics' first commercially available package. The company is already working on major enhancements for release early in 1988 (see related article), as well as customer support and training strategies.

Nektonics is one of an emerging class of software developers who recognize the importance and benefits of parallel computing. Because of the price/performance of parallel computers such as the iPSC® and iPSC®/2 systems, these software developers are targeting parallel architectures as the preferred computational platforms for the 1990s and beyond.

Nektonics, Inc. is headquartered in Bedford, MA, with offices in Cambridge, MA and Princeton, NJ. The company was founded in 1985 by a group of engineering and mathematics faculty from M.I.T. and Princeton University. The principal co-founders are Dr. Anthony Patera (M.I.T.) and Dr. Steven Orszag (Princeton), both leading researchers in computational fluid dynamics.

Today, Drs. Patera and Orszag serve as Chairman and Vice-Chairman of the Board of Nektonics, respectively, and continue to provide technical consultation as needed. Dr. Patera's research expertise is in heat transfer, spectral element methods, and parallel processing. Dr. Orszag is the founding principal investigator of the NSF-funded John von Neumann Supercomputer Center; his research expertise is in turbulence modeling, spectral methods, and supercomputing.

NEKTON Announced for iPSC®

continued from page 1

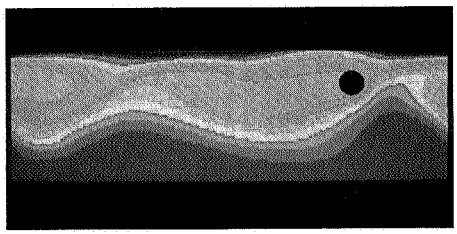
workstations can now be done on the hypercube, with greater speed and accuracy."

NEKTON is a new code, written in ANSI standard FORTRAN 77. While the computational approach is based on fundamental research supported by ONR, AFOSR, DARPA, and NSF, subsequent code development was funded by Nektonics.

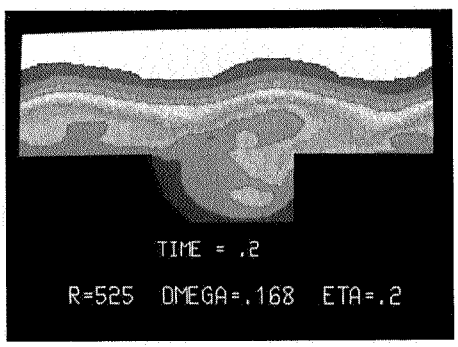
The base system was written for a virtual message-passing parallel processor with nested vectorization, making it an excellent fit for the iPSC-VX. Nektonics, working closely with iSC, has now tailored the code to take full advantage of the features of the iPSC. The customization included development of a specialized matrix multiply operation that achieves 8 to 15 MFLOPS (million floating point operations per second) per processor.

The NEKTON code solves the full incompressible, unsteady Navier-Stokes and energy equations. It is particularly well suited for detailed calculations of flows in the laminar and transitional regimes, for both 2D and 3D problems. The code is built around the *spectral element method* (see related article) and provides more accurate results than were previously attainable for transitional regions.

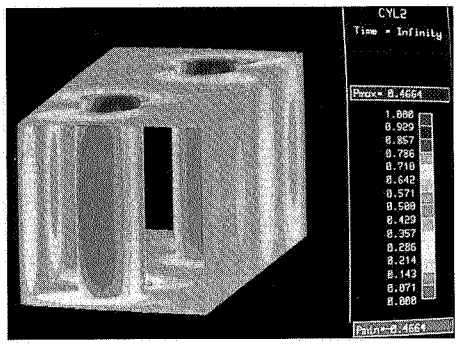
The package includes graphics oriented pre- and post-processors that run on a workstation attached to the iPSC. The pre-processor, PRENEK, uses a menu/mouse-driven user interface to simplify mesh generation and boundary condition specification. The post-processor, POSTNEK, uses menu-driven interactive graphics to assist in interpreting results. Full-color graphics enable clear presentation of flow, velocity, temperature, and pressure fields. The flow domain may be viewed at any angle; velocity, temperature, and pressure can be examined on arbitrary planes in the flow.



Isotherms for an Eddy-Promoter in an unsteady flow.



Isotherms for pulsed flow in a grooved channel.



Stokes flow past a staggered cylinder array in a three-dimensional duct (calculations performed on an iPSC-VX hypercube).

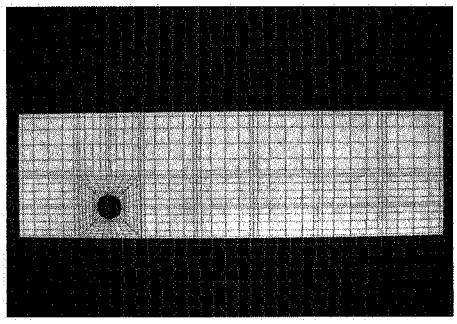
Two major enhancements are scheduled for release in early 1988: simulation of free surface phenomena, and a turbulence model developed by Dr. Steven Orszag of Princeton. Models of turbulent regions have not been commercially available heretofore. In addition, a release of NEKTON for the Intel iPSC®/2 Concurrent Supercomputer is expected early in 1988.

The NEKTON product license includes support by phone, mail, TELEX, and fax. More extensive support and consultation is available at additional cost. Additional information can be obtained from Nektonics, Inc. PO Box 22, Bedford, MA 01730, (617) 275-4011, or from Intel Scientific Computers.

The Spectral Element Method

A key factor in the accuracy and computational efficiency of the NEKTON package is a sophisticated mathematical technique known as the *spectral element method*. This method is a high-order, weighted residual finite element technique for numerical solution of time-dependent, incompressible fluid flow and heat transfer equations.

The spectral element method combines the geometric flexibility of finite element methods with the rapid convergence of spectral techniques. In a spectral element discretization, the computational domain is broken



Spectral element mesh for an Eddy-Promoter heat exchanger.

up into relatively few, large, spectral elements. Variables are approximated by Nth order tensor-product polynomial expansions, and convergence is achieved by increasing the order of the polynomial while keeping the number of elements fixed.

The spectral element method gives high accuracy with relatively few grid points by using high-order interpolation and by automatically clustering points near domain boundaries. Elements may be placed in regions of rapid variation to improve accuracy.

Solutions are achieved efficiently due to the relatively few degrees of freedom required, the use of tensor-product sum factorization, and the use of pre-conditioned iterative inversion procedures.

The iPSC®-VX release of NEKTON uses an innovative algorithm that maximizes the computation-to-communication ratio. Inasmuch, speedups are nearly linear with the number of nodes. With the Direct-Connect™ Routing scheme, the iPSC®/2 should provide even greater speed and efficiency.

Intel Demos Firsts in Parallel AI at IJCAI

Intel Scientific Computers demonstrated a number of firsts in parallel AI at the IJCAI (International Joint Conference on Artificial Intelligence) in Milan, Italy during the week of August 21 to 28. "It was an excellent conference," commented David Billstrom, Product Marketing Manager of iSC, "and we were pleased to show the AI community a number of first-ever parallel tools and demos." Intel teamed up with Artificial Intelligence Limited, an iSC distributor in England, to bring 14 software demonstrations to the show, in three overall categories.

For LISP programmers, Intel showed two new **programmer-oriented user interfaces: HYDRA** on a XEROX 1186 workstation, and **VIEWS** on a Sun-3 color workstation. Each iPSC node was viewed through a window using the standard windowing environments on the Sun and XEROX workstations. These were firsts in parallel AI, allowing programmers access to the CCLISP programming environment via familiar workstations, and drew praise from CCLISP users. HYDRA was developed by Artificial Intelligence Limited specifically to join the existing XEROX LISP and iPSC environments.

Another highlight of the CCLISP demonstrations was **NIRA (Northern Italian Renaissance Art)**, an art identification program. Using an Intel SugarCube™ with the XEROX 1186/HYDRA workstation interface, the user describes a painting via menus and pointers. NIRA uses a parallel inferencing algorithm on a distributed database to identify the era and artist, asking for additional information when necessary. Because of the small domain size, this program was an excellent fit for the four-node SugarCube, with very good speedups from parallel processing.

The second group of demonstrations centered around **SOPE (Systems Object Programming Environment)** from Advanced Decision Systems of Mountain View, CA. SOPE is a programming environment for object-oriented programming on a parallel computer. The primary demonstration was LOSP, a parallel deduction engine that uses a single proto-logical concept to execute deductive steps in highly parallel fashion. The LOSP program has excellent human interface features and runs on the Sun-3 workstation with an iPSC D4-MX system.

Flat Concurrent Prolog from the Weizmann Institute was shown in a compiled version, the first *compiled* language implementation for the concurrent logic community. The compiled FCP gives substantial speedups over the previous, interpreted versions of FCP. The demo included a concurrently executing polygon graphic modeling package implemented in FCP by Steve Taylor, Ph.D. student at the Weizmann Institute, Israel. The package showed a 14-fold speedup on a 16-node iPSC.

Intel and Artificial Intelligence Limited worked together with the Weizmann Institute in Israel and Advanced Decision Systems in California to bring hardware and software together and to integrate the systems at the show. In spite of 100° temperatures, a severe lightning storm, and heavy rains, all of the systems worked well.

"The show was an overwhelming success for parallel AI in Europe," continued Billstrom. "We were very impressed with the level of interest in parallel AI, and the questions we received showed a high level of understanding of the potential for concurrent computing in AI."

Conference Announcements

Conferences provide a gentle introduction to concurrent computing for the newcomer, and well-known rewards for the initiate. An additional benefit for all is that the following winter conferences are held in southern California:

• **Third Conference on Hypercube Concurrent Computers and Applications (HCCA3).** January 19-20, 1988. Co-sponsored by California Institute of Technology and Jet Propulsion Lab. To be held at the Pasadena Conference Center, Pasadena, CA. For information, contact the conference coordinator: Dr. Andrew Witkowski, Jet Propulsion Lab, MS 138-208, Pasadena, CA 91109, 818-354-2244
ARPAnet: andy@hamlet.caltech.edu.

• **AEROSPACE 88: AIAA 1988 Annual Meeting and International Aerospace Exhibit.** February 9-11, 1988. Sponsored by the American Institute of Aeronautics and Astronautics. To be held at the L.A. Airport Hotel, Los Angeles, CA. For information contact: AIAA, 370 L'Enfant Promenade, Washington, DC 20024.

Users Group Seeks Cubelib Submissions

The iPSC Users Group is actively encouraging submissions to Cubelib, the software library, said Victor Jackson, the administrator of Cubelib and an active Users Group member.

Cubelib provides a convenient way for iPSC users to share non-proprietary software, and to gain both feedback and recognition for their development efforts. The library includes tools and techniques to help shorten product development cycles and optimize cube use, as well as complete applications that can serve as *templates* for developing similar applications.

The following packages are currently available through Cubelib:

- Cube_use — utilities for monitoring use of the iPSC
- Eiscube — concurrent routines for symmetric eigenvalue/eigenvector computations
- Lincube — concurrent routines for solving systems of linear equations, including an efficient implementation of dense factor and solve routines
- GESBT — Generic Expert Systems Building Tool, for constructing expert systems in a concurrent environment
- Navier — a concurrent solution to the Navier-Stokes equations using an explicit Lax-Wendroff scheme
- Sparspak — concurrent routines for solving sparse systems of linear equations
- Suprenum — grid-oriented routines for problem partitioning
- Technotes — letters to customers, release notes, and technical notes published by iSC

Users can obtain information about the programs in Cubelib via electronic mail or conventional mail. By email, just send the message "send index"; addresses are:

...!intelisc!cubelib
for uucp users

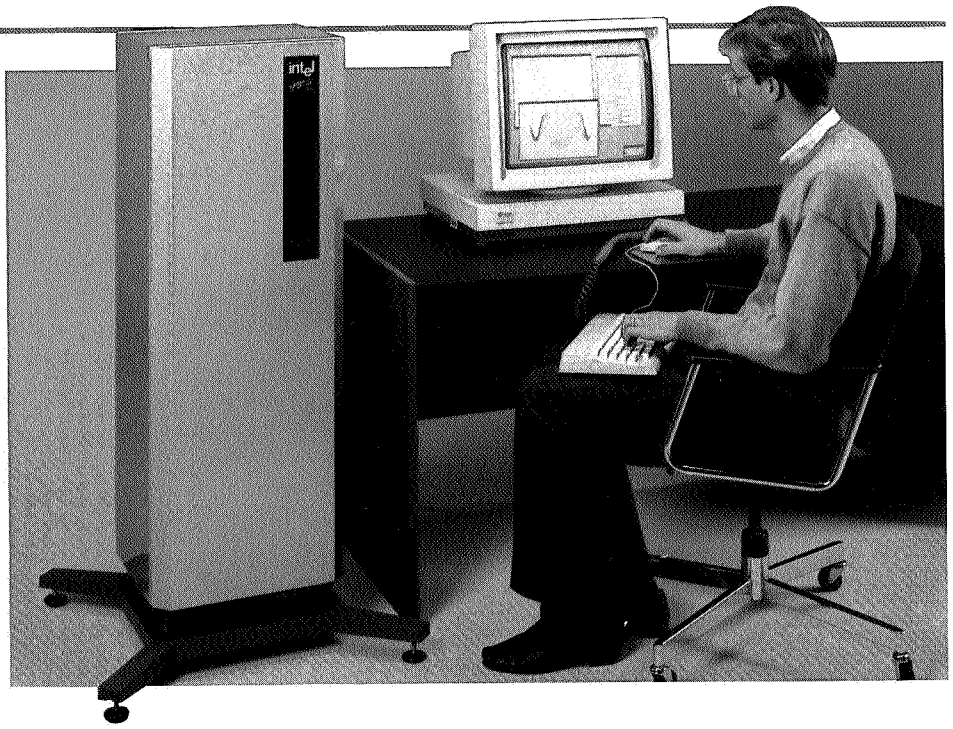
cubelib@isc.intel.com
for CSNET, Arpanet users

Send requests by conventional mail to the Users Software Librarian as listed below.

User submissions play a vital role in the success of the library, and must conform to certain minimal guidelines regarding generality, ease of use, robustness, and documentation.

For details on submission or current Cubelib contents, contact:

User Software Librarian
Intel Scientific Computers
15201 N.W. Greenbrier Parkway
Beaverton, OR 97005
503-629-7600



The iPSC®/2 Concurrent Supercomputer – Delivering on the Promise of Concurrent Supercomputing

"The iPSC®/2 Concurrent Supercomputer is going to change the way people think about supercomputing," said Charlie Bishop, iSC Marketing Manager, "just as the PC changed how we think about office computing. The performance of the iPSC/2 system initiates the era of *interactive* supercomputing, while the price makes supercomputing truly affordable."

The first-generation iPSC system successfully demonstrated the hypercube architecture as a viable approach to concurrent computing. Incorporating the knowledge and experience gained from the iPSC system, the 80386-based iPSC/2 system incorporates design innovations that substantially broaden the range of applications that are appropriate to concurrent machines. "We are seeing the 'first fruits' of these applications in high performance software packages such as the NEKTON fluid dynamics code," stated Bishop. "The iPSC/2 is the first true production code hypercube."

The iPSC/2 system consists of three major components:

- The *cube* — a system of 32 to 128 computational nodes, each built around Intel's 32-bit 80386 as a node processor, coupled with an 80387 floating point accelerator. Memory capacity is 1, 4, 8, or 16 MBytes per node.

Two numerical performance enhancements are available: the *SX Scalar Extension* triples the scalar performance of the basic iPSC/2 system, and the *VX Vector Extension* increases vector performance over that of the iPSC by an order of magnitude.

- The *Direct-Connect™ Routing Network* — connects all of the iPSC/2 processing nodes together. Although nodes are physically connected as a hypercube, Direct-Connect Routing offers performance that functionally connects every node to every other node.
- The *Concurrent Workbench™* — a systems programming support package that provides a variety of languages, a simulator, vector development tools, and a concurrent debugger.

The Concurrent Workbench software development environment is standard with

the iPSC/2 system, and is hosted on the System Resource Manager, accessible from a network of workstations such as the SUN-3. The System Resource Manager serves as the administrative console and a gateway to the cube for network users. The System Resource Manager is also an 80386-based system, and features 8 MBytes of memory, a 140 MByte hard disk, and an Ethernet connection.

The iPSC/2 system employs the same modular design and uses the same vector boards used in the first-generation iPSC. A principal objective in the iPSC/2 development effort was to provide an effective migration path for first generation iPSC users. Consequently, even though the hardware and system software have been substantially enhanced, software for the iPSC is upward compatible with the iPSC/2. This allows iPSC users to upgrade to a corresponding iPSC/2 system without losing any part of their software investment.

In addition, the iPSC/2 system offers major advances over the first-generation iPSC system, including:

- Increased node performance (4 to 5 times) with the 80386 node processor.
- Faster node-to-node communications (3 to 10 times) with a substantial reduction in multihop latency made possible with the iPSC/2's *Direct-Connect Routing* (see related article).
- Larger memory capacity, expandable from 1 Mbyte to 16 Mbytes per node.
- Significant improvements in the user development environment, and the benefits of a 32-bit system environment.

Performance

The combination of increased performance for each system node, the Direct-Connect Routing communications scheme, and larger memory capacity enables the iPSC/2 system to give *true supercomputer performance.* For example, a 32-node iPSC/2-VX machine performs a 2D FFT (Fast Fourier Transform) at 154 MFLOPS, 10 times faster than the first-generation iPSC. The 2D FFT represents the worst-case node-to-node communications in hypercube architectures. In addition, a 128-node iPSC/2 system executing the Gabriel Triangle symbolic benchmark for LISP systems tops the performance of the previous record holder, the CRAY 1S, by more than a factor of four.

Productivity and Ease of Use

In addition to the 80386 node processor, three factors contribute to the increased productivity of the iPSC/2 system: mainframe-quality languages and system support, the Concurrent Workbench system support package, and the Direct-Connect Routing communications scheme.

The move from a 16-bit to a 32-bit node architecture gives users access to a new class of production compilers and software tools. "People used to spend most of their time porting code to the limited, 16-bit environment of the original iPSC," stated David Billstrom, Product Marketing Manager. "Now *every node* of the iPSC/2 has the performance of a VAX 8600, *and* with a 32-bit software model. Users can spend their time solving problems, not porting code."

Another major factor in productivity and ease of use is the iPSC/2 system's Concurrent Workbench. Now, all of the software tools developed on the first iPSC system are gathered together in an integrated environment, and this environment appears to run on the user's Unix*-based workstation. Hosted on the System Resource Manager, the Concurrent Workbench allows the user to access, program, and control the iPSC/2 system in the more familiar and productive windowed environment of today's popular workstations.

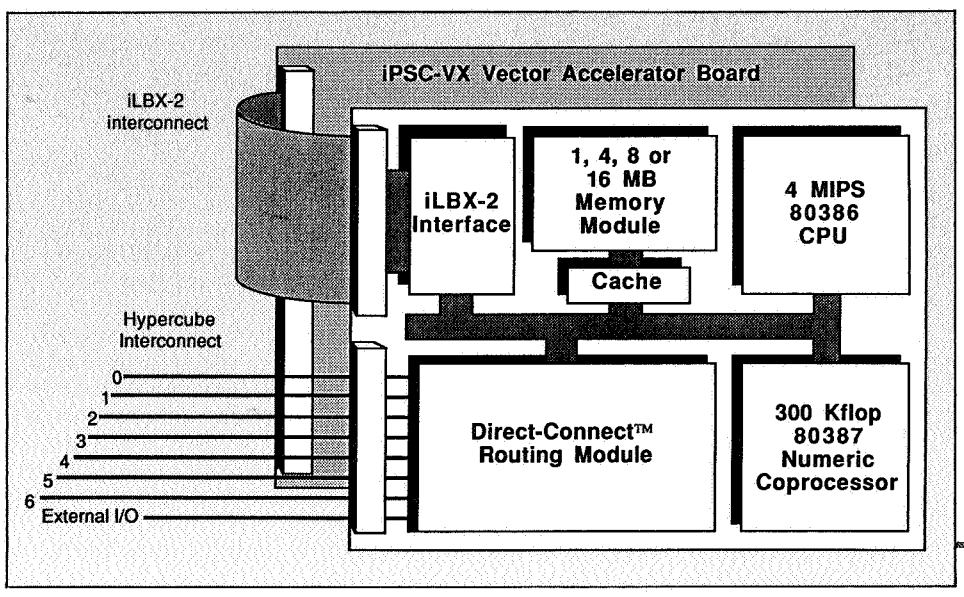
The programming of first generation hypercubes was characterized by an emphasis on problem decomposition to maximize "nearest neighbor" communications. Direct-Connect Routing enhances productivity by effectively removing these "nearest neighbor" constraints.

With Direct-Connect Routing, it is no longer necessary to precisely match problem and machine topologies to achieve high computational efficiencies (see related article). You can now program the hypercube as an ensemble of processors with an arbitrary communications network in which each node communicates efficiently with all other nodes.

Production Software

This fall, Intel and Nektonics, Inc. jointly announced the availability of the NEKTON fluid dynamics and heat transfer simulation code for the iPSC/2. NEKTON is the first production code for any large-scale parallel computer (see article, p. 1).

In addition, Intel plans to announce four other production-quality codes in the next few months: a molecular modeling package, an extruded materials modeling package, a VLSI device simulation code, and an event-based simulation code. These applications will bring the supercomputer performance of the iPSC/2 Concurrent Supercomputer to a new class of users, those who use computing resources as a standard productivity tool but do not have access to a traditional supercomputer.



Direct-Connect™ Routing Solves Node Communications Challenge

With the introduction of the iPSC®/2 Concurrent Supercomputer, Intel Scientific Computers has solved one of the major bottlenecks in hypercube computing, the problem of passing messages to distant nodes quickly and without degrading node processor performance.

"The benefits this brings are much more than just speed," stated David Billstrom, iSC Product Marketing Manager. "With Direct-Connect™ Routing, message passing times are essentially *uniform,* as though every node were *connected directly* to every other node. This feature is unique to the iPSC/2 system, and will change the way programmers use the hypercube architecture. It heralds a major change in hypercube technology."

As an example, a large system such as the 128-node iPSC/2 d7 system requires only about 10% longer to send a message between the most distant nodes than

continued on page 6

Direct-Connect™ Routing Solves Node Communications Challenge

continued from page 5

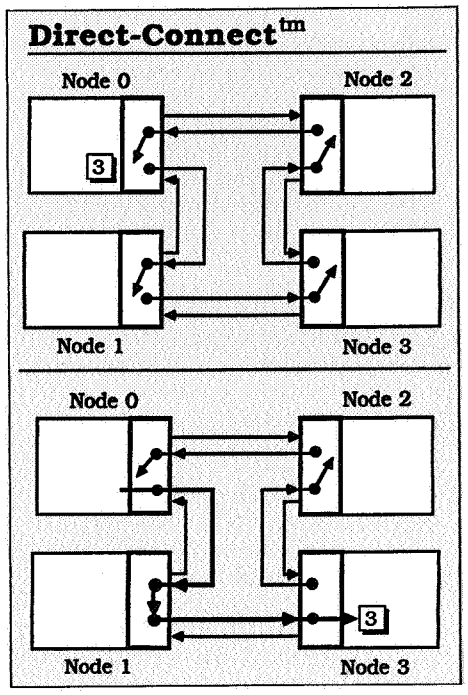
between "nearest neighbor" nodes. As a result, computational efficiency is essentially independent of the problem domain-to-machine topology mapping. This greatly simplifies problem decomposition and frees the programmer to concentrate on resolving other programming issues.

The Direct-Connect Routing scheme uses improvements in both hardware and software to shorten message-passing times. With Direct-Connect Routing, the earlier "store and forward" method used in the first-generation hypercubes is replaced by a hardware switching system, the *Direct-Connect Module*, on each node.

Each Direct-Connect Module (DCM) functions much like a telephone exchange with eight full-duplex channels. Seven of these channels are used for internode message passing, while the eighth channel provides connection to the System Resource Manager or I/O facilities. Because DCM channels are independent, up to seven messages can be passed simultaneously over each node, avoiding bottlenecks when several nodes must communicate at the same time.

The basic concepts for Direct-Connect Routing were developed at California Institute of Technology by Dr. Charles Seitz and Dr. Bill Daley, with research cosponsored by Intel Corporation and DARPA. In addition to uniform communications, DCR provides significantly improved node-to-node communications by reducing latency and increasing communication bandwidth. Latency — the time to send a zero-length message between nearest neighbors — is about 3 times faster than the original iPSC and an order of magnitude faster than many "real time" operating systems.

Experience has shown that the optimum performance from hypercube systems requires both small latency for short messages *and* high communications bandwidth for long messages. The Direct-Connect Routing scheme minimizes the time required to deliver short messages (less than 100 bytes), while allowing messages greater than 100 bytes to move at the full bandwidth of the network. Both types of messages benefit from this scheme, which first establishes a direct connection between the source and destination nodes, then streams the message through at essentially hardware data rates (see illustration and table).



Direct-Connect™ Routing uses a special algorithm for messages longer than 100 bytes. This algorithm first sends a header message to the destination node (Node 3). This header sets gates in each DCM module on the intermediate nodes, clearing a data path for the message. Once the destination node acknowledges receipt of the header, the message is streamed through at essentially hardware data rates, without "packetization," to the destination node.

Hop Length	Store & Forward Latency (μsec)	Direct-Connect Routing Latency	DCR Percentage	Latency Speedup
1	950	350	37%	2.7
2	1,300	352	27%	3.7
3	1,650	354	21%	4.7
4	2,000	356	18%	5.6
5	2,350	358	15%	6.6
6	2,700	360	13%	7.5
7	3,050	362	12%	8.4

This improvement is demonstrated by the iPSC/2 system's performance on the 2-dimensional Fast Fourier Transform (FFT). The FFT has a reputation for "worst case" communications on hypercubes, because all nodes need to trade intermediate results with all other nodes at roughly the same time. An iPSC/2 32-node system solves the 2D FFT at 154 MFLOPS, which is 10 times faster than the original iPSC. This speedup is due almost entirely to the increase in communications efficiency provided by Direct-Connect Routing.

The communications redesign also improves data rates between the System Resource Manager and the Cube, from about 80 KBytes/sec to a peak 2800 KBytes/sec, giving a system I/O speedup factor of 35 over the iPSC system.

"We are just beginning to understand the full benefit of Direct-Connect Routing and its impact on application performance," said Billstrom. "But the excitement of the future will be automated methods of decomposition and load balancing, and for these, Direct-Connect Routing just may be the enabling technology."

Users Group to Hold iPSC®/2 Workshop

The iPSC Users Group has scheduled a one-day users meeting and special iPSC®/2 system workshop for Monday, January 18, 1988 in Pasadena, CA. This places the workshop just before HCCA3, the third annual hypercube conference. Workshop arrangements will be convenient for HCCA3 attendees.

The workshop will include personalized, in-depth presentations, tutorials, and "inside information" to help users take full advantage of the new features of the iPSC/2 Concurrent Supercomputer. Topics for the workshop include new product features, staged enhancements to the iPSC/2 system, concurrent I/O solutions, benefits of the new user interface (with expanded front-end support), and iPSC compatibilty.

The iPSC Users Group provides a forum for sharing techniques, ideas, and tools among iPSC users, as well as an opportunity for input concerning new products and enhancements. Intel is working with the Users Group to stage the workshop and tutorial session. The Users Group, however, is an independent organization, with officers elected by group members.

Anyone with access (logon authority) to an iPSC system can join the Users Group, and there are no membership fees. For further information about the Users Group or the iPSC/2 workshop, contact Victor Jackson of iSC, 503-629-7704.

iSC Training Refocused for iPSC®/2 System

The iSC training program in programming concurrent computers is being updated and refocused to support the new iPSC®/2 Concurrent Supercomputer, said Victor Jackson, Senior Training Specialist for iSC. "The basic class has been modularized to provide individual emphases on numeric and vector computing," continued Jackson. "The class provides a solid introduction to the new iPSC/2 system, but users of the first-generation iPSC system will continue to find the class an excellent resource."

The basic sequence now begins with a three-day session, *Introduction to Numerical Concurrent Computing* that includes hardware and software overviews and basic concurrent programming techniques. This is followed by a two-day session, *Introduction to Vector Concurrent Computing,* that covers vector programming fundamentals for the iPSC/2 VX and use of the VAST-2 Fortran Vectorizer.

The iSC Training Center also offers *Introduction to Symbolic Concurrent Computing,* featuring the fundamentals of programming in Concurrent Common LISP. This workshop is offered alternate months. "We are very enthusiastic about the symbolic computing class," said Jackson. "With the price/performance and memory capacity of the iPSC/2 systems, we expect to see growing interest from the AI community." While familiarity with LISP programming is recommended, no background in concurrency is needed for the class.

Although you can come into either of the workshops "cold," a little advance preparation can make class time more productive, added Jackson.

"If you haven't programmed recently, it helps to review a bit, just to get the wheels turning again. Also, come with a specific problem to solve."

Fall Training Schedule

Introduction to Numeric Concurrent Computing
February 22-24
March 28-30
April 25-27

Introduction to Vector Concurrent Computing
February 25, 26
March 31, April 1
April 28, 29

Introduction to Symbolic Concurrent Computing
February 17-19
April 20-22

Concurrent Workbench™: Easing the Software Development Task

With 30 years of sequential codes for scientific and industrial applications, it's no surprise that software development is a major concern in concurrent computing. One of the prime objectives in the iPSC®/2 system design was to make software development as easy and productive as possible.

The result is the *Concurrent Workbench™,* a Unix*-based development environment that allows the user to access, program, and control the iPSC/2 system from his or her own workstation. This is a significant change from the earlier, first-generation systems, which required all development to be done on a XENIX-based Cube Manager.

Most Concurrent Workbench tools running on the System Resource Manager appear to run on the user's workstation. This means that the user can write, edit, and debug programs for the iPSC/2 system in an environment that is the most familiar, comfortable, and productive. The SUN-3 workstation from SUN Microsystems is the first workstation to be supported; other popular workstations will be supported in the near future.

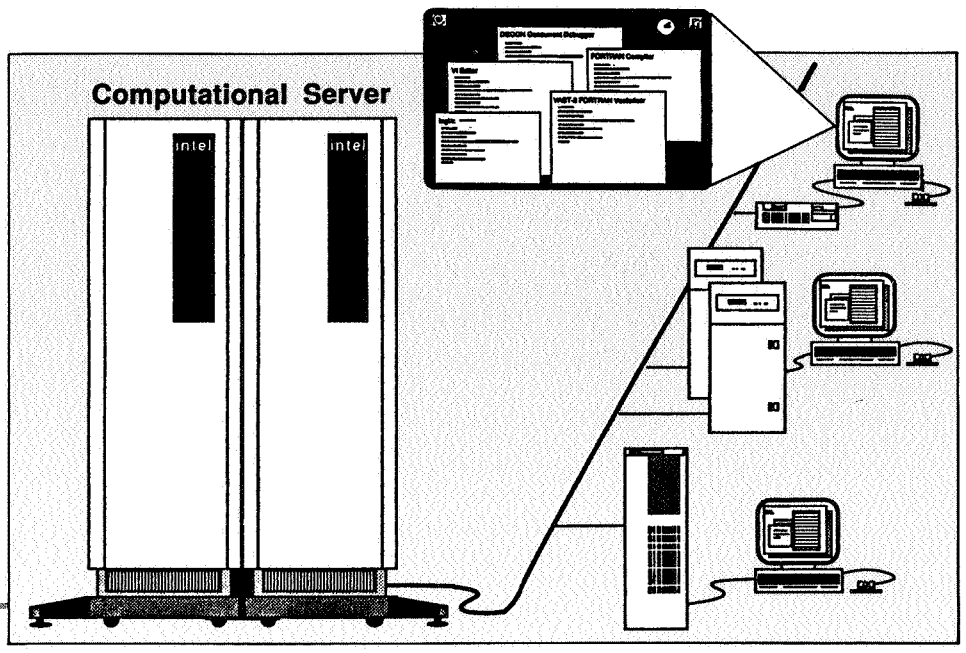
The Concurrent Workbench includes multiuser, network access to the iPSC/2 system. It supports *cube sharing,* which allows several users to partition the cube into functionally distinct subcubes and increases the versatility and productivity of the iPSC/2 system.

Based on AT&T Unix* V.3 on the System Resource Manager, the Concurrent Workbench provides a full set of software development tools. These tools include the iPSC/2 Simulator software package, C compiler, optimized FORTRAN 77 with extensions, and *DECON,* a high-level symbolic process debugger. DECON supports simultaneous debugging of node processes and internode communications for all nodes of the cube, and represents one of the most sophisticated debugging facilities available for concurrent systems.

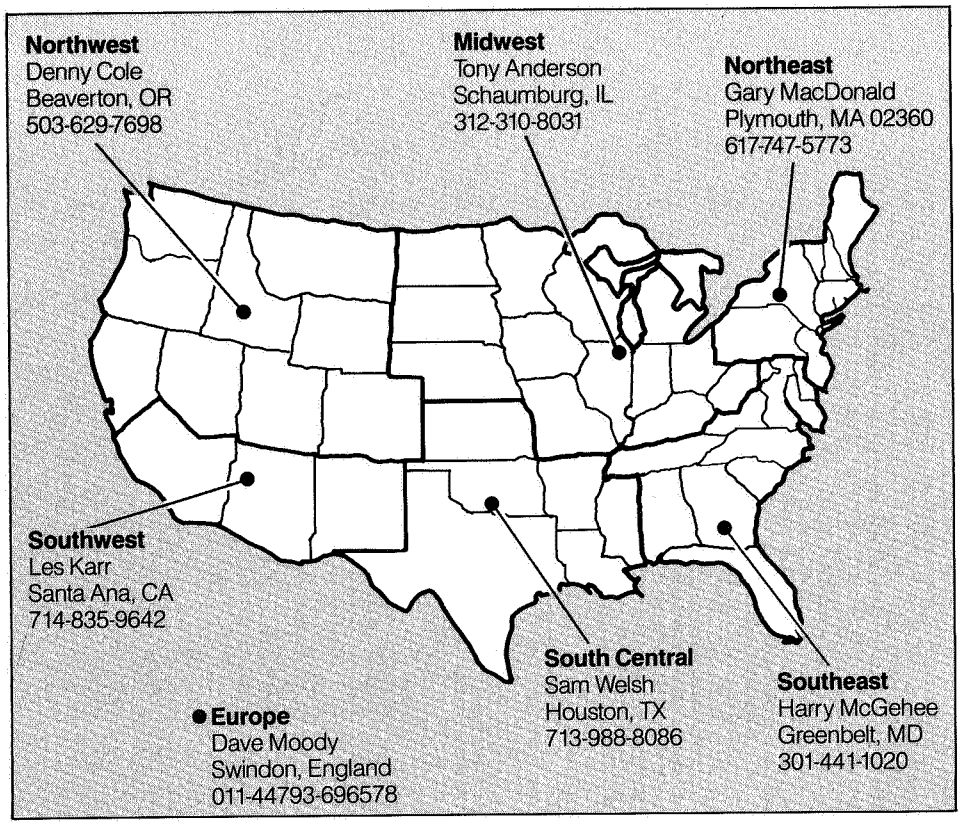
Other Concurrent Workbench tools are available at additional cost. For vector processing applications, the *VAST-2 FORTRAN Vectorizer* discovers sections of Fortran code that can be vectorized and translates these sections into vector commands that execute on the iPSC/2 VX machine. VAST-2 has been highly ranked among a group of supercomputer vectorizers by a national laboratory, and comes from Pacific Sierra Research, a leading supplier of vectorizing software for supercomputers.

For AI applications, a full implementation of *Concurrent Common LISP (CCLISP)* from Gold Hill Computers, Inc. is also available. LISP is the most popular AI language in the United States, and CCLISP is the first commercial LISP for a parallel computer. A single iPSC/2 node running CCLISP gives performance roughly equivalent to a Symbolics/LISP system. With the iPSC/2's memory structure, up to 128 nodes can be configured to run CCLISP, making the iPSC/2 one of the fastest AI systems available.

With the Concurrent Workbench™, the user can access, program, and control the cube from a convenient, windowed workstation environment. The SUN-3 workstation is the first to be supported; support for other popular workstations will be announced soon.



iSCurrents



For More Information...

"We work more like partners than vendors," says Charlie Bishop, Marketing Manager for Intel Scientific Computers. "Concurrent computing is still new to a lot of folks. So a lot of our work is simply technical consultation, helping people to understand the new technology, its benefits, and how to use it."

Each iSC Representative has several years of training and experience in the field, and each one provides a range of services, from informational seminars to post-sales support.

For more information about the iPSC®, iPSC®/2, or Sugarcube™ concurrent computing systems, contact one of these iSC Representatives.

Do you know anyone....

...who might be interested in receiving *iSCurrents*? If so, please contact:

iSCurrents
Intel Scientific Computers
15201 N.W. Greenbrier Parkway
Beaverton, OR 97006

Intel Scientific Computers
15201 N.W. Greenbrier Parkway
Beaverton, OR 97006

Address Correction Requested

INTEL SCIENTIFIC COMPUTERS
INTEL CORPORATION (UK) LTD.

Pipers Way, Swindon SN3 1RJ, U.K.
Telephone: +44-793-696000 Telex: 444447/8
Fax: +44-793-641440

At the beginning of 1988 we started shipments of the iPSC/2 concurrent supercomputer. During the year the number of systems delivered has almost equalled the total of iPSC/1 machines installed in the previous three years.

Here in Europe the customer base grew by over 250% allowing us to hold our first European Users Group meeting in Belgium during October (the next is to take place in France in October 1989, prior to the "1st European Workshop on Hypercube and Distributed Computers" organised by INRIA/IRISA).

Clearly, massively parallel computer systems have begun to receive acceptance in the industrial world, with many systems now being used in computationally intensive production applications. We believe that our recently announced concurrent input-output system (CIO) will have a similar impact in commercial database and transaction processing, as well as widening the scope of the iPSC/2 for scientific applications.

Our systems will next be displayed at "Supercomputing Europe 1989" in Utrecht from 21st - 23rd February 1989. I invite you to visit our booth for a close examination of the performance and applicability of the iPSC/2 for your own needs.

I would like to thank you for your interest in our products and extend the compliments of the season to you. Should you require any additional information on the iPSC/2 (or more copies of the year planner) please do not hesitate to contact Jean-Pierre Seingier (France), Peter Schuller (Germany) or myself.

Yours sincerely

David Moody
European Manager

INTEL SCIENTIFIC COMPUTERS
INTEL INTERNATIONAL LTD.

Pipers Way, Swindon SN3 1RJ, U.K.
Telephone: +44-793-696000 Telex: 444447/8

IMPORTANT UPDATE ON THE INTEL HYPERCUBE

Enclosed are details of major developments from Intel Scientific Computers in our family of general purpose concurrent and vector-concurrent supercomputers. This pack includes new product brochures, application briefs, technical notes and the latest edition of our newsletter "iSCurrents".

Highlights are :

0 Details of our second generation concurrent processor, the iPSC/2. Shipments of the iPSC/2 to being December 1987.

0 Major enhancements to the development environment.

- concurrent symbolic debugger
- remote hosting (supports full control of networked hypercube from other Unix-based systems and allows use of windows, graphics etc)
- cube sharing allows multi-user access to logical "sub-cubes"
- VAST-2 full-feature vectoriser for Fortran users on vector-concurrent systems.
- Support for hybrid systems comprising a mix of node types (ie. vector + A.I.)

0 Details of the entry level "SugarCube" concurrent workstation, available at prices ranging from only **$45,000** up to **$69,000** for the 80MFLOP (26MFLOPS at 64-bit precision) iSGR-VX/d2. These systems are delivered complete with software and may be used standalone or networked with other machines. "SugarCube" is 100% compatible with larger iPSC systems and incorporates an identical development environment.

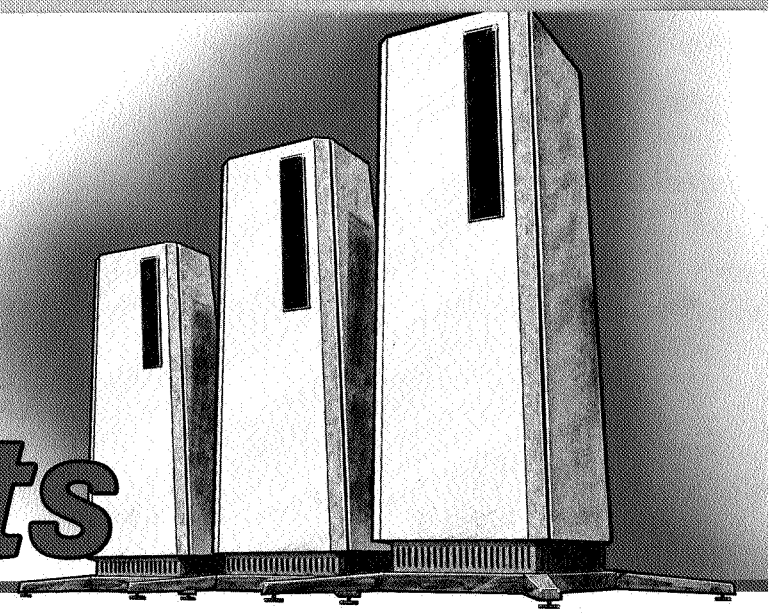
0 Benchmark results for the iPSC-VX (**3 times Cray-1S** performance on Monte Carlo simulation, **227 MFLOPS** on 32 nodes for seismic modelling, 2D FFTs resulting in worst case communications traffic, yet creditable performance).

0 The latest issue of our newsletter "iSCurrents".

These developments have further increased iSC's technological lead in the field of massively parallel computer systems. If you have a project which could benefit from the ease of use, advanced software features and superior price-performance profile of the iPSC family please call me.

Yours sincerely

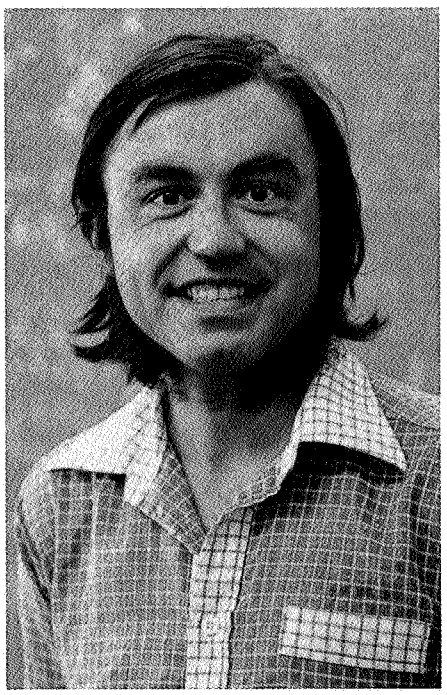
David Moody
European Manager



iSCurrents

Published by Intel Scientific Computers *Summer 1987*

Computational Science at Caltech: An Interview With Geoffrey Fox



Dr. Geoffrey Fox is Associate Provost for Computing at California Institute of Technology, and Director of the Caltech Concurrent Computation Program (C³P).

"We're at a watershed," said Dr. Geoffrey Fox, Associate Provost for Computing at California Institute of Technology and one of the men who has put hypercubes at the forefront of concurrent computing. "Although much theoretical *computer science* remains to be done, we now have the necessary tools to do *computational science* — that is, to apply real supercomputing power to solve current research problems in science and engineering."

Fox is one of the most active members of the concurrent computing community. His initial interest in the hypercube grew out of the need for more computing power to solve numerical problems in high-energy physics. Fox collaborated with Dr. Charles Seitz and others during development of the original hypercube design at Caltech, with research partially funded by Intel Corporation. Seitz designed and built the first hypercube (the "Cosmic Cube"), and Caltech/Jet Propulsion Lab have constructed two additional generations of the design.

The emphasis on "real computing for real science" sets the tone for work in the Caltech Concurrent Computation Program (C³P), which Fox directs. "We are refocusing on the uses of supercomputing and on developing software tools to support the full range of scientific research at Caltech. We have now about 40 separate codes running on the hypercube architecture."

C³P includes the efforts of about 15 people full time, and Fox has asked Paul Messina from Argonne National Labs to become Project Director for the ongoing work of C³P. Tools are under development to support all pure and applied research at Caltech, including:

- Applied Math and Computer Science: computer graphics, CAD, load balancing algorithms, mathematics and logic
- Biology: modeling of cortex and neural networks
- Chemistry and Chemical Engineering: protein dynamics, chemical reaction dynamics
- Engineering: plasma physics, finite element analysis, condensed matter simulations, vortical flows, image processing
- Geophysics; seismic waves, geodynamics, normal modes of earth
- Physics: fluid jets in astrophysics, high-energy physics, lattice gauge theory, molecular dynamics

Fox's own research interests remain strongly applied: high-energy physics, multiple target tracking, simulation of cortical and neural processes (a cortex simulation is now running at Caltech), load balancing algorithms, and others. He has also found time to coauthor *Solving Problems on Concurrent Processors* with several others at Caltech. The book, scheduled for publication this fall by Prentice-Hall, presents a general approach to concurrent computing and describes the CrOS III — or *crystalline* — implementation of that approach.

This emphasis on using supercomputing tools to solve applied problems in science and engineering is a key feature of C³P, according to Fox. "The success of this program will not be measured by the number of academic papers published, but by the quality of the science

(continued p.2)

In this Issue of iSC

iSCurrents

Volume 2 **Summer 1987**

iSCurrents is published quarterly (Spring, Summer, Fall, Winter) by:

Intel Scientific Computers
15201 N.W. Greenbrier Parkway
Beaverton, OR 97006
503-629-7629

Geoffrey Fox, (cont.)

that results from these projects, by the number and significance of scientific discoveries that result from making supercomputing available to the Caltech research community."

Fox also commented on the challenges that face supercomputing today. "There is a critical need for software, both tools and applications. The best thing for concurrent computing today would be for someone to dedicate $15 million a year *just to converting existing codes*. We have 30 years of scientific applications written for sequential machines. It's going to take time and money to bring those into the concurrent arena. It's not very glamorous work, but it is absolutely necessary if concurrent computing is to become commercially viable."

Summer and Fall Training Schedule

Programming Concurrent Computers

July 6-10
August 10-14
September 21-25
October 19-23

For additional information, contact Intel Scientific Computers, Training Center, 15201 N.W. Greenbrier Parkway, Beaverton, OR 97006, 503-629-6279.

Do you know anyone . . .

. . . who might be interested in receiving ***iSCurrents***. If so, please contact:

iSCurrents
Intel Scientific Computers
15201 N.W. Greenbrier Parkway
Beaverton, OR 97006

CrOS III — A High-Performance Communications System For Hypercubes

Simplicity and speed are the keys to CrOS III, the latest version of the *crystalline* node-to-node communications system for hypercubes. The crystalline system was developed by Dr. Geoffrey Fox and his associates at California Institute of Technology. "Call it stripped down or call it elegant," said Fox. "Either way, CrOS III provides the level of performance and portability that are essential to develop a broad range of applications software for the hypercube architecture."

CrOS III is a set of functions designed to optimize node-to-node message passing in a hypercube architecture. It trades some of the functionality of a full node executive (such as the iPSC system's NX) for increased speed and efficiency. CrOS III is available for iPSC systems and is applicable to a wide range of scientific applications. Support for CrOS III is available from Caltech and from ParaSoft, a newly formed Los Angeles company that provides commercial software for the iPSC system and other hypercube environments.

Adam Kolawa, a recent Ph.D. under Fox and now on the staff at Caltech, is one of the developers of CrOS III. "CrOS III is fully compatible with the iPSC system's NX programming environment," said Kolawa. "Any program that runs under NX can be easily modified to run with CrOS III under NX, with significant speedups. Software conversion is easy. For nearest neighbor messages, just change CALL names for message send and receive routines. For longer distance messages, the Crystal Router utility simplifies conversion."

The crystalline system was initially developed to match the problem topologies that Fox encountered in his research work in high-energy physics. "You gain the most from concurrency when the machine's interconnection topology matches the natural topology of the problem," said Fox. "Many problems in the physical sciences are characterized by 'nearest-neighbor' communications in a domain with regular geometric decomposition. Grid methods for boundary value problems are the classical example. But you see the same structure in image processing and other areas. The name *crystalline* came from this geometric regularity."

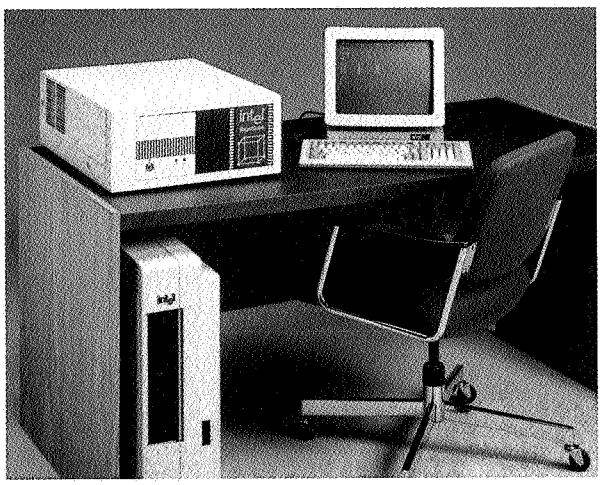
But crystalline has turned out to be much more general than first suspected, said Fox. "Crystalline works well for any problem that is 'loosely synchronous'. That is, there is a compute stage during which nodes compute independently and in parallel, then a communicate stage in which the nodes communicate results of the compute stage among themselves, then another compute stage, and so on. This includes geometrically regular problems such as grid-based numerical methods and image processing. But it also includes many simulation methods such as Kalman filters, used in multiple target tracking applications. In fact, probably 90% of the scientific applications at Caltech could run efficiently under crystalline."

This potential for a wide range of applications is another attractive feature of the crystalline system. Adam Kolawa, along with Jon Flower, John Salmon, and Marc Goroff, are developing additional support tools for CrOS applications. *CUBIX* allows a single version of source code to run on both a sequential machine and a single node of the hypercube, making program development easier and faster. *PLOTIX* provides vector graphics output to both soft and hard copy devices, including standard Tektronix terminals and Hewlett Packard plotters.

Kolawa, Flower, Salmon and Goroff have formed ParaSoft, a company that will provide commercial support for CrOS, CUBIX, and PLOTIX. "We believe that the CrOS approach provides the speed and versatility needed to build a large base of hardware-independent applications codes," said Flower. "We are committed to providing CrOS support for both current and future Intel machines."

Noncommercial versions of CrOS III, CUBIX, and PLOTIX, including iPSC versions, can be obtained from Adam Kolawa, 206-49 California Institute of Technology, Pasadena CA 91125, 818-356-2907.

SugarCube™ System Sweetens Entry Into Concurrent Computing



The SugarCube™ System offers a complete concurrent workstation for as little as $44,950. SugarCube™ systems serve as excellent entry-level machines for educational needs, as well as development workstations for large-scale concurrent computation and OEM applications.

"With the rapidly expanding interest in concurrent computing, there's a critical need for an exploration tool priced under $50,000," said Paul Wiley, SugarCube™ system product manager at iSC. "That's why we developed the SugarCube™ system." The SugarCube system offers a complete concurrent workstation for $44,950, with a vector workstation for under $70,000. And applications developed on the SugarCube system are binary-compatible with larger iPSC systems, making it an excellent entry vehicle into concurrent computing, as well as an ideal development workstation.

The SugarCube system comes in four configurations: basic eight-node system, extended memory system for AI research, vector system, and hybrid (vector and extended memory) system. The AI and hybrid systems include Concurrent Common LISP from Gold Hill Computers at no additional cost.

The SugarCube system is an extension of the iPSC family of 80286-based concurrent computers, but repackaged for size and cost-effectiveness. Each SugarCube system includes a multi-node hypercube, a Cube Manager, and a full range of development tools. Standard tools include the "NX" node executive, a concurrent debugger, communication routine library, FORTRAN, C, and the iPSC system SIMULATOR for off-line program development. In addition, CCLISP is standard on the AI and hybrid systems, and the VAST-2 FORTRAN Vectorizer from Pacific Sierra Research is available at additional cost for the vector and hybrid systems. (See related article on development tools.) Standard service, support, and training are also included with SugarCube.

Wiley sees several uses for SugarCube systems. "First, the SugarCube™ system is an excellent place to begin exploring concurrent computing for a minimal initial investment. And with *complete upward compatibility* with the larger iPSC systems, that investment is well protected." As the customer expands to larger iPSC systems, the SugarCube system continues to serve as a program development workstation. Completed programs can then be run with full data or a full complement of processing nodes on the larger iPSC system.

"In addition, more educational institutions can now afford concurrent computing," said Wiley. A typical university scenario includes students doing routine coursework with iPSC SIMULATOR packages on UNIX- or XENIX- based PCs, VAXs, or workstations. The SugarCube system is then used for projects requiring speed or actual hardware-based timing measurements.

The SugarCube system also makes an ideal development station for current large-system iPSC users. SugarCube systems can attach to the same TCP/IP Ethernet, and source and binary programs can be exchanged with a larger, central iPSC system, providing a cost-effective way to reduce the user load on current iPSC systems.

Finally, the SugarCube system's price/performance makes it an excellent choice for OEM use in production computing applications. "We expect the SugarCube™ system to stimulate interest in concurrent computing among OEMs, for both individual and shared systems," said Wiley.

SugarCube configurations are:

- iSGR/d3—basic system of eight standard nodes, each with 512K local memory. Designed for general purpose concurrent computing applications and research.
- iSGR-MX/d2—four memory nodes with 4.5 MBytes of memory each. Designed for the large programs and data structures required in symbolic and AI applications. Includes Gold Hill's CCLISP.
- iSGR-VX/d2—a vector system with four vector nodes. Designed for large-scale compute-intensive applications, with a peak performance of 26 MFLOPS.
- iSGR-HX/d2—a hybrid system with two VX vector nodes and two MX memory nodes. Designed for applications that require both numeric and symbolic processing. Includes Gold Hill's CCLISP.

For details on SugarCube systems, contact Intel Scientific Computers, 15201 N.W. Greenbrier Parkway, Beaverton, OR 97006, 503-629-7629.

iPSC Software Development Tools

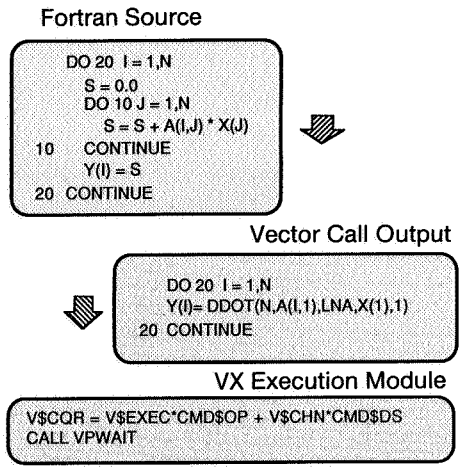
The iPSC software development environment offers a well-stocked tool kit to aid software developers. Some of the most important tools are described here.

The VAST-2 Vectorizer automatically vectorizes sections of FORTRAN code for execution on a vector node (see illustration). This means easier conversion of existing code for vector machines, and easier, more efficient programming. In addition, system vector performance can be optimized with shorter vectors than before. VAST-2 is produced by Pacific Sierra Research, a well-known developer of vectorizing software for supercomputing environments.

The iPSC *SIMULATOR* software package simulates the iPSC concurrent environment on machines running XENIX or Berkeley UNIX. Priced under $500, the

(continued p.4)

iSCurrents



VAST-II VECTORIZER

(continued from p.3)

SIMULATOR package offers a method of exploring concurrent computing on existing sequential systems, with a very small initial investment. SIMULATOR is included with all iPSC and SugarCube systems.

The iPSC system's concurrent debugger, is the first commercially available symbolic debugger for large-scale distributed-memory concurrent machines. In addition to conventional process-based symbolic debugging capabilities (monitor and control any process on any node), the debugger provides examination, tracing, and control of message traffic between nodes. A FORTRAN version is included in the current operating system (Release 3.1). A C version will be included in the next system release (4.0), later this year.

System Release 3.1 also supports *hybrid cube* capability. This allows a single hypercube to contain both vector and symbolic nodes, giving access to two different programming models in the same system. Numeric computations using vector (VX) nodes can call on symbolic (MX) nodes running CCLISP for interpretation of results. Alternatively, symbolic nodes can control computations on numeric nodes, as needed.

In addition, Release 3.1 includes a prototype version of a *remotely connected host* facility for user evaluation and comment. The remote host facility allows the user to control the iPSC system from SUN-3 workstations.

Several users on SUN workstations can subdivide the iPSC system among themselves, with each user assigned a subset of the iPSC processing nodes. This *cube sharing* capability allows users to maximize utilization of cube resources, apportioning the cube out to several smaller tasks or dedicating it to a single user for larger tasks.

For more information about any of these tools, contact Intel Scientific Computers, 15201 N.W. Greenbrier Parkway, Beaverton, OR 97006, 503-629-7629.

SugarCube™ System Is Ideal Educational Package

With the announcement of the SugarCube system, Intel Scientific Computers now offers the ideal entry-level package for educational concurrent computing needs:

- The SugarCube system serves as a low-cost concurrent computer, well suited to classroom computing needs.
- The iPSC SIMULATOR software package, included with the SugarCube system, allows students to prototype codes on a variety of familiar UNIX and XENIX workstations. These codes can then be run on the SugarCube system for true concurrent processing.
- Each SugarCube system includes one training credit for iSC's *Programming Concurrent Computers* workshop, a five-day training class that provides lectures and hands-on lab sessions to build concurrent computing expertise.

In addition the CrOS III Communications system from Caltech runs on all SugarCube systems, and is documented in *Solving Problems on Concurrent Processors* by Dr. Geoffrey Fox, et al.

In Future Issues . . .

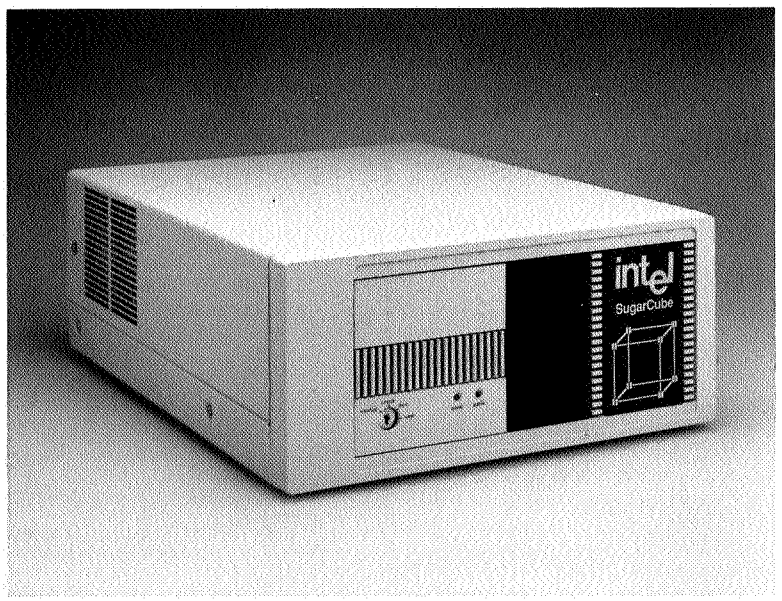
. . . of ***iSCurrents*** we'll feature iPSC applications in image processing, modeling and simulation, and CAD, plus a special feature on AI and symbolic computing, and other developments in the field of concurrent computing.

Intel Scientific Computers
15201 N.W. Greenbrier Parkway
Beaverton, OR 97006

intel

SugarCube™ Concurrent Computing Workstation

Product Brief



SugarCube™

Concurrent Computing Workstation

The Intel SugarCube™ system is a concurrent computing workstation designed for application developers, individual researchers, and concurrent computing OEMs. The system is an extension of the iPSC™ family, and supports the complete software development environment of the iPSC systems. SugarCube systems support both memory and vector enhancements to meet a broad range of needs.

The SugarCube offers:

- A low-cost eight processor entry-level concurrent computer system
- A productive program development environment
- Upward compatibility to all iPSC systems and software
- Network support for shared access to the computer
- A system designed for workstation use in office or lab

A Scalable Architecture

The SugarCube system is an ensemble of independent computers, or nodes, connected in a hypercube network topology. This distributed memory architecture minimizes the system bottlenecks that limit scaling to larger systems, and permits extension of the architecture to hundreds of processors.

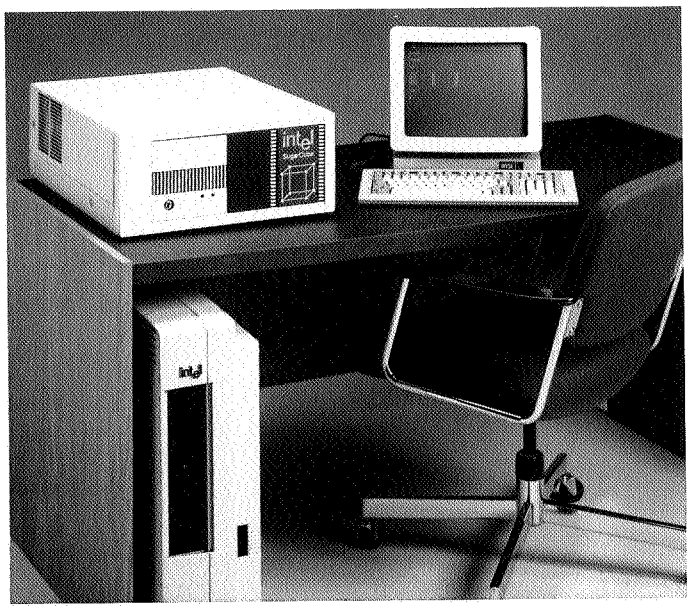
The SugarCube™ System

The SugarCube system consists of the Cube and the Cube Manager. The Cube comprises up to eight processing nodes. Each node is an independent single board computer that includes its own local memory and point-to-point communication channels. The operating system on each node manages node resources, ensures multi-process memory protection, and provides message routing and communication services.

The Cube Manager supports the program development environment and provides system-level control for the Cube. It also supports I/O services for the Cube, and serves as the gateway for Ethernet-connected remote users of the Cube.

Programming the Cube

The Cube features standard programming languages, including FORTRAN, C, and LISP, with compatible concurrent message-passing libraries. Applications are implemented on this software architecture as multiple processes that communicate by passing data messages. Multiple processes can exist on each node to allow the development of sophisticated algorithmic models. With no shared resource, this communicating process model lets users develop applications that are independent of physical Cube size, and that can be scaled to larger systems as computational needs dictate.



Intel SugarCube™ systems consists of the Cube Manager (bottom left), user terminal, and Cube (top left), which provides the computing power of 4 or 8 processing nodes.

Four SugarCube™ Models

SugarCube systems are available in several models to meet a broad range of concurrent computing needs.

Standard SugarCube System: Standard systems consist of eight nodes, each with 500 KBytes of local memory, and support the needs of concurrent computing research and application development.

Extended Memory System: The SugarCube-MX system provides a four node system with 4.5MBytes of memory on each node--useful for symbolic research and memory-intensive applications. For example, symbolic applications using CCLISP™ require large-memory configurations to support the programs and data structures generated in the LISP environment.

Vector System: Designed for large-scale computationally intensive numerical applications, the four-node SugarCube-VX system supports a high-performance floating-point processor on each node. VX systems can achieve performance increases approaching 100 fold over an equivalent number of standard nodes.

Hybrid System: Applications requiring both numeric and symbolic computing are supported with hybrid versions of SugarCube systems. Two vector nodes and two enhanced memory nodes can be configured in a single SugarCube chassis. Hybrid applications are also supported by the system software, which provides compatible message-passing between vector and symbolic nodes.

SugarCube™ Software

The Node Operating System

The programmer views the Cube as an ensemble of processing nodes with an arbitrary interconnection. Each node is independent and asynchronous. The node operating system, NX (Node executive), enables any node to send a message efficiently to any other node.

NX is a multitasking operating system that supports process-to-process message passing. Consistency is maintained for both inter-node and intra-node process communication. This capability allows the user to emulate much larger systems, by creating virtual nodes using multiple independent processes in each physical node. Using this technique sophisticated programming strategies can be studied by simulating Cubes of varying dimension.

Program Development Tools

The SugarCube system is provided with the most complete set of program development tools available for any concurrent machine:

Integrated Languages and Hardware: A variety of demanding applications for concurrent computers require dissimilar languages and hardware, which in turn require complex interfaces. This is the case for hybrid applications, which use the numeric segment of the system to process and reduce incoming data, and the symbolic portion to analyze these results. Standard call sequences and message-passing protocols in C, FORTRAN, and LISP provide the basis for supporting these complex applications in the SugarCube system and other iPSC systems.

Concurrent Debugger: SugarCube systems support a powerful multiprocess symbolic debugger. It manages simultaneous run-time monitoring of multiple executing processes and node-to-node message passing between those processes. At the process level the debugger supports capabilities including code and data break-points, trace-points, single-step execution and data examination capabilities. Process-to-process communication debugging offers event breakpoints, message status checking and system message queue examination.

Communication Library: Many concurrent computing problems have a natural decomposition which corresponds to a topology that can be simulated on the hypercube. A user library of communication subroutines is provided with the SugarCube to simplify program development for such common communication topologies as ring , tree, and two- and three-dimensional mesh topologies. Global operations are also included.

VAST-2™ FORTRAN Vectorizer : For numerical applications the VAST-2 option provides an efficient user interface between FORTRAN application programs and SugarCube-VX processors. It decomposes loops into functions that execute on VX processors. VAST diagnostic messages provide a powerful tool for optimizing vector content in code by pointing to vectorization inhibitors.

Concurrent Common LISP™: CCLISP™, from Gold Hill Computers, is the first commercially-supported concurrent version of Common LISP. In addition to the message-passing constructs of C and FORTRAN, CCLISP also implements Message Streams and Remote Evaluation between nodes.

Shared and Multiuser Support: SugarCube systems can be shared among a network of workstations, allowing users to partition a single physical Cube into several subcubes that are dedicated to individuals. The workstations serve as remote hosts for the Cube, and the Cube Manager becomes a transparent network gateway and compile server. This capability allows users to program the Cube from a familiar environment, with familiar development tools.

iPSC System Simulator: Another useful tool for developing and testing parallel programs is the cube Simulator. Supported on the Cube Manager and a variety of UNIX-based workstations, it provides a full simulation of the SugarCube and iPSC hypercube environment for concurrent program development.

Training courses provide an opportunity to become familiar with programing concepts and development tools for SugarCube and iPSC systems. Laboratory sessions allow the development of practical skills, and the opportunity to pursue special concurrent computing interests with the Intel staff.

Training

One Intel training credit is provided with the purchase of SugarCube systems. A Vector Course credit is also provided with SugarCube-VX systems. The courses offered include:

- iPSC Concurrent Programming Course
- iPSC Vector Concurrent Programming Course
- iPSC Gold Hill LISP Concurrent Programming Course (available from Gold Hill)

Service and Support

SugarCube systems are installed by the purchaser, with Intel Scientific Computers providing pre-installation consultation for site preparation. A 90-day on-site warranty is provided for SugarCube hardware. System software is supported for one year after installation at no additional charge . Software support includes software updates, technical reports, software problem reporting, and technical information phone service.

SugarCube Specifications

Model No.	Nodes	Cube Dimension	Memory MBytes	Peak Performance MIPS	MFLOPS
iSGR/d3	8	3	4.0	8	.4 (64-bit)
iSGR-MX/d2	4	2	18.0	4	.2
iSGR-VX/d2	4	2	6.0	4	26
iSGR-HX/d2	4	2	12.0	4	13

Electrical		Environmental		Physical	
Power	800 Watts, 47-63 Hz	Noise	50 dBA	Height	7.75 in.
Voltage	115 VAC (90-132)	Temp., oper.	10° to 35° C †	Width	17.50 in.
Current	10 Amps	Altitude, oper.	0 to 10,000 ft.	Depth	22.25 in.
	220 VAC (180-264)	Humidity oper.	85% non-cond.	Weight, Cube	50 lb.
	6.0 Amps	† minus 1° per 1,000 ft. altitude		Cube Man.	50 lb.

Safety	RFI/EMI
UL listed to UL478 5th edition	Verified to FCC class A (industrial) requirements
CSA certified to CSA C22.2 No. 154 -M1983	Complies with VDE 0871 Level A (industrial)
Certified VDE or TUV for compliance to IEC435	

Cube Manager

CPU	80286/287 8MHz	**Hard Disk**	140 MBytes
Memory	2 MBytes	**Tape Backup**	60 MBytes streamer
Network Interface	Ethernet TCP/IP (optional)	**Floppy Disk**	360 KBytes, 5 1/4 in.

Complete specifications are included in the iPSC System Product Summary and iPSC-VX Product Summary.
Specifications are subject to change without notice. Information contained herein supersedes all previously published information

For more information contact
Intel Scientific Computers
15201 N.W. Greenbrier Parkway
Beaverton , OR 97006

(503) 629-7629

SC7030/0487/10K/S/VJC
ORDER NUMBER 280610-001

PRINTED IN U.S.A. 4/87

THE INTEL iPSC™/2 System
Product and Market Information

Intel Scientific Computers

15201 N.W. Greenbrier Parkway
Beaverton, OR 97006 USA
Chris Wain (503) 629-7631

Piper's Way
Swindon SN1 1RJ, ENGLAND
Dave Moody 793 696000

Table of Contents

I. Introduction

A. Current state of the technology

The market for scientific computing systems can be divided by performance into three general categories: Super-minicomputers with peak performance below 10 million floating-point operations per second (Mflops), Mini-Supercomputers with performance ranging from 10 to 100 Mflops, and Supercomputers with performance greater than 100 Mflops. This classification can be expanded still further to include both sequential and parallel architectures. The table below shows representative products for each segment.

Classification	Performance	Sequential	Parallel
Super-Minicomputers	<10 Mflops	VAX 8800 IBM 4300 Gould Elxsi	Sequent B21000 Encore MultiMax Flexible Flex-32
Mini-Supercomputers	10-100 Mflops	Convex C1 SCS-40 FPS M/64 series Cydra	Alliant FX-8 Multiflow
Supercomputers	>100 Mflops	Cray X-MP Amdahl/Fujitsu NEC SCX	Intel iPSC-VX N Cube/10 FPS T-series

In general, **Super-minis**--though heavily used for scientific applications because of their availability--find their principal benefit in more general-purpose applications. **Mini-Supercomputers** are designed primarily for scientific computing, and typically incorporate vector processing features to boost performance on these applications. **Supercomputers** find their use almost exclusively in scientific applications, and apply the most elaborate architectural methods for achieving the highest possible performance.

For all sequential machines, code compatibility and the ease with which existing application code can be ported are primary considerations in their design; performance is of secondary importance. As a result, the cost per

calculation, or price-performance, is essentially identical for all systems, no matter what the performance range.

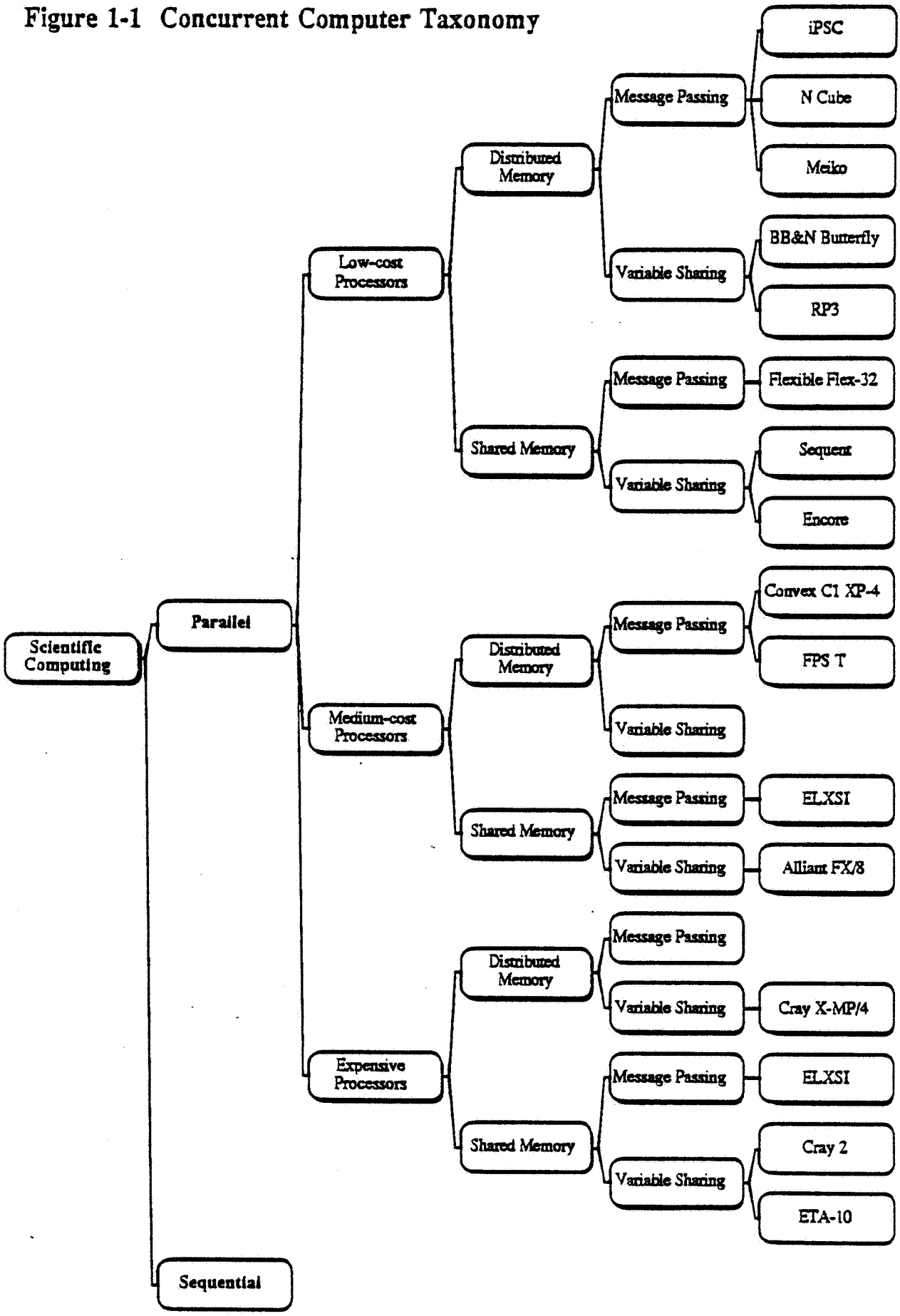
The lower-end parallel machines reflect many of the same capabilities and constraints of their corresponding sequential counterparts. For example, parallel super-minis support general-purpose applications, and parallel mini-supers utilize vector processing techniques to boost performance on scientific applications. Both types of machine attempt to provide a high degree of compatibility at the application level for rapid porting of commercial applications, and reflect similar price-performance capability.

Parallel Supercomputers on the other hand stand out by their break with tradition from the other computational alternatives. Hypercube architectures dominate, and, three features distinguish these machines from other parallel architecture systems: first, they use distributed memory (each processor has its own local memory) instead of a shared common memory; second, they use message based operating systems instead of shared variable operating systems; and, third, they can be scaled to hundreds and eventually thousands of processors, as opposed to a dozen or so.

Because of these differences, hypercube machines are often referred to as **concurrent systems**. Computer scientists refer to them as distributed-memory message-passing (DMMP) architectures. See Figure 1-1. The term "concurrent" denotes the more specific form of parallelism whereby multiple processors work together on a single computational problem, sharing the load. "Parallel" is the more general term that often is used to describe other forms of parallelism, like pipelining and multiprocessing.

Hypercube systems are distinguished from their shared memory counterparts both in performance and price-performance. These benefits come at a cost—program development. Automated methods for porting sequential code to distributed-memory message-passing machines are still in the research stage. Some form of human intervention is necessary to make the translation. Thus, the "effort for benefit" argument applies. Simply stated, hypercube machines offer the best performance for the price of any computational alternative, by a wide margin. The effort is development time, the benefit is supercomputer performance for minicomputer prices.

Figure 1-1 Concurrent Computer Taxonomy



Application Architecture Processors Memory Communication Examples

The use of familiar languages, like FORTRAN, C and LISP, minimize this programming effort, so that it is seldom necessary to completely rewrite an application. But, best results do require a familiarity with the application and a knowledge of the code.

B. Key hypercube market factors

The hypercube architecture marks a computer revolution similar to the development of Cray class supercomputers. Typically, advances in scientific computing have occurred along the lines of evolutionary developments, not revolutionary changes. Consequently, different metrics for success apply to hypercube machines compared to more conventional architectures. As a developing technology, the primary objectives for hypercube manufacturers have been to achieve maturity in software development tools, provide stability in the system environment, and maintain flexibility for migrating to compatible future generations of machines.

In the two years since the commercial introduction of hypercube machines, use has been most prevalent in research activities associated with concurrent algorithms and applications. Applications have developed along two principle lines: numeric applications, typical of engineering and scientific computing needs; and symbolic applications, characteristic of artificial intelligence, database applications, and defense applications including battle management and battlefield simulation. Significant results have been obtained for a variety of these "high-value applications" (applications that justify the developmental effort for the improved performance obtained). Applications that are approaching commercial introduction include fluid and thermodynamic modeling, molecular modeling, and structural analysis.

As mentioned above, effectiveness as a productive research and development vehicle has been the principle measure of success for hypercubes. In this regard, the first generation Intel iPSC system has been effective. The philosophy behind its development was to capitalize on the price and maturity advantages of merchant silicon, and to both track and push the rapidly developing knowledge base of concurrent computing with system enhancements that combine these technologies. The experience gained from this first generation machine has provided critical guidelines for defining the capabilities of the second generation machine.

C. First Generation Experience

The iPSC system was the first hypercube machine to be made available as a commercial product. A principal concern at its introduction was the perceived difficulty associated with parallel programming. Existing applications had been developed for sequential machines using languages that assumed shared memory architectures. Experience at Caltech using these languages (C and later Fortran) had demonstrated that suitably motivated (interested) researchers without any special background in concurrent computing were able to program highly efficient applications, and attain good performance without too much additional effort. The case for migrating existing programs and full applications to hypercube architectures had yet to be demonstrated. The concern was that the effort would outweigh the benefit, or that old algorithms could never effectively utilize such a change in architecture.

The answer to this question lay in the methods for problem decomposition. In effect, once a problem has been divided into small pieces, and distributed to the processors of a concurrent system, the programming model at each processor is identical to the old sequential shared memory model. Thus, for a wide variety of applications, the heart of the application code remains unchanged. The core problem being solved by each processor is identical to that solved by the sequential machine, only smaller. This has wide-reaching implications when considering the importance of maintaining computational accuracy and stability in mathematical simulations.

The additional effort for concurrent programming involves writing the code that knits together these loosely connected portions of the computation that are being executed in different processors. The message-based operating systems of hypercubes provide structure (discipline) and significantly simplify the communication interaction between tasks. Additionally, concurrent execution is often easy to manage because it is close to the model of the application.

Interestingly, many naturally concurrent tasks have been implemented at great effort using sequentially executing code. This is particularly true in problems and applications relating to naturally occuring phenomenon such as

physical sciences and imitating cognitive processes. For these, a concurrent programming model is preferred over a sequential one.

Some Results

The results of the past two years have been very encouraging, and confirm Intel's belief that concurrent programming is not difficult, just different. One user at the University of Houston, investigating electromagnetic scattering phenomenon, ported an application program of about 1400 lines to the iPSC system by simply changing a dozen lines of code.

Oak Ridge National Labs, in an attempt to understand the problems associated with concurrent programming on distributed memory machines, solicited a half dozen applications from researchers at the lab who wished to migrate their applications to a hypercube. These applications were chosen without regard to their expected adaptability to a concurrent implementation. In some cases the applications involved thousands of lines of code, but the entire project was completed in less than six months. Of the six, only one did not result in improved performance or meet project objectives. This one problem was successfully ported, but ran more slowly because of communication overhead. The 'computation-to-communication' ratio required for the application was not balanced to the capabilities of the original iPSC system. Some of the capabilities of the iPSC/2 system address this issue.

Recently, a month-long effort was undertaken to move a seismic simulation problem to the iPSC-VX system. This project resulted in a performance of 227 Million floating-point operations per second (Mflops) on 32 nodes, several times faster than Cray supercomputer performance—for about one tenth the investment.

Also, Nektonics has implemented their Nekton fluid and thermodynamics modeling application to the iPSC system and simulations that previously required 4 hours on a DEC 8600 were reduced to 3 minutes. This work required about six months of programmer effort.

Many other examples can be cited. Particularly exciting is the opinion of some researchers that concurrent programming may be the most natural method for writing computer programms.

D. Second Generation Needs

Achieving high computational efficiency requires the proper balance between computational elements, communication elements, and the memory elements of the system. Optimizing the cost and capability of a system for certain classes of applications requires modularity and upgradability in the system elements. Modularity also extends the product life by permitting evolutionary enhancements as technology becomes available. Further, modularity broadens the application base, making hypercube machines more versatile for a wider variety of applications.

Second generation machines require greater sophistication in the development environment to allow more productive utilization of programming resources. Languages that evolved from personal computer environments have limitations and restrictions that are not acceptable in the large-scale computing environment. Mainframe quality in languages and tools is essential.

II. Technology and Product

A. System requirements

Experience using the first generation iPSC system, introduced in 1985, identified the need for a stable and mature development environment for supporting a sophisticated user community. With users coming from the mainframe and supercomputer environments, it was essential that iPSC/2 system aspire to that level of robustness.

The iPSC/2 system builds on lessons learned with the first generation counterpart. Modularity is a principal requisite to provide the means for adapting and upgrading system components as user needs change, and as technology develops. Evolving the performance and capability of the system as technology matures preserves the investment of the user.

Intel's strategy is to apply modularity to elements that are expected to demonstrate the most rapid rate of change. For the iPSC/2 system these are the Direct-Connect™ communication module, scalar and vector floating-point accelerators, memory modules, and modular software that provides flexible interfacing to different user environments. Arithmetic capabilities are supported in a modular fashion using an arithmetic co-processor socket that supports both the 80387 co-processor or the Weitek 1167 arithmetic accelerator module. The iLBX™-2 interconnect is maintained for the iPSC-VX vector arithmetic accelerator, and for future enhancements.

Hypercube communication enhancements

Applications experience using the first generation iPSC system established the need to efficiently support two different communication requirements: minimum delay for short messages, and maximum communication bandwidth for long messages. The Hypercube Routing Module of the iPSC/2 system uses a communication technology called **Direct-Connect Routing**, which works in a similar way to a telephone exchange at each node. This approach avoids the incremental delay associated with "store and forward" message relay mechanisms used in all first generation hypercubes.

Direct-Connect routing minimizes short message latency (delay) by supporting message routing in hardware. As a result, communication latency is essentially the same to any node in the system, regardless of the number of

intermediate nodes in the connection. For long messages, Direct-Connect routing makes it possible to send messages of any size (up to the capacity of memory) to any node in the system at the full communication bandwidth.

Direct-Connect routing dramatically alters the programming methodology used for hypercube machines. Because of incremental performance cost for communicating beyond nearest neighbors, first generation hypercubes required programmers to organize the computational problem to minimize multi-hop communication. Direct-Connect routing imposes virtually no added penalty on multi-hop communication. Consequently, the machine can be viewed by the programmer as an ensemble of processors with an arbitrary interconnect. From an application standpoint, Direct-Connect routing reduces latency by as much as a factor of 6 for short message multi-hop communication, and increases bandwidth by up to 10 fold for long messages, over that the first generation iPSC system.

Node performance enhancement

The transition from a 16-bit to a 32-bit node architecture is comparable to the move from minicomputers to super-minis. Language limitations and operating system restrictions of the 16-bit addressing mechanisms simply disappear. Hypercubes are capable of handling very large problems, and with that comes large application codes. Users of first generation machines often spent more time on porting code to the personal computer oriented languages and operating systems than in developing parallel algorithms. To the programmer, the parallelization task was minor compared to the task of language and operating system migration.

The languages and tools selected for the iPSC/2 system are derived from the super-mini and mainframe technology. Unix V.3 provides the basis for the development environment. The node operating system, NX/2, has itself been completely redesigned and reimplemented to take full advantage of the 32-bit capability of the 80386 node CPU.

Language support comes from production compilers originating in the minicomputer world, which utilizes 32-bit pointers, and providing extensive high quality code generation optimizations. For instance, mixed language applications are supported, allowing the programmer to use the language best

suited for the different tasks that make up an application: C, FORTRAN, or LISP. FORTRAN not only adheres to FORTRAN-77 standards, but also includes DoD extensions and popular VMS extensions. Also, the capabilities of the processor—address space and speed—allow for a full, rather than a partial, implementation of Common LISP.

Modular memory enhancement

Memory technology has historically been the area of most rapid development. Memory capacity has doubled every two years for the last decade and a half. Today, using surface mount manufacturing techniques and readily available components, it is possible to configure an 8-Mbyte memory in the area of a 3x5 inch card. Within the lifetime of the iPSC/2 system, this capacity can be expected to quadruple.

Research has shown that hypercube systems are best adapted to medium- and large-grain applications, for which the large resident memory capacity of hypercube systems are one of their most attractive features. For numerical applications, computational efficiency typically improve as the size of the problem on each node increases. GigaByte size problems are not uncommon. For symbolic applications, many Mbytes of memory can be require on each node just for the program code, and significant problems can easily require 8 Mbytes or more per node. Simply stated, the memory requirements for hypercube applications may be as insatiable as those experienced by other computer system technologies.

Modular Software

Experience with the first generation system demonstrated that the most rapid changes in concurrent systems were in software. For example, the first generation iPSC system experienced two complete reimplementations of the node-based operating system kernel after product introduction. Each new release resulted in significant performance increases. The new iPSC/2 kernel—**Node eXecutive** (NX/2)—follows that tradition by preserving the message model established with the original iPSC system, thereby maintaining application level compatibility for the user.

Further, NX/2 is faster than its predecessor because many of the communication duties previously handled by the operating system are now

handled by the Direct-Connect routing hardware. Modularity in the original design made this transition relatively easy to effect, and maintained user code compatibility. Also, the modular architecture allowed new features to be "plugged-in" efficiently. Most prominent of these are space sharing—dividing the hypercube into subcubes—by means of physical and virtual node identifiers, and support for distributed development. The **Concurrent Workbench™** provides users with an interface to the iPSC system from a development workstation. This interface gives the user the ability to control, access, and program the iPSC system from a foreign workstation environment. The tools originally provided on the System Resource Manager (SRM) appear to the user to run on the workstation.

B. iPSC/2 System

A typical iPSC/2 system contains three component elements: the **Cube** that contains the node processors connected in a hypercube topology; the **System Resource Manager** that serves as a gateway to the cube for network users, supports the Multi-user Concurrent Workbench™ software, and provides I/O services; and a **network of user workstations** for application development and access to the cube.

The iPSC/2 system is supported in three configurations: the standard system with 1 to 8 Mbytes of memory per node and up to 128 nodes, the extended memory iPSC/2 MX system with 16 Mbytes per node expanding up to 64 nodes, and the vector iPSC/2 VX system with up to 9 Mbytes and 20 Mflops (single precision) peak performance per node. Each of the systems are provided with the 80387 numeric co-processor and can also support the Weitek 1167 arithmetic accelerator. Table 2-1 summarizes these system configurations and their capabilities.

IPSC/2	d4	d5	d6	d7
Nodes	16	32	64	128
Memory Capacity (8 Mbyte nodes)*	128	256	512	1024
Communication Channels	32	80	192	448
MIPS	64	128	256	512
MFLOPS (single precision)	8	16	32	64
MFLOPS (double precision)	5	10	19	38
Whetstones (MWIPS)	56	112	224	448
Peak Memory Bandwidth (MBytes/sec)	256	512	1024	2048
Peak Communications Bandwidth (MBytes/sec)	80	160	320	640
Peak Messages/sec (Kmessages/sec)	49	98	197	394

IPSC/2 VX	d4	d5	d6
Nodes	16	32	64
Memory Capacity	144	288	576
Communication Channels	32	80	192
MIPS	64	128	256
MFLOPS (single precision)	320	640	1280
MFLOPS (double precision)	107	213	427
Whetstones (MWIPS)	56	112	224
Peak Memory Bandwidth (MBytes/sec)	768	1536	3072
Peak Communications Bandwidth (MBytes/sec)	80	160	320
Peak Messages/sec (Kmessages/sec)	49	98	197

* Memory sizes are 1, 4, 8, and 16 MBytes per node

Table 2-1 iPSC/2 Configurations and Specifications

Each configuration supports **FORTRAN, C and Concurrent Common LISP™**. The iPSC/2 vector system is also supported by the **VAST-2 FORTRAN vectorizer**. All systems operate under the **Concurrent Workbench™** software that provides shared access to the cube from workstation networks. The Concurrent Workbench software provides an efficient programming environment so that users can benefit from local workstation tools with which they have become familiar, and share access to the Cube with other users. The iPSC/2 **Concurrent Debugger (DECON)** provides facilities to monitor the execution of processes on multiple nodes and supports monitoring the message traffic between those processes.

C. iPSC/2 Enhancements

The combination of Direct-Connect routing and communication performance enhancements result in order-of-magnitude increases in communication performance. This impacts fundamental assumptions regarding the programming

of hypercubes--that the topology the problem must be mapped to the machine. With communication to distant nodes equally efficient as communication to adjacent nodes, the programmer ignores topology and distribution of the computational task over the hypercube is similar to shared memory parallel machines.

Direct-Connect Routing

Direct-Connect routing simplifies the programming problem by making it possible for data to reside on any node. Because message latency and communication performance are essentially the same for communication between any two nodes in the system, performance is independent of the location of data. This programming model has been supported in the node operating system of the iPSC system from its initial release. Consequently, applications developed for the first generation iPSC system will run faster, and without modification on the iPSC/2 system.

Direct-Connect routing also eliminates communication collision problems. The benefit of this is exhibited in a worse case communication example that occurs in the two-dimensional fast fourier transform (FFT) algorithm. The computation requires that a series of FFT's be performed on each node, after which each node must send portions of its result to all other nodes. This transpose of row to column data orientation swamps the store and forward hypercube routing approaches of first generation systems. Direct-Connect routing allows the communication to proceed with almost no interference. The result is a 10x speedup in communication, and an overall six-fold increase in performance over the first generation iPSC system. See Figure 2-1.

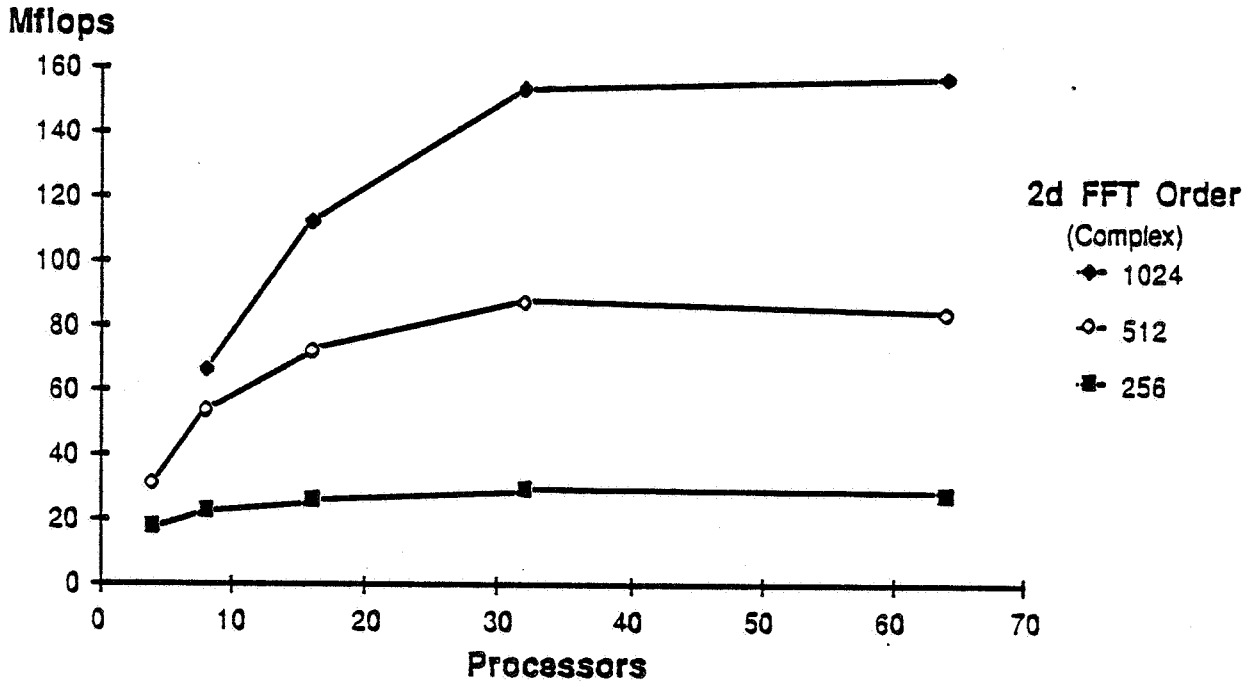


Figure 2-1 2d FFT Performance for iPSC/2 VX System
Performance vs. number of processors is shown for three problem sizes. Each node runs FFT algorithm at about 10 Mflops. Computational efficiency varies with the amount of work on each node relative to communication load, and ranges from a high of 85% (512 x 512 problem on 4 nodes) to a low of about 5% (256 x 256 problem on 64 nodes). This illustrates scalability—using the right size machine for a given problem size. Concurrent computers find their prime use in solving large problems efficiently, not small problems faster. Note how performance increases with problem size for same number of processors.

Long messages up to 10 times faster

Speeding up the communication data rate produces a better communication balance for the computational power of the iPSC/2 VX vector processor. The result is improved computational efficiency and realizable performance approaching linear speed-up. An example of this benefit can be seen in the matrix factor computation, the compute-intensive portion of the LINPACK benchmark. This implementation uses a parallel version of LINPACK called LINCUBE. The performance of the 16-node vector machine more than doubles, and a 64-node vector machine increases 3 fold, yielding a performance of approximately 90 Mflops. See Figure 2-2 below.

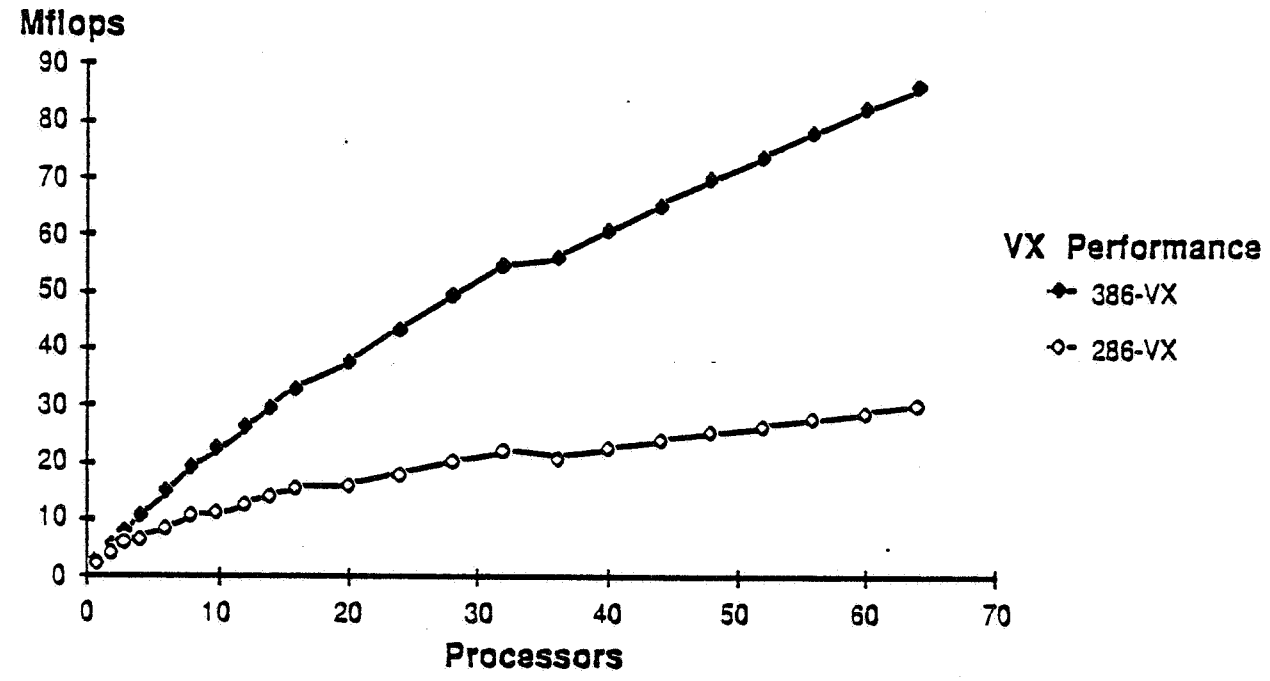


Figure 2-2 LINCUBE Matrix Factor Performance
Performance is shown as a function of the number of processors, with comparison curves for first and second generation iPSC vector systems. For this computation, the problem size is allowed to increase in proportion to the number of processors. Thus, the curve indicates the performance for the largest problem that will fit in the designated number of nodes. LINCUBE is a concurrent version of LINPACK that runs on hypercubes. The matrix factor is the compute-intensive portion of the LINPACK benchmark, and includes both global and local communication requirments.

Short message latency improved by 3X

Improving short message latency is particularly beneficial for symbolic and artificial intelligence (AI) applications. This feature also benefits the efficiency of automatic load-balancing methods. The combined benefit of a factor of three reduction in message latency and a 4X improvement in CPU performance produces an overall 10-fold increase in performance over the first generation iPSC system for the Gabriel Triangle benchmark using CCLISP™. For a 128 node iPSC/2 system, this result is nearly five times faster than the benchmark run on the Cray 1S. See Figure 2-3.

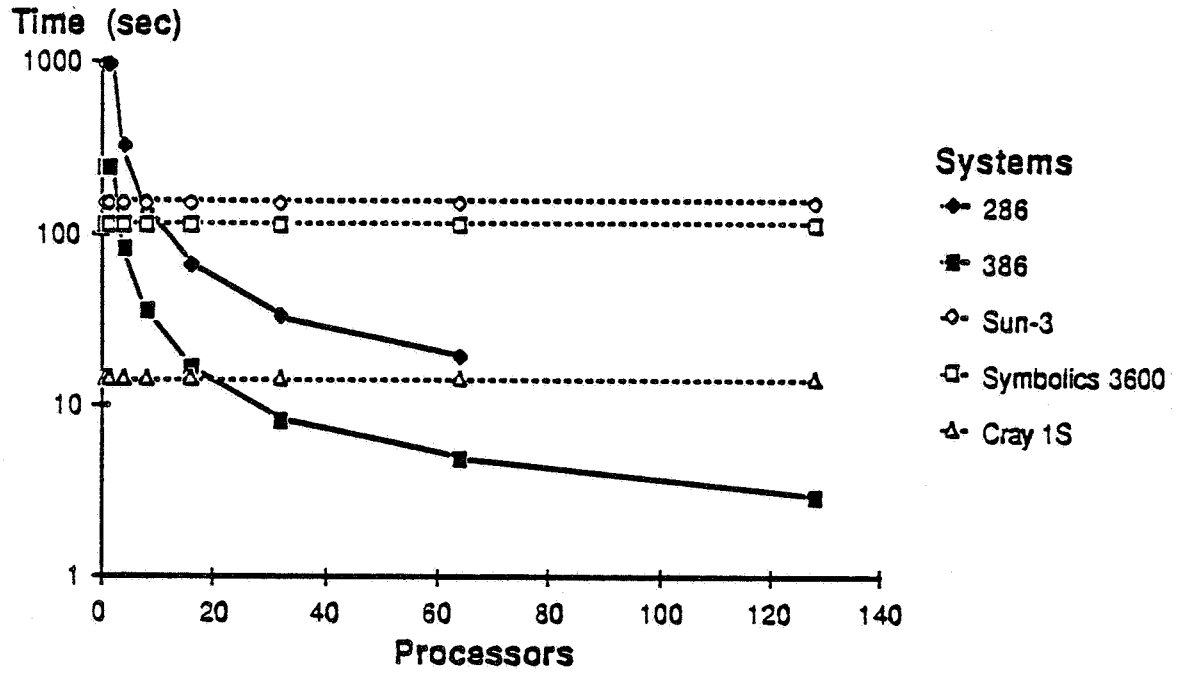


Figure 2-3 Gabriel Triangle Benchmark Performance
Execution time versus number of processors are shown for a variety of systems, including first and second generation iPSC systems. Triangle is one of the Gabriel benchmarks used to evaluate symbolic machines. Triangle adapts well to concurrent computing because it involves an exhaustive search of possible moves, for determining the possible solutions to a peg puzzle, where the pegs are arranged in the shape of a triangle. The problem is executed by a master processor passing out work to worker nodes as they report completion of previously assigned work (sometimes called "hungry puppy" model).

I/O communication 60 times faster

A complete redesign of the communication hardware has increased the data rate between the Cube and the System Resource Manager from about 80 KBytes/second (typical Ethernet TCP/IP performance) to a peak 5 MBytes/second. This makes possible faster communication to external devices, and more efficient support for I/O services like disks and displays. Direct-Connect routing supports direct access between the SRM and each node in the Cube at the full bandwidth of the port. Future system enhancements will utilize the external interface port on multiple nodes to support multiple I/O interfaces transferring data concurrently.

Node processing and memory enhancements

By effectively coupling a variety of new technologies, the node design of

the iPSC/2 system achieves a dramatic step forward in computational power and configuration flexibility--all within the same form factor as the first generation design. VLSI component transistor count has nearly quadrupled, and memory capacity has increased 16 fold through the use of surface mount manufacturing techniques. The 80386 microcomputer on each node of the iPSC/2 system has increased performance by 3 to 5 times over the 80286 of the first generation iPSC system. The 80386 also supports a sophisticated programming model on each node, providing both multi-process and multi-task functionality. Additionally, the 80387 arithmetic coprocessor, standard on every node, boosts numerical performance by a factor of five over its predecessor, the 80287.

Memory options support from 1 to 16 Mbytes of memory on each node, allowing system configurations with up to a Gigabyte of on-line resident memory. Also, arithmetic options available on iPSC/2 systems include the Weitek 1167 Scalar Numeric Accelerator and the iPSC-VX Vector Numeric Accelerator. These provide a variety of optimization options to the user for maximizing applications performance for the investment dollar.

System software tools

iPSC/2 system programming is supported under Unix V.3 which resides on the System Resource Manager (SRM). The SRM supports the Concurrent Workbench™, manages access to the Cube, and manages Cube I/O resources. The Concurrent Workbench provides multiuser network access to the Cube and shares the Cube among multiple users. Using the Concurrent Worbench software, a programmer can develop an application on a local workstation and access all or portion of the cube for application work.

A typical development session, illustrated in Figure 2-4, would proceed as follows: The first step involves developing or migrating an existing application to the programmer's local workstation. The iPSC simulator running on this workstation might be used to initially debug segments of the concurrent application code. For execution on the Cube, the user would request one or more nodes (a sub-cube), and compile the application using the transparent services of the SRM as a network compile server. The

resulting object program is then downloaded to the Cube, and execution is initiated under the control of the Concurrent Debugger, DECON.

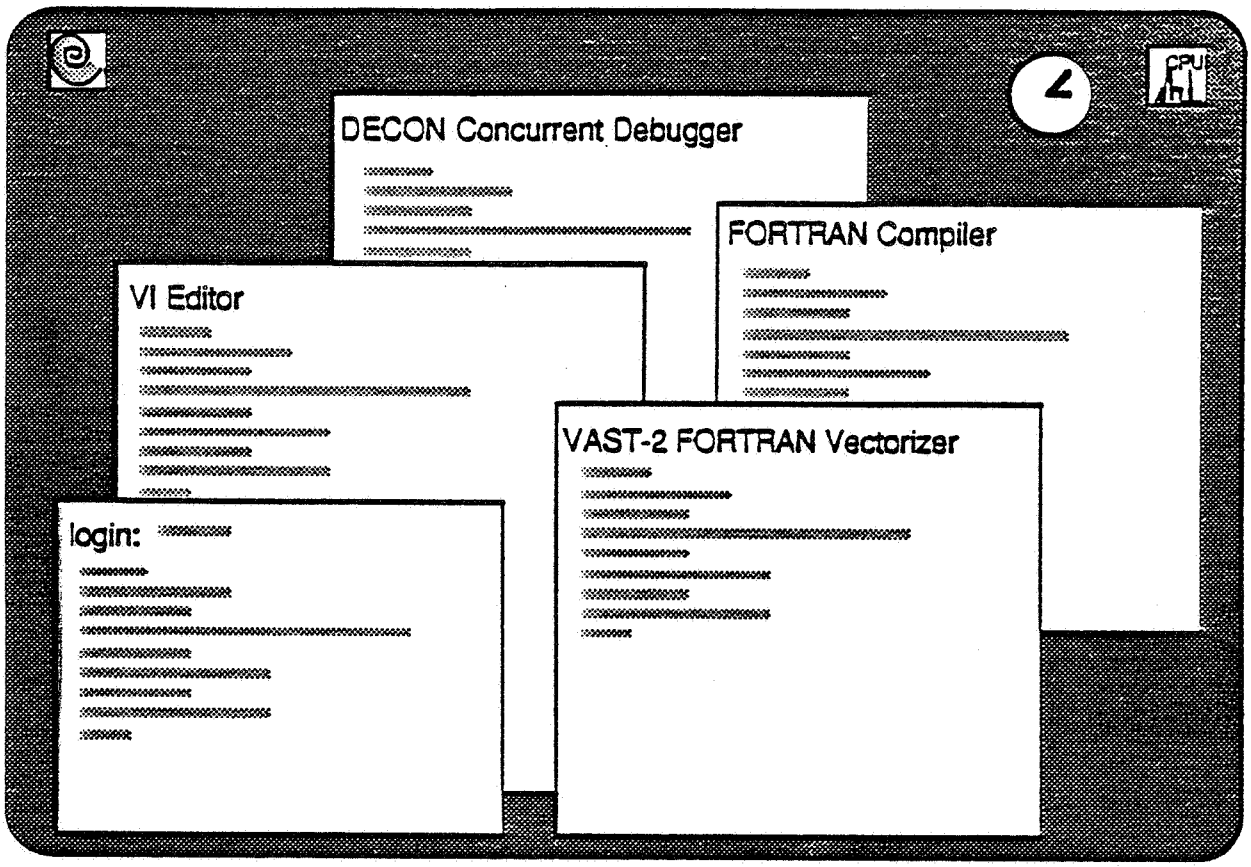


Figure 2-4 Workstation View of a Program Development Session for the iPSC/2 system

DECON is a symbolic debugger that supports simultaneous process-level and communications-level debugging on all nodes of the cube. It addresses in a sophisticated way one of the most critical deficiencies of concurrent systems today: effective tools for debugging concurrent applications. From the workstation, the programmer can edit changes, recompile and perform further testing via the debugger until acceptable results are obtained.

For iPSC-VX applications, VAST-2, a supercomputer-class FORTRAN vectorizer, is used for vectorizing FORTRAN application code, and translating the application code into an executable form for the vector processor. VAST provides diagnostic outputs that identify source loops that can not be vectorized, and indicates the reasons. This aids the programmer in optimizing the FORTRAN source, and in using VAST-2 directives for improved

vectorization. In a recent evaluation by a U.S. national laboratory, VAST-2 was ranked first among a group of minisupercomputer and supercomputer vectorizers for its ability to vectorize a series application benchmarks. The evaluation was based on the ability of vectorizers to "discover" vectorizable segments of code. Often this process involves restructuring the original code into a form that is vectorizable. VAST-2 was better than other highly regarded supercomputer and minisupercomputer vectorizers and vectorizing compilers.

For AI and other symbolic applications, **Concurrent Common LISP™** (**CCLISP™**) will be the primary language for iPSC/2 users. CCLISP for the second generation machine has been improved by extending it to a full Common LISP implementation (as adopted by DARPA), and by providing a unique windowed access for the user to concurrent executing processes. Experience with the the original CCLISP developed by Gold Hill and Intel for the iPSC system made it evident that a sophisticated user interface was needed, giving the developer simultaneous access and control of several LISP environments, one on each node. This has been implemented as **CCLISP Windows** for a variety of popular workstations, including Sun-3 and the XEROX 1186. CCLISP differs from other languages on the iPSC system in that the compiler and its associated environment is located on each node, rather than being cross-compiled on the SRM. This provides added versatility to the user.

D. **Summary**

The iPSC/2 system provides supercomputer performance at 1/10th the price of alternative systems. The symbolic performance is unexcelled. Compared to the first generation system, the iPSC/2 system achieves four times the performance for twice the price, and achieves that capability in the same form factor as the first generation machine.

III. Hypercube Market

A. Segmentation

A recent research report published by OVUM, Ltd. segments parallel computing into two primary categories: shared memory architectures they call "Farms", and distributed memory architectures they call "Cubes". Their projection indicates that shared memory machines will dominate parallel computer sales into 1990, beyond which distributed memory machines will take over. This view is strongly based on the belief that automatic methods will be available for efficiently "parallelizing" applications for distributed memory machines toward the end of the 1980's, and the superior price-performance of these machines will result in a wholesale switch to the new technology.

The market segments tracked by OVUM include Supercomputers, Minisupercomputers, Minicomputers, Workstations, and Symbolic Processors. Shared memory architectures dominate minicomputer products and are a segment for which hypercubes make no inroads. Hypercubes have a strong position in all other markets, and are the exclusive domain of workstations.

B. Market Size

The market segments that are most closely tracked by the products of Intel Scientific Computers are the Supercomputer, Minisupercomputer and Symbolic Processor segments. Here, we are defining the segment by the functionality of the product, not its price. Supercomputer capability is represented by the iPSC/2 VX systems. These provide supercomputer level performance for about a tenth the price of traditional supercomputers.

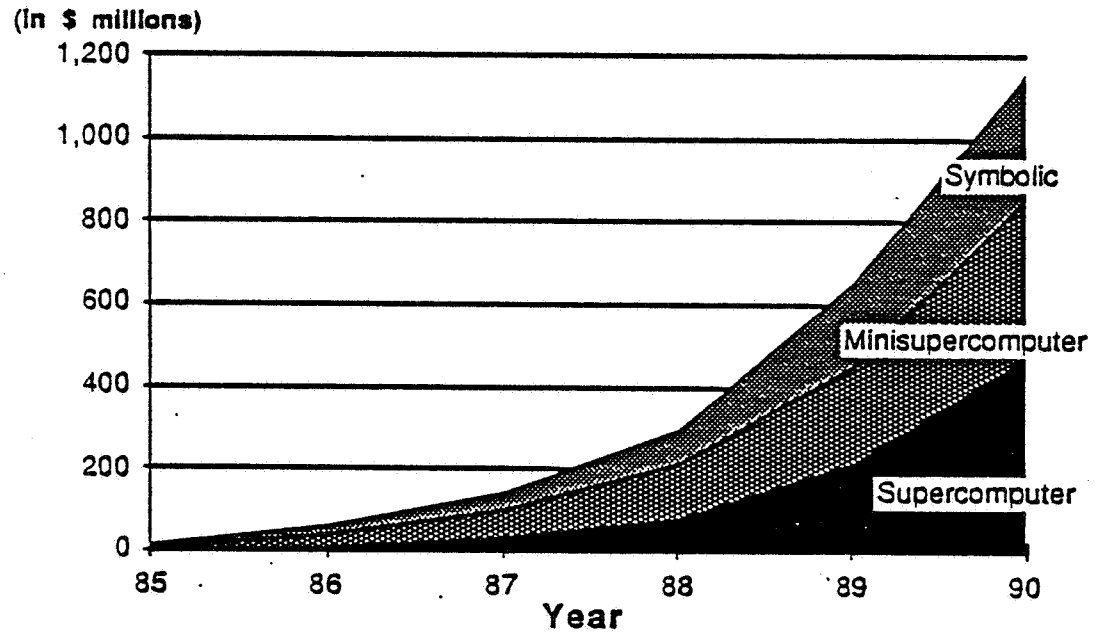
The Minisupercomputer segment is represented by the VX version of the SugarCube, a small desk top or floor-mount version of the iPSC family that supports up to 8 nodes. The 26 Mflop double precision peak performance of the system is comparable to that of current minisupers, but price is only about a tenth.

Symbolic processing is supported on all models of the iPSC/2 system and is supported by extended memory options available on these machines. The

advantage of concurrent symbolic processing is that performance exceeding all other computational alternatives is possible at low cost.

Figure 3-1 shows OVUM's growth of hypercube sales for the three segments of Supercomputer, Minisuper and Symbolic. These figures may be optimistic, but we believe most of the underlying assumptions are sound. For instance, by Intel's estimate, world-wide sales for distributed memory machines in 1986 was about one third of OVUM's projection. We would also contend that the growth will be more moderate, approaching a compounded annual average growth of about 50% over the next 5 years.

This technology will take a while to take hold, and initial introduction will be through production applications that give these machines supercomputer performance for a substantially lower investment. This option will be particularly attractive to supercomputer time share users. By owning their own



Source: Parallel Processing, the challenge of new computer architectures, OVUM, Ltd., 1988

Figure 3-1 Hypercube Market Sales Projection

hypercube machine, they will benefit from interactive computing, faster turn around of computational jobs, and the ability to optimize the application and architecture for the specific application.

Eventually, distributed memory architectures will reach commodity status, but

today we have only a hint of the ways in which supercomputer power on the desktop can change the way we work.

printed in U.S.A. 8/87

ISC

Intel Scientific Computers

Intel iPSC™ System
Applications & Programming Tools

APPLICATION BRIEFS

• **The Mandelbrot Set**	The fundamental mathematics of chaos and turbulence is perfectly parallel and supercomputer fast on the iPSC-VX™.
• **Solving The Wave Equation**	Efficient concurrent formulation of a classic 2-D wave equation solver on the iPSC™.
• **LINPACK and EISPACK**	Numerically stable, concurrent versions of these popular linear algebra packages.
• **Othello**	In this demo, the iPSC builds a gametree to find the best strategy for playing Othello™, a 2-person strategy game.
• **Seismic Modeling**	227 megaflop iPSC-VX/d5™ solution of an oil-industry seismic simulation problem.
• **Monte Carlo Simulation**	Supercomputer performance on the iPSC-VX™ from a perfectly parallel version of this powerful numeric method.
• **Multidimensional Fourier Transforms**	High performance 2D FFTs on the iPSC-VX™ despite worst case communication traffic.
• **Electromagnetic Scattering**	A 12-line change creates a concurrent version of a 1400-line sequential code using LINPACK.
• **Flat Concurrent Prolog (FCP) for the iPSC**	Concurrent programming language, based on logic programming, which can increase programmer productivity.
• **FCP: Matrix Multiply**	High-level process mapping capabilities of FCP.
• **FCP: Virtual Machine Demonstration**	Demonstrates 3 applications running simultaneously on 3 virtual machines.
• **Concurrent GESBT**	Expert system tool supporting rapid prototyping of cooperating expert systems.
• **MACE**	Concurrent object-oriented programming environment for developing large-scale AI applications on concurrent computers.
• **Naval Battle Management Simulation**	Uses concurrent GESBT as a tool for construction of multiple communities of cooperating expert systems.
• **Solving The N Queens Problem**	Illustrates the use of MACE on the iPSC™ to solve a classic computer science problem.
• **CrOS III**	The Crystalline Operating System (CrOS III), is a very fast, nearest neighbor communication system that offers: speed, portability, programming simplicity, and existing applications.

CONTINUED ON REVERSE SIDE OF PAGE

TECHNICAL NOTES	• **Vector Concurrent Computing**	A quick look at VX node performance for basic vector operations and matrix math routines.
	• **A Closer Look At Amdahl's Law**	Concurrent speedup is not limited by Amdahl's law when problem size grows with the number of processors.
	• **Performance of CCLISP™ on the Intel iPSC**	Speedup data shows the high performance potential of concurrent computers for AI applications.
	• **Triangle: A Concurrent Version of the Gabriel Benchmark**	Classic AI tree search problem shows speedup of 14.5X on a 16-node iPSC™ concurrent computer.
PRODUCT BRIEFS	• **Concurrent Debugger**	Convenient symbolic debugger which monitors and controls execution of a concurrent application on all processors.
	• **Simulator**	A program simulating the iPSC's parallel computing environment.
	• **VAST- 2™ FORTRAN Vectorizer**	An optimizing FORTRAN compiler simplifies vector *and* scalar programming for the iPSC-VX™
	• **iPSC-VX™ Vector Library (VecLib)**	The same program runs on either a standard or a vector iPSC™ system using VecLib, a common library of standard vector and mathematical operations.
	• **Nekton**	NEKTON™ is a state-of-the-art fluid dynamics and heat transfer numerical simulation package for the iPSC. Nektonics, Inc developed, markets, and supports this product.
	• **Concurrent Artificial Intelligence**	The SugarCube™, recently introduced by Intel Scientific Computers, provides an ideal system to support university education and research in concurrent AI.
	• **SugarCube, Introducing Concurrent Computation**	A complete system consisting of a concurrent computer, a multi-user development workstation, complete programming support for FORTRAN and C, and networking capability to UNIX™.
	• **Introduction To Concurrent Computing**	A description of a possible course in concurrent computing to be offered to graduate and advanced undergraduate students in science and engineering.
	• **Vector Concurrent System**	The vector SugarCube provides a solution to the need for a low-cost system to support research and education in parallel and vector computing techniques.
REFERENCE	• **Papers**	Two reference lists of papers: • Functional & Logic Programming with Large-Scale Concurrent Computers (side 1) and Distributed Memory Architectures for Artificial Intelligence (side 2). • Concurrent Applications on the Intel iPSC.

Intel Scientific Computers, 15201 NW Greenbrier Parkway, Beaverton, OR 97006 (503) 629-7629
Intel Scientific Computers, Intel International, Ltd. Pipers Way, Swindon SN3 IRJ England 011-44-793-875437

Application Brief
Intel Scientific Computers

The Mandelbrot Set on the Intel iPSC™

Bill Hughey
Intel Scientific Computers

PROBLEM DESCRIPTION

The underlying mathematical theory of the Mandelbrot Set, developed by Benoit B. Mandelbrot at IBM's Yorktown Research Center [1], is related to the mathematics of turbulence, chaos, and fractals, which are used to synthesize texture in computer graphics. Working with geometric forms, Mandelbrot developed a field known as "fractal geometry."

A computer program allows a user to view the boundary of the Mandelbrot set. In addition, the user can "zoom in" for a closer look at any magnification at any part of the set.

To define the Mandelbrot set, consider an arbitrary complex number c and the sequence of complex values z_k defined by:

$$z_0 = 0$$

$$z_{k+1} = z_k^2 + c$$

The sequence involves increasing powers of c,

$$0, c, c^2+c. (c^2+c)^2, \cdot \cdot \cdot$$

If c is outside the circle of radius 2, then the sequence rapidly diverges to complex infinity. But for certain distinguished points c in the complex plane, it turns out that the sequence remains bounded. This set is the Mandelbrot set. For example, $c = 1 + 0.286i$ is in the set, but $c = 1 + 0.287i$ is not.

THE iPSC SOLUTION

The interesting computer experiments come from points just outside the boundary of the set. The color graphics output screen is regarded as viewing a small section of the complex plan including a portion of the boundary of the Mandelbrot set. By definition, points outside the set lead to sequences which approach infinity. But how fast do these sequences diverge? This question leads to the following algorithm.

```
for each pixel on the screen do
begin
   let (x,y) be the coordinates of the pixel;
   let  c  be the complex number, c=x+iy;
   z := 0;
   cnt := 256;
   while (abs(z)<2) and (cnt>0) do
   begin
      z := z*z+c;
      cnt := cnt -1
   end
   set the pixel color to a value proportional to cnt
end
```

The screen displays a rectangular section of the complex plane. The initial section is (-1.25, 1.25) in the vertical direction and (-2.33333, 1) in the horizontal direction and contains the entire Mandelbrot set (which is colored black). The screen has 1,309,600 pixels (1280 x 1020). To compute the picture the screen is broken up into 170 strips, each with 1,280 x 6 pixels. The host sends strips to the nodes and the nodes run the algorithm described above. After finishing a strip, a node sends the result to the host and receives a new strip. On arrival at the host, each new strip is displayed on the screen.

After a picture is finished, it is possible to zoom in. The cursor is moved to the desired location and a new region is recomputed and displayed on the whole screen. It is also possible to refine the current picture by doubling the iteration limit (and changing the color map).

The node program is written in assembly language and the inner loop takes nine (real) floating point operations. The green light on each node is on while the node is computing and the red light is on while it is communicating with the host. After a picture is completed, the total number of floating point operations needed to compute the picture and the megaflop rate are displayed on the host.

---

[1] A.K. Dewdney, *"A computer microscope zooms in for a look at the most complex object in mathematics"*, Computer Recreations, Scientific American, August, 1985

Application Brief
Intel Scientific Computers

Solving the Wave Equation on the Intel iPSC™

Cleve Moler & Peter Ross
Intel Scientific Computers

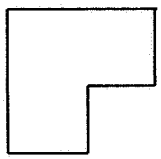
PROBLEM DESCRIPTION

The wave equation provides a mathematical model of a wide range of physical phenomena, including vibrations, resonances, and propagations of disturbances. This document describes one approach to solving a model wave problem using the Intel iPSC™, a hypercube-connected multiprocessor. It is possible to allocate the work to be done among the processors in such a way that the speed of the computation is determined by the speed of the graphics output device.

The differential equation governing waves is:

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$$

The model problem involves a 2-dimension region, L, formed from three unit squares,



The problem requires $u(x,y,t) = 0$ on the boundary of L. The interest in this particular problem derives from the interior corner, which leads to singularities in the derivatives of u and low accuracy from conventional finite difference and finite element methods.

The mathematical basis of our solution method [1] is an eigenfunction expansion,

$$u(x,y,t) = \sum_{k=1}^{n} a_k \sin(\mu_k t) v_k(x,y)$$

This form of the solution vanishes at $t = 0$, thereby satisfying one initial condition. The second initial condition, a specified initial velocity, determines the coefficients a_k.

The eigenvalues μ_k and eigenfunctions $v_k(x,y)$ are characteristic of the region L. The eigenfunctions can be accurately approximated with an expansion involving polar coordinates (r,θ) and fractional order Bessel functions $J_\alpha(r)$,

$$v_k(x,y) = \sum_{j=1}^{m} c_j^k J_{a_j}(\mu_k r) \sin(a_j \theta)$$

The quantities a_j are taken to be multiples of 2/3, so that the eigenfunctions satisfy the boundary conditions on the two sides of L that form the interior corner and have the correct singular behavior near the corner. The eigenvalues μ_k and the expansion coefficients c_j^k are determined by a least-squares algorithm which specifies that the eigenfunctions nearly vanish on the remainder of the boundary.

---

iPSC is a trademark of Intel Corporation

This approach always produces an analytic function $u(x,y,t)$ which is an exact solution of the wave equation, defined for all values of all its arguments. The only two discretization parameters are the upper summation limits, m and n. They determine the accuracy of the approximations to boundary conditions and initial conditions, respectively.

THE iPSC SOLUTION

The implementation of this method on the Intel iPSC involves four stages and illustrates several ways in which the portions of an algorithm can be allocated among the processors in a multiprocessor system. The output of the demonstration program is a graphical display of a moving projection of the solution.

Stage		
1	**The Eigenfunctions**	The first stage involves only the geometry of the domain L and n of the processors, where n is the number of eigenfunctions in the approximation of the solution. Processor number k computers the eignenvalue μ_k and the coefficients c_j^k The computation uses subroutines DQRDC and DTRSL from [2] and subroutine FMIN from [3] to find null vectors of rank deficient matrices whose entries involve Bessel functions evaluated on the boundary of L. Each processor saves its individual results for use in the later stages.
2	**The Initial Conditions**	A particular point in L is specified as the origin of an impulse that determines the initial velocity. Processor k computes the coefficient a_k determined by that impulse. These coefficients are assembled into a vector of coefficients which is passed to all the processors.
3	**The Display Grid**	Each processor is responsible for a portion of a display grid covering the region. The portions are allocated so that all processors have roughly the same number of display grid points. In the third stage, each of the processors evaluates all n eigenfunctions at each of its grid points. The processors communicate with each other to obtain the required coefficients. Each processor saves the values associated with its display grid points.
4	**The Time Steps**	The host computer sends a value of t to all processors. Each processor computes a linear combination of the eigenfunction values saved in Stage 3. The coefficients in the linear combination depend upon t. The result is the set of values of the solution $u(x,y,t)$ for the grid points owned by that processor. These results are then assembled into a single list of values for the entire domain which is sent back to the host to be displayed. This fourth stage is repeated for successive values of t to produce the successive frames for the moving display.

References:

[1] L. Fox, P. Henrici and C. Moler, *Approximations and bounds for eigenvalues of elliptic operators*, SIAM J. Num. Anal., pp. 89-102, 1967.

[2] J. J. Dongarra, J. R. Bunch, C. B. Moler and G. W. Stewart, *LINPACK Users' Guide*, SIAM Publications, 368 pp., 1979.

[3] G. E. Forsythe, M. A. Malcolm and C. B. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, 288 pp., 1977.

Application Brief
Intel Scientific Computers

LINPACK and EISPACK on the Intel iPSC™

Roger Golliver, Bill Hughey, & Cleve Moler
Intel Scientific Computers

LINPACK [1] is a library of FORTRAN subroutines for analyzing and solving systems of linear algebraic equations and least squares problems involving dense, stored matrices. EISPACK [2], [3] is library of FORTRAN subroutines for solving various matrix eigenvalue problems.

One way in which matrix computations can be done on the iPSC™ is to use the standard, sequential LINPACK and EISPACK subroutines simultaneously on each of the individual processors. The parallelism is obtained by solving many different matrix problems at the same time, with no communication between the processors. The memory available on the standard model of the iPSC provides storage for matrices of order up to about 180 on each processor.

Larger matrix computations can be carried out on the iPSC by distributing the data and the arithmetic across many processors [4]. The algorithms in LINPACK are organized so that almost all the innermost loops access the columns, rather than the rows, of the matrices. A frequent operation involves adding scalar multiples of one column in a matrix to each of the other columns. This column orientation is carried over to the parallel generalizations of the algorithms. A matrix of order n is distributed over p processors by storing n/p columns in each processor. If n is not exactly divisible by p, then some processors get one more column than others. A double precision matrix of order 1000 can be stored on 32 processors, for example, with 32 columns in the first 8 processors and 31 columns in the remaining 24 processors. A matrix of order 2000 can be stored on 128 processors. The iPSC-VX has over twice as much memory per node, and hence can handle a matrix of order 1000 on 16 processors.

A typical LINPACK or EISPACK factorization algorithm has an outer loop on an index, say k, which goes across the columns of the matrix. In the k-th step of the parallel generalization, the processor which holds the k-th column serves as the root processor for that step. The root processor does a short computation involving only the k-th column. The result of that computation is broadcast to all the other processors, using a "spanning tree" communication pattern that makes effective use of the hypercube interconnection between the processors. Each processor then uses this new data to modify its own columns. As the outer loop proceeds, different processors serve as root at different steps.

For large problems, most of the time in these parallel algorithms is spent doing arithmetic; only a small portion of the time is spent waiting for the root processor or participating in the communication. The idle time and communication costs are asymptomatically negligible - as the order of the matrix is increased, the portion of processor time attributable to overhead goes to zero.

This column oriented approach to parallel matrix computation has several significant advantages:

- The parallel algorithms retain the numerical stability properties of the underlying sequential algorithms.
- Available software can be readily modified to implement the parallel algorithms.
- Larger problems can be solved by increasing the number of processors, or the memory on the individual processors.
- Operations on entire columns provide long vectors for efficient use of the VX processor.
- Communication and load balancing overheads are asymptomatically negligible as problem size increases.

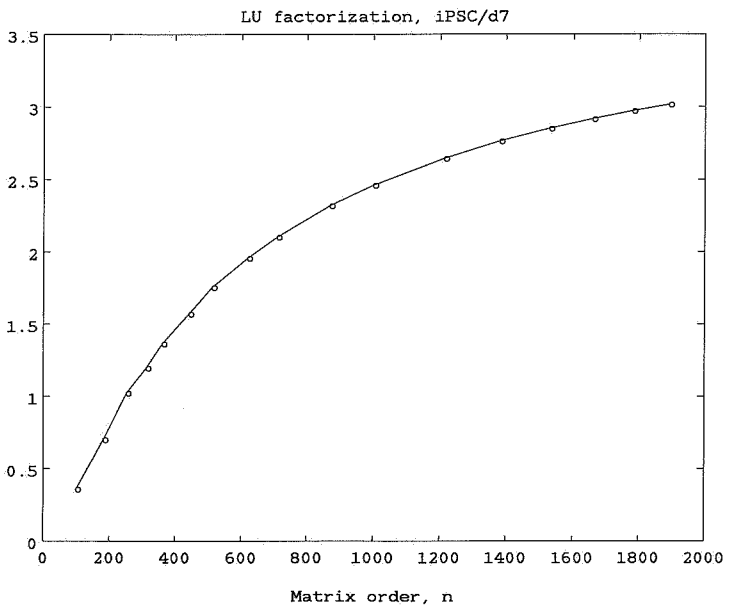
iPSC PERFORMANCE

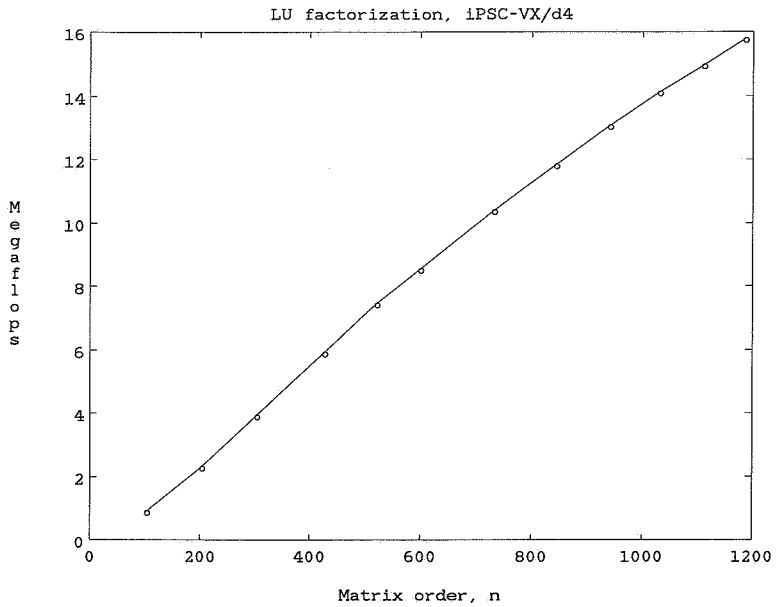
The most important subroutine in LINPACK is DGEFA, which uses Gaussian elimination with partial pivoting to compute the LU factorization of a double precision, general matrix. The parallel generalization of this routine uses exactly the same algorithm on a matrix distributed by columns across the processors. It has the same roundoff error and accuracy properties. Its performance, measured in megaflops, on two different models of the iPSC is shown in the two graphs.

The first graph is for an iPSC/d7, which has 128 processors using 80287 math coprocessors. For the inner loop of DGEFA, which is the vector operation DAXPY, this coprocessor can approach 0.03 megaflops. For very large problems, 128 processors approach 128 times 0.03, or about 3.8 megaflops. The available memory limits the problem size to about $n = 2000$. The performance obtained is a little over 3 megaflops, which is over 75% of peak performance.

The second graph is for an iPSC-VX/d4, which has 16 processors and a vector floating point unit associated with each processor. A single VX processor would approach 2.66 megaflops on very long DAXPY's. As the problem size is increased to $n = 1200$, which is the maximum allowed by current memory size, the processing rate approaches 16 megaflops for 16 nodes, or about 37% of peak performance. This percentage will be increased by adding more memory per node and by improving the performance of the hypercube message passing hardware.

Jack Dongarra of Argonne National Laboratory has continued the collection of the performance data originated in the LINPACK Users' Guide and periodically issues a technical report which has come to be known as "The LINPACK Benchmark." Dongarra's data comes primarily from running unaltered versions of DGEFA and its companion DGESL. Our performance results do not quite belong in Dongarra's collection because we have altered the source code to incorporate message passing.





References:

[1] J. J. Dongarra, J. R. Bunch, C. B. Moler and G. W. Stewart, *LINPACK Users' Guide*, SIAM Publications, 368 pp., 1979.

[2] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema and C. B. Moler, *Matrix Eigensystem Routines -- EISPACK User's Guide*, Springer-Verlag, Lecture Notes in Computer Science No. 6, Second Edition, 552 pp., 1976.

[3] B. S. Garbow, J. M. Boyle, J. J. Dongarra and C. B. Moler, *Matrix Eigensystem Routines -- EISPACK Guide Extension*, Springer Verlag, Lecture Notes in Computer Science No. 51, 343 pp., 1977.

[4] Cleve Moler, "*Matrix Computation on Distributed Memory Multiprocessors*," Hypercube Multiprocessors 1986, (Michael Heath, editor), 181-195, SIAM, 1986.

Application Brief
Intel Scientific Computers

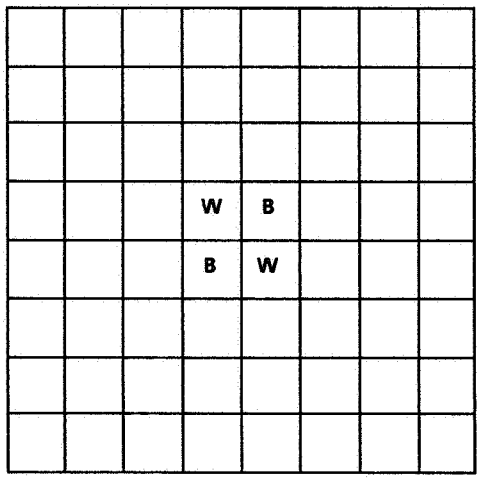
The Intel iPSC™ Plays Othello™

Joe Brandenburg & David Scott
Intel Scientific Computers

PROBLEM DESCRIPTION

Othello™ is a two person strategy game. It is derived from an earlier form of the game called "Reversi", which was played in England in the late 1800's. Two players...called black and white...place disks on an 8x8 grid to try and capture their opponent's disks and end up owning the most disks at the end of the game.

The initial position is as follows:



Rules of the Game

1. Black moves first.
2. Players alternate moving.
3. If a player has no move available, he loses his turn.
4. If neither player can move, the game is over.
5. When the game is over, the player with the most disks wins.
6. To make a move, a player must make a capture.
7. A capture occurs when the disk played makes a straight line (horizontally, vertically, or diagonally) with one or more enemy disks and a friendly disk at the other end (without any intervening spaces). All of the intervening enemy disks are captured (turned over) and become friendly disks.
8. All possible captures created by a play must be taken.

Playing The Game on The iPSC

All input to the game is controlled by using a cursor as follows:

- Move the cursor to the desired location on the screen by using the numeric keypad.
- Press ENTER when the cursor is at the desired location.

In addition to locating moves, the cursor is also used to choose a color and choose a search depth for the iPSC™. Instead of moving, a player can quite the game and can also display the iPSC's evaluation of the current position.

Strategy Hints

1. Corners are very valuable because they can never be captured.

2. Playing next to the corner is dangerous because this may open a pathway to the corner for your opponent.

3. To force your opponent to play next to the corner, you must eliminate all of his other available moves.

THE iPSC SOLUTION

As in all game playing programs, the iPSC creates a gametree of possible continuations and then searches the tree to determine the best move. Because the search must be truncated, an evaluation function is used to determine the value of the leaf nodes of the tree. These values are then propagated back up the tree. Alpha-beta pruning is used to reduce the number of branches which must be examined.

To make use of all of the nodes on the iPSC, it is necessary to distribute the search of the gametree. This was done by making node 0 the "supervisor" and all of the other nodes "workers." Node 0 performs all of the communiction with the host and maintains the highest part of the game tree. It also sends positions to the workers for either building or solving.

Building simply computes all legal moves and returns the list to node 0. Solving expands the position to a specified depth (given by the player at the beginning of the game) in order to determine its value and then returns this value to node 0.

The green light on a node is on if the node is computing and the red light is on if the node is waiting for a message.

Application Brief
Intel Scientific Computers

Seismic Modeling on the Intel iPSC™

Geoff Chesshire, Cleve Moler, Peter Ross, and David Scott
Intel Scientific Computers

PROBLEM DESCRIPTION

Seismic simulation, or forward seismic modeling as it is often called, is used by the oil industry to evaluate the accuracy of geophysical interpretations derived from processed field seismic data. Because there is no absolute way of determining the true structure of the substrate, indirect means are used. Information is combined from a variety of sources, including: geological data on the prehistoric formation of the region, gross structures determined from gravitational measurements, and, finally, low resolution images derived from seismic surveys (soundings) of the subsurface. The combined interpretation produces a hypothetical subsurface structure. This interpretation is expressed mathematically by specifying how physical properties, especially density and sound speed, vary in a 2-dimensional slice or a 3-dimension subsurface region.

The seismic modeling problem then consists of solving the acoustic wave equation corresponding to the assumed properties and comparing the solution with actual field recorded data. A high degree of correlation gives a high confidence that the interpretation is correct.

If $\rho(x,z)$ and $c(x,z)$ are the hypothetical density and sound speed, respectively, then the 2-dimensional version of the wave equation is:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \rho \left(\frac{\partial}{\partial x} \left(\frac{1}{\rho} \frac{\partial u}{\partial x} \right) + \frac{\partial}{\partial z} \left(\frac{1}{\rho} \frac{\partial u}{\partial z} \right) \right)$$

Boundary conditions are obtained from spatial symmetry, periodicity, or absorption assumptions at the edges of the region. The initial conditions model the pulse generated by a dynamite or air gun blast at a source point near the surface. The solution is to be sampled at several specified receiver points and compared with the field data.

THE SOLUTION

The solution method uses finite differences and explicit time marching. An n-by-n grid covers the entire region. The computation is distributed across p processors by setting $m = n/p$ and assigning an n-by-m vertical strip to each processor. Typical values are $n = 800$ and $p = 16$, so each processor handles an 800 x 25 grid. The simplest approximation takes $\rho \equiv 1$, leaves c as a function of x and z, and uses second order differences in t, x and z.

Three n-by-m arrays are stored on each processor. The indices correspond to the entire range of the vertical coordinate, z, and to a portion of the horizontal coordinate, x, associated with a processor-dependent offset, s. Because $c(x,y)$ does not depend upon t, the values in one of the arrays stay fixed during the computation. Assume $\Delta x = \Delta z$, let $\lambda = \Delta t/\Delta x$ and compute:

$$\gamma_{ij} = (\lambda\, c\,(j\Delta x + s, i\Delta z))^2$$

The values in the other two arrays vary with the time step. At the k-th time step, the approximation to the solution at the previous time is:

$$v_{i,j} \approx u\,(j\Delta x + s, i\Delta z, (k-1)\,\Delta t)$$

and the approximation to the solution at the current time is:

$$u_{i,j} \approx u\,(j\Delta x + s, i\Delta z, k\Delta t)$$

In one time step v is updated to approximate the solution at $(k+1)\,\Delta t$. The difference formulas require data stored in other processors, so each time step begins by swapping "shadow columns" of the u array between pairs of neighboring processors. For each processor, the entire time step (except for some details involving boundary conditions) is:

```
send column 1 of u to left neighbor
send column m of u to right neighbor
recv column 0 of u from left neighbor
recv column m+1 of u from right neighbor
t=t+Δt
for j=1 to m do
  for i=1 to n do
    v[i,j] = 2u[i,j] - v[i,j] + Y[i,j] (u[i+1,j] + u[i-1,j] + u[i,j+1] +u[i,j-1] - 4u[i,j])
  end i loop
end j loop
```

Several refinements of this basic algorithm are possible. Variable $\rho(x,y)$ and higher order difference schemes increase the accuracy. The message passing can be overlapped with the computation. Total execution time is reduced by "domain trimming", which limits the ranges on the i and j loops for small values of t to regions known to have non-zero values of u, and for large values of t to regions known to influence the results at the receivers. When domain trimming is used, concurrent load balancing is improved by making each processor responsible for more than one strip.

iPSC-VX PERFORMANCE

This problem is particularly well suited to parallel/vector computation. The amount of data that must be passed between processors in each time step is fairly small compared to the amount of computation required for the step. The inner loop vectorizes nicely, and the vectors involved are long enough to provide effective utilization of vector processors.

With the iPSC-VX machine, it is possible to go beyond vectorization. The entire inner loop, which accesses portions of three arrays and updates one column of v, can be regarded as a kernel operation. It is worthwhile to consider it a differencing operation, rather than a vector operation. For grid size n, the loop involves $5n$ reads from memory, $3n$ multiplications, $6n$ additions, and n stores to memory. The performance of a custom microcode routine implementing this kernel in single precision approaches 8 megaflops per processor.

On the iPSC-VX/d5 with 32 processors, a problem with a 1344-by-1344 grid takes 0.0715 seconds for an untrimmed time step, including communication. At $9n^2$ floating point operations per step, this corresponds to 227 single-precision megaflops.

Application Brief

Intel Scientific Computers

Monte Carlo Simulation With The iPSC-VX™

Ray Asbury
Intel Scientific Computers

THE PROBLEM

Monte Carlo techniques represent an important aspect of scientific computation. These techniques are typified by taking a random distribution of starting points for a calculation and analyzing statistically all of the different answers from these starting points. Monte Carlo is used in computational chemistry, atomic structure, and radiation scattering problems, to name a few.

The iPSC™ Monte Carlo demonstration uses a calculation of the energy of a helium atom to show performance boosts via parallel architecture. In addition, it demonstrates the potential of the high-speed numeric vector processor.

BENCHMARK RESULTS

Processes	Groups/ Process	Moves	Total Groups	Group Moves (1000's)	iPSC-VX (seconds)	Cray 1™ (seconds)
8	100	10	800	8	12.16	4.6
8	100	50	800	40	57.82	27.98
8	100	100	800	80	114.32	45.96
8	100	200	800	160	227.92	91.73
16	100	200	1600	320	228.56	183.0*
32	100	200	3200	640	230.04	367.0*
32	25	200	800	160	59.88	91.73

THE iPSC SOLUTION

The helium problem takes an ensemble of 100 groups, each group made up of 128 atoms, and calculates the average energy of each group. It then averages the energy of all the groups to obtain an answer. The calculation for the energy of each group is as follows:

- For each of the 128 atoms, generate random coordinates for the two electrons and calculate an energy and value for the wave equation, Ψ.

- Then, for each of the 128 atoms, move each electron a small random distance and repeat the energy and Ψ calculation.

- The new energy is accepted as better if the probability function $(\Psi_{new}/\Psi_{old})^2$ is greater than some random number between 0 and 1. This means that the new energy will always be accepted if Ψ_{new} is larger than Ψ_{old}, and randomly if not.

- This move and accept/reject is then continued for a user specified number of times. The average energy of each of the 128 atoms at the end of these calculations is reported as the group energy.

The accuracy of Monte Carlo techniques relies on large samples of calculations. While the individual calculations are independent of one another, on a serial machine each calculation must, of course, follow the previous one. Because the full problem solution requires many of these individual calculations, large chunks of CPU time can be used up on a serial machine. On a parallel machine, the individual calculations can be done independently and simultaneously on each of the processors in the machine. This is an example of what we call a "perfectly parallel" application. If there were enough processors to match each of the individual calculations, the entire problem could be completed in the time it takes for one processor to complete one calculation.

The input to this problem consists of the number of times the electrons will be moved, the number of groups per process, and the number of processors which will work on the problem. For a given number of moves, the important factor is the number of processors times the number of groups.

PERFORMANCE RESULTS

This benchmark has been run on a Cray 1S and on the iPSC/d5-VX (see table on reverse side of this sheet). The measured results show that for different problem configurations, the iPSC-VX can run from 0.3 to 1.5 the performance of the Cray. Doubling the size of the machine would double this performance.

Application Brief
Intel Scientific Computers

Multidimensional Fourier Transforms on the Intel iPSC-VX™

Cleve Moler and David Scott
Intel Scientific Computers

PROBLEM DESCRIPTION

Two and three dimensional discrete Fourier transforms provide excellent examples of both the potential and the challenge of the iPSC-VX™.

The input data for 1- or 2-dimensional problems form a complex n-vector or a complex n-by-n array, respectively. The best performance is obtained when n is a power of 2. Let $\omega = e^{-2\pi\iota/n}$ where $\iota = \sqrt{-1}$. Then the 1- and 2-dimensional, discrete transforms are:

$$\hat{y}_k = \sum_{i=0}^{n-1} \omega^{ik} y_i$$

and

$$\hat{y}_{k,l} = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \omega^{ik+jl} y_{i,j}$$

THE iPSC SOLUTION

The high performance floating point unit available at each processor of an iPSC-VX can calculate 1-dimensional FFT's very rapidly. For example, one 1024-point, single precision complex FFT can be done in 4.7 milliseconds. In order to take advantage of this performance, it is desirable for the 2-dimensional algorithm to do a number of complete 1-dimensional transforms on each processor. Consequently, the data are initially distributed by strips, with each of the p processors holding $m = n/p$ rows of the complex array, y.

Stage 1 In the first stage of the algorithm, each processor does m independent 1-dimensional transforms of size n. This stage is "perfectly parallel" - there is no communication between the multiple processors.

Stage 2 The second stage of the algorithm redistributes the partially transformed data so that it is stored by columns. This stage, which is essentially a multiprocessor, in-place matrix transposition, consists entirely of data rearrangement and message passing between processors. There are no arithmetic operations. Moreover, it is an extreme case of message passing in which every processor must send a different message to every other processor.

Stage 3 This stage is a repeat of the first, with each processor doing m more 1-dimensional transforms of size n. This leaves the transform, y, distributed by columns.

PERFORMANCE

With n = 1024 and single precision complex arithmetic, the calculation requires 8 megabytes of storage. This can be distributed across the 16 processors of the iPSC-VX/d4 with 64 rows of data using 1/2 megabyte in each processor. The current model of the iPSC-VX/d4 does the first and third stages in about 1/3 of a second each. This corresponds to over 10 million floating point operations per second for each processor, or over 160 megaflops total, for the arithmetic operations alone.

In the communication phase, each processor sends a message to each of the other processors. In the current design, a message not directed to a nearest neighbor node must be received by an intermediate node and then re-transmitted to its destination (store and forward). This contention in the hypercube reduces the effective megaflop rate for the complete transform to a little over 17 megaflops. Improvements in the message passing hardware currently under development will eliminate the need to perform the store and forward step. This is expected to reduce the second stage time to something comparable with the first and third stages, thereby boosting the effective rate to over 100 megaflops.

A problem with $n = 2048$ could be distributed across the 64 processors of an iPSC-VX/d6. Each processor would have only half as many rows as the $n = 1024$ problem, but each row would be twice as long. The complexity of the FFT algorithm grows as $n \log n$, so the time required for each of the arithmetic stages should increase by a factor of just $\log 2^{11} / \log 2^{10} = 1.1$. With improved hardware, the communication stage should also still take about 1/3 of a second. Thus, a problem with four times as much data and slightly more than four times as many arithmetic operations can be done with four times as many processors in about the same total time. The effective arithmetic rate would be near 400 megaflops.

Similar considerations apply to 3-dimensional problems. Each of the processors would do a number of independent 2-dimensional transforms, then a single communication stage would rearrange the data so that the transform in the third dimension could be done. For example, it should be possible to do a 128-by-128-by-128 problem on a VX/d5 with the above communication hardware improvement in about one second.

Application Brief
Intel Scientific Computers

Porting an Electromagnetic Scattering Code to the Intel iPSC™

Cleve Moler and David Scott
Intel Scientific Computers

PROBLEM DESCRIPTION

Professor Don Wilton, of the University of Houston, together with several colleagues and students, has developed a FORTRAN program for calculating the electromagnetic scattering and radar cross section of objects of arbitrary shape in three dimensions [1]. The program finds approximate solutions to Maxwell's equations using a finite element method applied to the electric field integral equation.

The surface of an object is modeled with planar triangular patches. A complex-valued electric current is associated with each edge in the collection of triangles. These currents, i, are related to each other by a system of simultaneous linear equations generalizing Ohm's Law.

$$Z\,i = e$$

Here e is a vector computed from an externally imposed electric field and Z is a large, dense, complex matrix whose elements are obtained by numerical quadrature of functions involving the distances between all possible pairs of surface triangles. After this equation has been solved for i, far-field quantities - such as radiation patterns and radar cross sections - can be computed by taking the inner products of i with a number of independent vectors, q_k. This is essentially a large matrix-vector product,

$$s = Q\ i$$

The number of unknowns, n, is equal to the number of distinct interior triangular edges, which is approximately 3/2 the number of triangles. Values of n in the range of a few thousand are needed to accurately model complicated objects. Solution of a non-sparse, non-symmetric, complex linear system of order 2000 requires over 32 megabytes of storage and over 20 trillion floating point operations.

PARALLEL PROGRAMS

The original program, intended for a conventional sequential computer, uses LINPACK subroutines CGEFA and CGESL to solve the complex linear system. Parallel generalizations of these subroutines are now available for the Intel iPSC™, so the question becomes:

How much effort is required to parallelize the rest of this program?

The program has four main sections. The time spent in each section increases at different rates with increasing n.

Step		Work	Parallelization
1.	Geometry initialization	$O(n)$	Don't bother
2.	Assemble Z and e	$O(n^2)$	How?
3.	Solve $Zi = e$	$O(n^3)$	Already done -- LINCUBE
4.	Compute Qi	$O(qn)$	Perfectly parallel

Step	Description
1. Geometry Initialization	This step takes so little time that the parallel program simply duplicates the calculation on each processor.
2. Matrix Assembly	Performing the second step, the matrix assembly, in parallel, requires some thought and analysis, but very little actual code change. The computation involves a double loop over the surface, examining all possible pairs of triangles. For each pair, several integrals are evaluated by numerical quadrature. Each of these integrals affects the 9 elements of Z that describe the interactions of the 3 edges on one triangle with the 3 edges of the other. LINCUBE expects Z to be distributed by columns, with column j stored on the processor (j-1) $mod\ p$. Consequently, a test is inserted near the beginning of the body of the double loop. Each processor computes only those integrals which affect its matrix elements.
3. Solve $Zi = e$	This step dominates the execution time as n increases. The call to the LINPACK equation solver in the original program is changed to a call to a LINCUBE version of the same subroutine. These subroutines are described in another Intel Scientific Computers Application Brief.
4. Far-Field Calculations	This step is almost trivial to convert to a parallel form, but the approach involved applies to many other algorithms. The main loop is:

```
      DO 499 I = 1, NFACES
      ...
  499 CONTINUE
```

This is changed to

```
      ID = MYNODE()
      P = 2**CUBEDIM()
      DO 499 I = ID+1, NFACES, P
        ...
  499 CONTINUE
      CALL GSUM ( ... )
```

The system functions **MYNODE** and **CUBEDIM** return the processor identification number and the hypercube dimension, so **P** is the number of processors. The loop body is executed for successive values of **I** on different processors and all processors execute the loop body the same number of times, plus or minus one. Finally, the library subroutine **GSUM** combines the quantities generated on the individual processors.

It took a few hours of discussion involving the various authors of different portions of the program to understand the modifications necessary to make it parallel, but the actual textual changes made involve only about a dozen lines out of roughly 1400 lines of code.

The performance of the program that resulted could still be improved. In the matrix assembly step, most of the integrals are computed by three different processors. This redundancy could be nearly eliminated by using a different edge numbering scheme. In addition, a rearrangement of some of the loops in the matrix assembly step would improve the vectorization within an individual processor.

---

[1] S. M. Rao, D. R. Wilton and A. W. Glisson, "Electromagnetic Scattering by Surfaces of Arbitrary Shape," *IEEE Trans. Antennas Propagation*, vol. AP-30, No. 3, 1982, pp. 409-418.

Application Brief
Intel Scientific Computers

Flat Concurrent Prolog (FCP) for the Intel iPSC™

Joe Brandenburg & David Billstrom
Intel Scientific Computers

INTRODUCTION

Flat Concurrent Prolog (FCP) is an experimental concurrent programming language which has been implemented on the Intel Personal Super Computer (iPSC™), the first commercial hypercube. FCP is a process-oriented, logic-based language. It is an implementation language for other concurrent logic languages such as Concurrent Prolog, Parlog, and Guarded Horn Clauses[1].

As in Prolog, a program is expressed as a set of Horn clauses. FCP clauses have the form:Where H is

$$H \leftarrow G \mid B$$

the head, G is the guard, "|" is the commit operator, and B is the body. FCP guards consist of simple test predicates.

FCP differs from Prolog in that there is no backtracking control strategy. Instead, FCP views each literal in a clause as a specification of a process which may execute concurrently. Two constructs exist for synchronization: a *guard* and *read-only* annotation. Guards enforce local constraints to limit the search for a solution. The read-only annotation provides data-flow synchronization. In contrast, Prolog uses a *cut* construct and textual ordering of clauses to control and constrain the search space of a computation. The organization and simplicity of FCP make it amenable to parallel execution on the hypercube and eases the task of writing parallel programs.

FCP has an efficient sequential implementation, comparable in speed to commercially available Prolog compilers. The practicality of the language has been demonstrated on a number of non-trivial programming tasks. These tasks include: a boot-strapping compiler, sections of an operating system, and a programming environment. Additionally, it has proved to be an effective tool for the design of parallel algorithms and systems.

TECHNICAL FEATURES

The prototype system currently available consists of two parts:

- An FCP interpreter which executes on the iPSC concurrent computer
- A programming environment called Logix

FCP Guards

The guard in an FCP clause contains simple predicates defined in the language. These predicates (for example, $integer(x), x > y$) are used for arithmetic comparison, data type inspection, and type manipulation. Disallowing complex user-defined predicates in the guard has two major implementation consequences. It eliminates the need for a hierarchical binding environment and the distribution of the commit operator.

FCP Communication and Synchronization

FCP implicitly provides methods for communication and synchronization. Communication is accomplished with either shared logical variables or streams. The latter is implemented using list structures. Data-flow synchronization is achieved by annotating variables to be read-only. If a process attempts to bind a read-only occurrence of a variable, the process suspends and waits until the variable is bound by another process.

Process Mapping and FCP

Simple techniques have been developed to map processes and code to processors. These separate the task of mapping the algorithm to a virtual machine and the task of mapping a virtual machine to the underlying physical machine.

Logix

Logix is a programming environment under which FCP programs can be developed and executed. The environment includes development tools (compiler, debugger), a module system for program organization, and various system services, such as a file system and terminal I/O.

ORDERING INFORMATION

The Flat Concurrent Prolog Interpreter and LOGIX Programming Environment are available now for the iPSC. Logix is also available for Sun and VAX™ computers. Please see the software release agreement for further details. The cost is for media and shipping expenses only. Complete the license agreement and send with $250.00 (U.S. dollars) to:

Steve Taylor
FCP Interpreter and LOGIX for the iPSC
The Weizmann Institute of Science
Rehovot 76100, Israel

---

1. Shapiro, E. *Concurrent Prolog: A Progress Report.* Computer. August, 1986, Vol. 19, No. 8, pp 44-58.

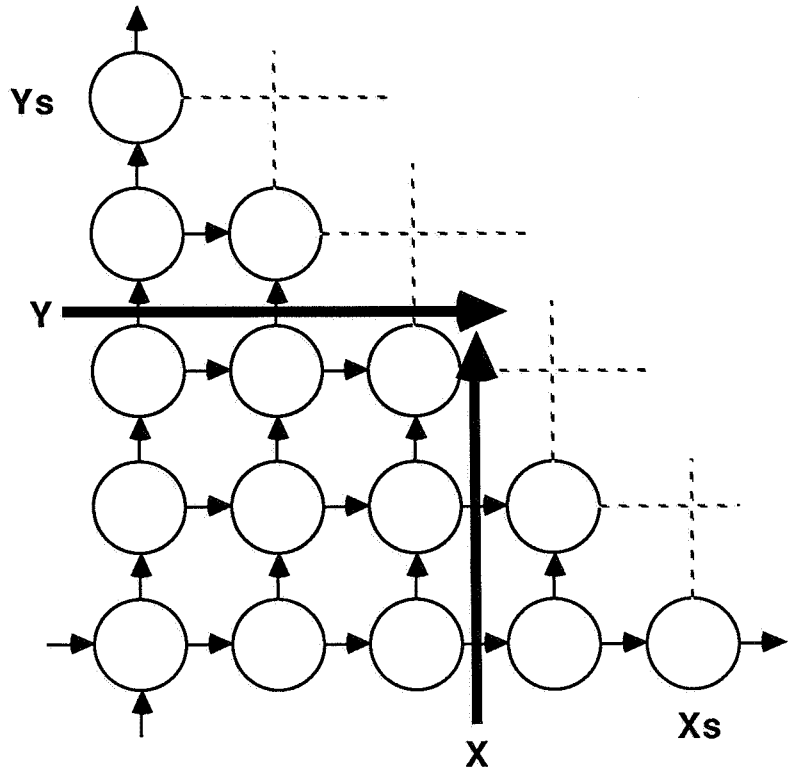
FCP: Matrix Multiply on the Intel iPSC™

Joe Brandenburg & David Billstrom
Intel Scientific Computers

The Flat Concurrent Prolog (FCP)[1] Matrix Multiplication is an example of a simple program distributed over a concurrent computer. The example demonstrates programming the Intel Personal Super Computer (iPSC™) with a high-level process mapping tool.

The program first establishes a mesh-connected (torus) virtual machine on the iPSC. Then, the elements of two N by N matrices, which are initially in a single processor, are distributed across all the processors. A product calculation is performed at each processor node and the data is contained in a distributed structure. By pipelining the distribution of the data, the program can achieve $O(_n)$ communication complexity. The computation at each node process is an inner product and thus is $O(_n)$.

The process mapping is achieved by a simple turtle-like programming notation[2] which dynamically moves processes between processor nodes. Streams between processes are used to transfer both data and other processes. In FCP, streams are denoted by the cons cell [X | Xs].



Process Structure For Matrix Multiply

iPSC is a trademark of Intel Corporation

The complete program for the virtual machine is only seven (FCP) clauses long. The program, shown below, is a simple high-level specification with relatively little concern for execution on a parallel machine.

For further discussion, refer to the paper by Taylor, Av-ron, and Shapiro.[3]

Actual FCP Code for Matrix Multiply

```
mm([Xv | Xm], Ym, [Zv | Zm]) <--                    % matrix multiply
    vm(Xv, Ym?, Ym1, Zv)@left,
    mm(Xm?, Ym1?, Zm)@fwd.
mm([], Ym, []).

vm(Xv, [Yv | Xm], [Yvl | Ym1], [Z | Zv]) <--        % vector multiply
    ip(Xv?, Xv1, Yv?, Yv1, Z),
    vm(Xv1?, Ym?, Ym1, Zv)@fwd.
vm(Xv, [], [], []).

ip(xv, Sv1, Yv, Yv1, Z) <-- ipl(Xv?, Xv1, Yv?, Yv1, 0, Z).   % inner product

ip1([X | Xv], [X | Xv1], [Y | Yv], [Y | Yv1], z0, z) <--
    Z1 := (X*Y) + Z0,
    ip1(Xv?, Xv1, Yv?, Yv1, Z1, Z)
o[1([], [], [], Z, Z).
```

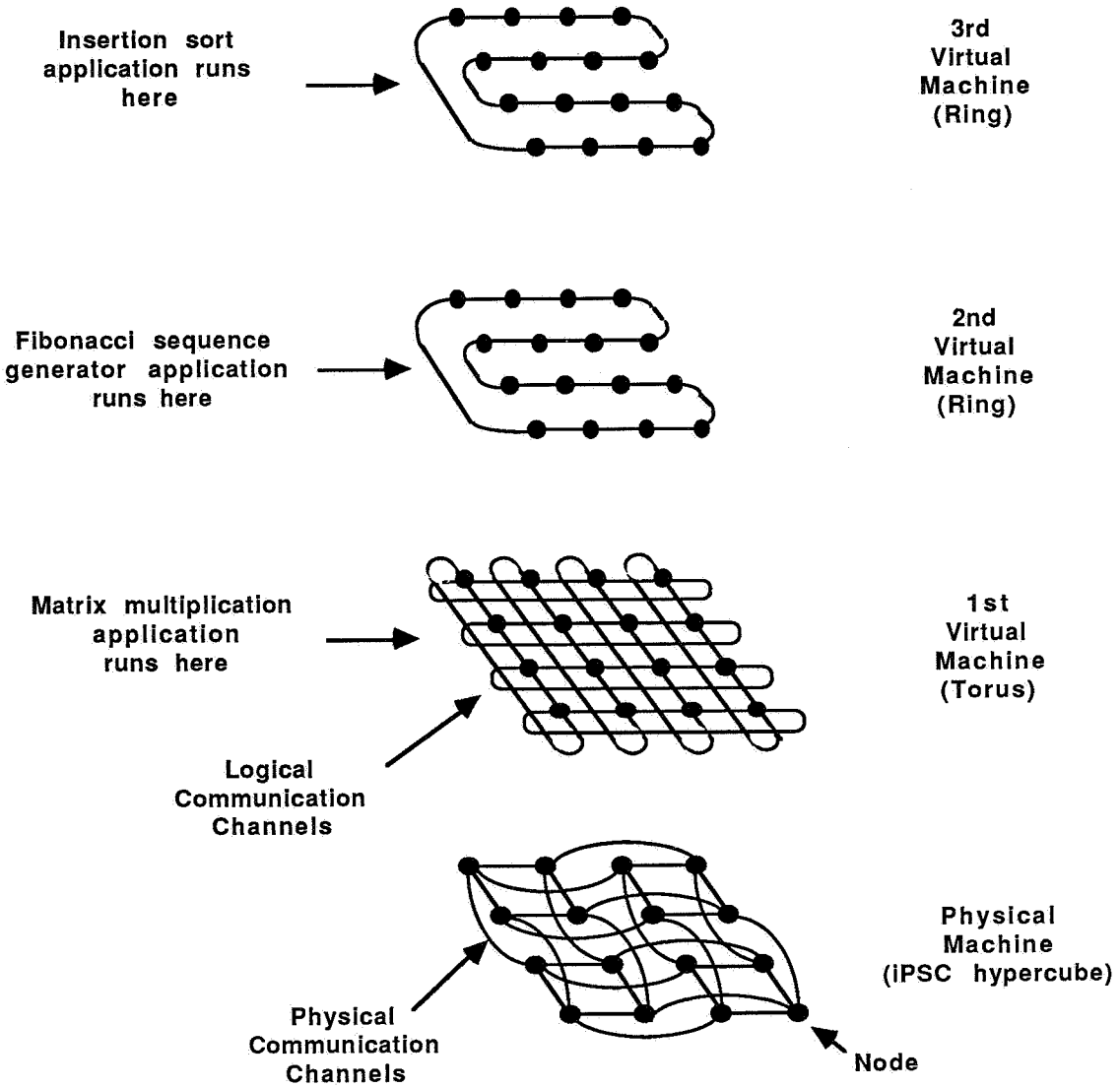
---

1. Mierowsky, Taylor, Shapiro, Levy, and Safra. *The Design and Implementation of Flat Concurrent Prolog*, Department of Computer Science, Weizmann Institute of Science, Rehovot Israel, Technical Report CS85-09; July, 1985.
2. Pappert. *Mindstorms: Children, Computers, and Powerful Ideas.* Basic Books, New York, New York; March 1980.
3. Taylor, Av-ron, and Shapiro. *A Layered Method for Process and Code Mapping.* Department of Computer Science, Weizmann Institute of Science, Rehovot Israel, Technical Report CS86-17; March, 1986.

FCP: Virtual Machine Demonstration on the Intel iPSC™

Joe Brandenburg & David Billstrom
Intel Scientific Computers

This program demonstrates the layered approach to process and code mapping. The program spawns three virtual machines and runs applications on each virtual machine. The illustration below shows the two types of virtual machines and the actual sequence of the demonstration.



The tasks of mapping an algorithm to a virtual machine and the mapping of the virtual machine to the physical machine can be separated using Flat Concurrent Prolog (FCP). The same technique can be used with actual code segments.

iPSC is a trademark of Intel Corporation

THE DEMONSTRATION

Initially, the FCP program creates a torus virtual machine. On this machine, a matrix multiply[1] is mapped and begins running. While the application is running on this virtual machine, a second virtual machine is spawned.

The second virtual machine is a ring architecture. Again, using FCP, a different application is initiated. This application is a Fibonacci sequence generator. While this application and virtual machine are executing, a third virtual machine is spawned.

The third virtual machine, which is also a ring architecture, begins executing an insertion sort application. This application is specifically designed to exercise short-circuit and incomplete message programming techniques.

All of the virtual machines run concurrently and applications or programs run concurrently on each virtual machine. The mapping of applications, or processes, to machines is accomplished by the use of FCP streams and a logo-like turtle graphics language.

RESULTS

This demonstration verifies the presence of a language, Flat Concurrent Prolog, for concurrent logic programming research. The notation enables relatively quick construction of prototype parallel algorithms, even when the physical machine does not directly support a problem's natural process model.

1. Refer to ISC Application Brief entitled, *"FCP: Matrix Multiply on the Intel iPSC."*

Application Brief

Intel Scientific Computers

Concurrent GESBT
Generic Expert System Building Tool

Artificial Intelligence and Decision Aids Division
Science Applications International Corporation

INTRODUCTION

The GESBT system is a tool for building simple expert systems. Concurrent GESBT is an enhanced version of GESBT that supports multiple, cooperating expert systems on a concurrent computer such as the iPSC™ system. The concurrent version offers a method of sharing objects and moving objects between multiple expert systems. It also enables an expert system to create another expert system. *Concurrent GESBT enables rapid prototyping of cooperating expert systems.*

GESBT

GESBT is a simple expert system building tool. It provides the necessary facilities for a LISP-literate computer user to prototype and develop expert systems using production rules. The rules supported are of the *if...then* format. The basic representation of data in the GESBT system is the *object*. All objects are named, and each has attributes and values. The only restriction on the number of objects, or their size, is the available physical memory. Values and attributes are unrestricted in type.

Rules are implemented as objects with special attributes: antecedent and consequent. The values of these attributes are unrestricted, which permits flexible and expressive rules. All the usual operators for logic and comparison are supported, included: and, or, not, =, <, and >. Free variables may be used in antecedents rather than explicitly named objects. This causes the system to instantiate the variable in all ways that cause the antecedent to be true. The user may use any Common LISP function or a user-defined Common LISP function within the antecedent. This allows considerable flexibility for the LISP-familiar programmer.

The consequent (then) is as unrestricted in form as the antecedent. The usual effect of a true antecedent is an *assert*. Assert modifies an object's attribute values. Objects may be created, other knowledge basis activated, and I/O to the user's keyboard initiated. As with the antecedent, users may supply their own Common LISP functions, which may refer to the instantiated variables of the antecedent.

PROGRAMMING MODEL

A *knowledge base* is a collection of objects and rules. GESBT allows more than one knowledge base to exist at a time and a knowledge base may be activated by another knowledge base to resolve some subset of the problem.

Each knowledge base is implemented as an object. As an object that may be activated, each carries several special attributes. The most important of these special attributes are the lists of rules and objects that make up the knowledge base. This representation enables an active object (a knowledge base) to create a new knowledge base object and then activate it. This is the basic facility required for intelligent systems with a capacity to "learn", e.g., create new intelligent systems based on current knowledge and interactive data.

GESBT DEVELOPMENT TOOLS

The GESBT system provides four windows...controlled by a mouse...for development. This comfortable user interface is designed to encourage rapid prototyping, and to integrate well with the Common LISP available from Gold Hill Computers for both the PC AT and the iPSC systems. GESBT runs on the PC AT.

CONCURRENT GESBT

Concurrent GESBT adds the facilities of communications and transparent global objects to collections of GESBT systems. These facilities are simple and straightforward so that existing GESBT expert systems may be quickly converted to Concurrent GESBT.

The programmer places each knowledge base on a processor node of the concurrent computer. Then, the Concurrent GESBT communication facilities are used to copy and move objects between the knowledge bases.

Concurrent GESBT adopts the notion of *global* and *local* objects. During development of the community of knowledge bases, the programmer specifies which objects are global. Global objects may be accessed from every knowledge base, on every processor node. This access is completely transparent to the user. However, accessing global objects on other nodes is relatively expensive compared to accessing objects resident locally. Thus, the use of global or local objects is important to the programmer's overall design. Global objects may use inheritance and inheritance is preserved transparently to the user.

Concurrent GESBT Functions

copy-object	place a copy of an object on a node
send-object	send object to a node; make it global
announce-object	make object global
grab-object	bring global object to this node
remote-defun	defines a function on a node
remote-load	causes a remote load of file to node

GESBT Functions (transparent to object location)

valof	return value of attribute
assert	update attribute/value pair
user-assert	user level assert
activate	invoke a knowledge base
dvalof	**valof** with inheritance
print-object	with inherited values, with dynamically evaluated attributes

--- partial list of GESBT functions ---

STATUS

Communities of expert systems have been implemented with Concurrent GESBT, such as a prototype battle management simulation. The strongest aspect of Concurrent GESBT is the simple structure which enables quick prototyping of cooperating expert systems.

AVAILABILITY

Science Applications International Corporation's (SAIC) Artificial Intelligence and Decision Aids Division provides government, industry, and military customers with decision-aiding systems incorporating artificial intelligence, decision analysis, operations research, and human factors engineering.

The Concurrent GESBT system requires CCLISP™, the Concurrent Common LISP for the iPSC concurrent computer. The iPSC is available from Intel Scientific Computers and CCLISP from Gold Hill Computers. Runtime versions of the Concurrent GESBT kernel are available to iPSC users at a nominal cost from SAIC and Gold Hill Computers. Source licenses are also available from SAIC. Please contact:

Artificial Intelligence and Decision Aids Division
Science Applications International Corporation (SAIC)
1710 Goodridge Drive
McLean, Virginia 22101

(703) 827-4857

Application Brief

Intel Scientific Computers

MACE (Multi-Agent Computing Environment)

Distributed Artificial Intelligence Group
Computer Science Department
University of Southern California

INTRODUCTION

The MACE (Multi-Agent Computing Environment) system[1] is a development and execution environment for distributed *agents*. Agents are intelligent entities, capable of performing simple tasks. The MACE system is both a tool set and a test bed for the programming model *cooperating agents*.

Distributed agents, under MACE, form the basic building blocks for a wide variety of symbolic parallel programming models. MACE is essentially a tool for programming concurrent computers in an object-oriented style. Already, demonstrations of programming models such as Contract Net[2], Distributed Blackboards[3], and Rule-Agents (Rule-based systems)[4] have been written in MACE.

THE PROGRAMMING MODEL

Agents themselves should take on much of the burden of actual system construction and operations. The task of the programmer should be to establish initial form, directions, and policy. Thus, programmers become managers of interdependent processes, rather than specifiers and implementors of entire systems in detail. They specify staffing requirements with job descriptions - high-level descriptions of the special and general requirements of agents. Agents are assumed to come with basic skills in their domain of expertise. For example, a quality-control agent might have knowledge about testing techniques, statistical quality control, error-discovery procedures, and corrective action.

Programmers direct and organize agents by specifying their relationships to one another and by issuing policies and goals - high-level directives and constraints on behavior. Agents carry out and interpret the policies and goals in the light of their own local circumstances. In this way, agents are problem solvers and systems are adaptive without explicit detailed control by a central actor or overseer.

MACE FOR THE iPSC

An initial goal of the MACE system was quick and immediate use of commercial concurrent computers, as they became available. To this end, the system was first implemented as a simulator, with an interpreter. MACE agents can be implemented on the simulator and statistics gathered on execution time, message routing, traffic loads, agent and processor breakdowns, timing, synchronization, and trace information.

Since the inception of the MACE simulator, the first commercial large-scale distributed memory, message-passing machine, the Intel Personal Super Computer (iPSC™) has become available. Originally available with C and FORTRAN, Concurrent Common LISP (CCLISP™) from Gold Hill Computers is now available. MACE was ported in two weeks from the iPSC system from its original hosts: a network of PC AT personal computers and a TI Explorer™ workstation.

The machine currently on site at USC's Distributed Artificial Intelligence Group laboratory is a 16-node iPSC with 4.5 megabytes of memory per node.

MACE SYSTEM - THE AGENTS

Agents are semi-autonomous and communicate via messages. An agent consists of local procedure(s) and data as well as an inference engine and list of attributes. Conceptually, agents resemble the data-types in ROSS[4], ACTORS[5], and Lisp FLAVORS[6]. The fundamental data-type in such high-level languages is frequently called the object. Agents are similar to *objects* but do not support inheritance.

Agents are implemented as property lists of the agent name. Therefore, only one copy of an agent may reside on a node of the iPSC concurrent computer at any given time. The complete definition of a simple agent is:

```
(agent <name>
      (local-function-defs <local function definitions>)
      (engine <engine definition>)
      (acquaintances <list of acquaintances>)
      (attributes
            ((behaviors <list of behavioral rules>)
             (<attribute> <value>)
             ...
             (<attribute> <value>))))
```

MACE SYSTEM - THE KERNEL

The execution environment for MACE is two or more MACE kernels, each resident on a processor in a network. The kernel supports message passing and schedules the execution of agents.

MACE kernels are resident on each node of the iPSC. A *user-kernel* resides on one of the nodes, *independent-kernels* reside on each of the other nodes. The user-kernel initializes MACE, constructs the system agents (that provide system services) and initializes global variables. It also provides the user interface.

System-agents construct *user-agents* from descriptions specified with the Agent Description Language. System-agents also monitor execution of agents, handle errors, and provide an interface to the user.

AVAILABILITY

The MACE system and various components is on-going research at the University of Southern California. For information, contact Dr. Les Gasser, Distributed Artificial Intelligence Group, Computer Science Department, University of Southern California, Los Angeles, California 90089-0782. (213) 743-7749.

---

1. Gasser, L., *MACE, A Multi-Agent Computing Environment.* USC-DPS Group, University of Southern California, March, 1986.
2. Smith, Reid. *The Contract Net Protocol: Communication and Control in a Distributed Problem Solver.* IEEE Transactions on Computers, C-29(12), December, 1980.
3. Lesser, V. and L. Erman. *Distributed Interpretation: A Model and Experiment.* IEEE Transactions on Computers, C29(12), December, 1980.
4. Gasser, L., and Tenorio, M. *Rule-Agents: A Distributed, Object-Oriented Approach to Production Systems Using MACE.* Working paper, USC-DPS Group, Computer Science Department, University of Southern California, March, 1986.
5. Gasser, L., C. Braganza, N. Herman, L. Liu. *MACE Reference Manual.* USC-DPS Group, University of Southern California, July, 1986.
6. McArthur, D. and P. Klahr. *The ROSS Language Manual.* Rand Corporation, 1982.
7. Hewitt, C. *Viewing Control Structures as Patterns of Passing Messages.* Artificial Intelligence, pp 323-364, 1977.
8. Weinreb, D., D.Moon, and R. Stallman. *Lisp Machine Manual.* Massachusetts Institute of Technology, 1983.

Application Brief
Intel Scientific Computers

Naval Battle Management Simulation with Concurrent GESBT on the Intel iPSC™

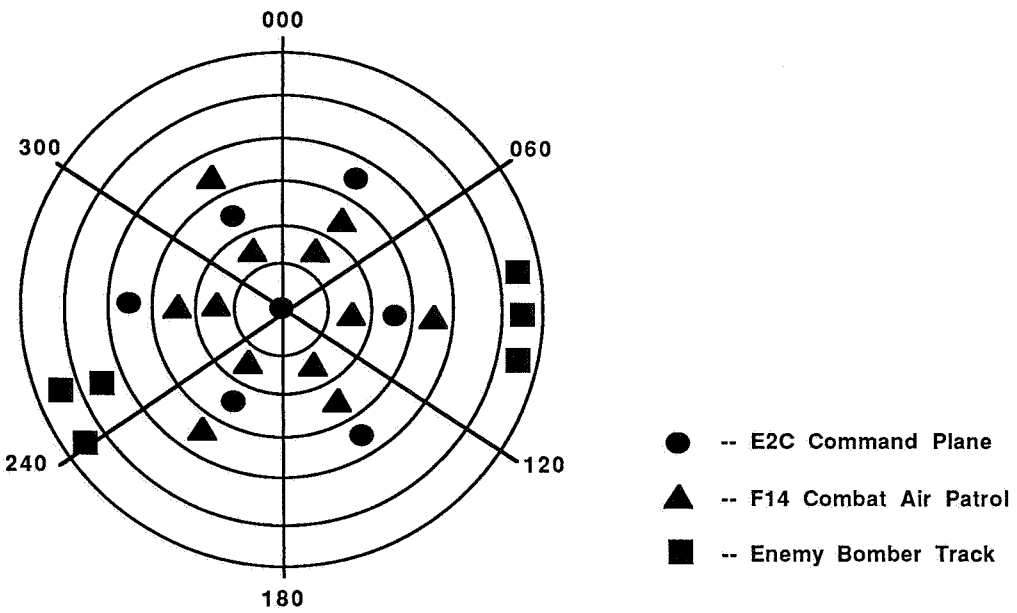
David Billstrom, Intel Scientific Computers
Michael Blanks, Science Applications Intenational Corporation

PROBLEM DESCRIPTION

This demonstration is a prototype of one part of a large-scale battle management simulation tool. Eventually the potential of a large-scale concurrent computer....the iPSC™....will be exploited by a multitude of communities of expert systems. This demonstration consists of one of these communities of expert systems. This hierarchy of expert systems may be appropriate for many large-scale expert systems applications, not just battle simulation.

The community of expert systems currently on the iPSC represents the battle management of an aircraft carrier battle group. The problem was deliberately chosen for its complexity. The complexity is due to the fact that the area covered by the battle group is too large for a single expert system to reasonably cover...as is the case with many potential expert system applications.

Therefore, the area about the battle group is divided into six sectors. An expert system conducts defense strategy for each sector, and communicates with the neighboring expert systems which conduct defense strategy for each of their sectors. An illustration of the graphic display of the defense sectors is shown below:



The area about the aircraft carrier group is divided into six sectors, each with 60° of the horizon and 600 miles radius from the ship. Each sector is commanded by a Naval E2C command plane. The defensive knowledge of the commander of each E2C aircraft is captured in a knowledge base. The knowledge base for each sector is identical. The knowledge base governs the defense of a sector but also communicates with other E2C commanders (knowledge bases) about threats and response strategy as well as borrowing combat air patrol aircraft (fighters) from those sectors. Threats are bombers, seeking to destroy the aircraft carrier and defense units are friendly fighter aircraft, each with a limited number of missiles.

THE SOLUTION

In the eventual simulation tool, many instances of the community of experts that represent the battle group will run on the iPSC. Each community will act on the exact same collection of enemy threats, but will use different defense strategies. Upon resolution of the simulated battle, each community will report back to a central monitor, which will record the relative success of the community against a set of threats. The central monitor reports which community answered the threat scenario in the optimal way. That particular defense strategy is then reported to the human battle commander.

The benefit of a simulation tool structured this way is the ability to run many different scenarios against the same situation at a speed much faster than real time. This offers the human a number of optimal choices before he or she needs to make the choice. A large-scale computer...such as the iPSC...offers the faster-than-real-time performance needed. A concurrent system offers a method to capture the knowledge used to execute a realistic simulation.

This type of large-scale simulation tool is not restricted to battle management, but could be applied in many areas such as complex factory assembly line planning, vehicular route planning, and biological life systems simulation.

IMPLEMENTATION USING CONCURRENT GESBT

Concurrent GESBT is a tool for building multiple, cooperating expert systems[1]. It allows expert systems to create other expert systems and move and copy objects among expert systems. The programmer determines which processor nodes of a concurrent computer receive the expert system.

The battle management simulation was mounted on the iPSC concurrent computer by building a single knowledge base for E2C defense strategy and then allocating six of these knowledge bases. Each of the knowledge bases was placed on a single node of the iPSC. This one-to-one mapping of knowledge bases to processor nodes was not required. Future versions would probably use only three processor for six knowledge bases. The mapping, under programmer control, is dependent on the structure of the problem. For the current community, each sector conducts defense strategies relatively independently. Therefore, each sector knowledge base is dedicated to a complete processor.

There are two other knowledge bases serving each E2C commander knowledge base. Both reside with the knowledge base they serve, on the same iPSC node. One provides expert knowledge about how to move objects in the sector, the other manages the sector-specific resources (planes and missiles).

Another knowledge base was created to track community-wide resources. This knowledge base holds the strategy of the central command, the aircraft carrier. Sector knowledge bases communicate with this resource knowledge base to report and ascertain the status of fighters. This knowledge base is allocated its own iPSC node. An eight iPSC node serves as the initializer for the entire simulation. After starting the other seven nodes, it waits for completion of the simulation.

For this implementation, the entire community of knowledge bases representing the aircraft carrier battle group was mounted on eight iPSC processor nodes. The community uses only a portion of the iPSC computational resources, because the intent is to install many of these communities at the same time on the iPSC in the near future.

---

1. See SAIC publication Concurrent GESBT, August, 1986.

Application Brief

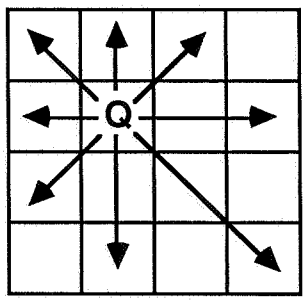
Intel Scientific Computers

Solving the N Queens Problem with the MACE System on the Intel iPSC™

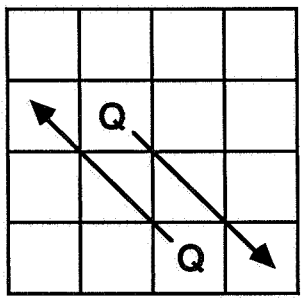
David Billstrom & Joe Brandenburg, Intel Scientific Computers
Les Gasser, University of Southern California

PROBLEM

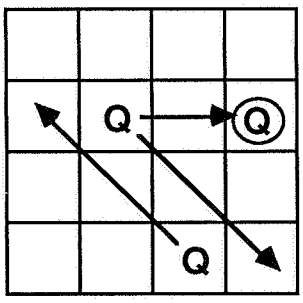
The N Queens problem is a classic computer science problem with an exponentially increasing number of solutions[1]. Simply, given a chess board of N dimensions, where should N queen chess pieces be placed? The constraint is that each queen must be immune from the attack of every other queen. Because queen pieces in chess may move in straight lines, for any distance, along X, Y, and diagonal axes, there are few eligible placements.



Queen Movements



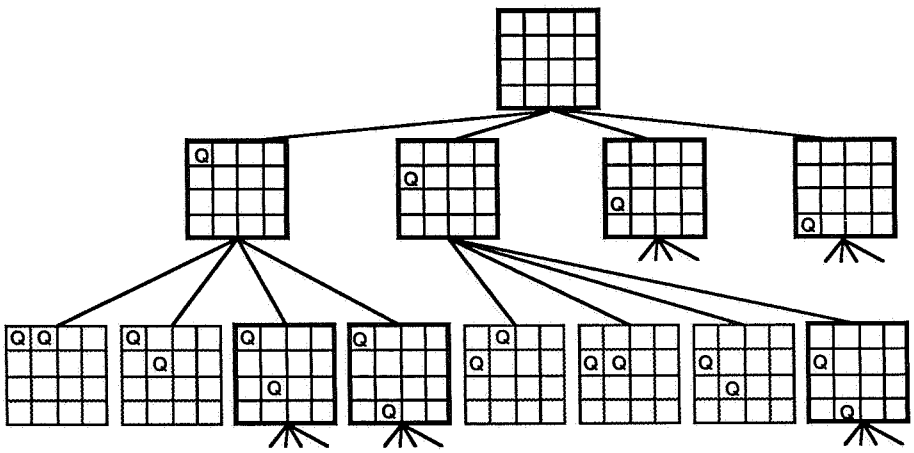
Two Legal Placements



An Illegal Placement

SOLUTION

There are several approaches and algorithms to solve the N Queens problem. The one chosen here exhaustively searches all possible chess piece placements on a given board by balancing the work among the iPSC™ nodes. The iPSC is a large-scale concurrent computer. The Contract Net protocol[2] was used to balance the network. The MACE (Multi-Agent Computing Environment) system[3] was used to implement contract net. This approach was chosen to demonstrate the ease of use of the MACE system. The illustration below shows a portion of the tree representation of the problem.



iPSC is a trademark of Intel Corporation
Explorer is a trademark of Texas Instruments, Inc.

CONTRACT NET

Contract Net is a programming model that may be applied to a network of processor nodes. The model facilitates task-sharing in cooperation to solve problems. All processor nodes participate mutually in the sharing of the task, which allows a loosely-coupled collection of nodes. The advantages of a loosely-coupled system are extendibility, flexibility, and reliability.

One processor is designated the monitor; the others are workers. Following the Contract Net protocol, the monitor attempts to form a contract with a processor node for some amount of work. The normal method of negotiating a contract is for the monitor to advertise the work to be done to all nodes. Each of the nodes evaluates the work with respect to its own specialized hardware and/or software resources. If the node has sufficient interest in the work and capability to do the work, it submits a bid. The monitor may receive a number of bids for the advertised work and chooses one or more of those bids to award the contract.

USING THE MACE SYSTEM TO IMPLEMENT CONTRACT NET

The MACE (Multi-Agent Computing Environment)[4] system enables the implementation of the monitor and the workers as *agents*. Agents can be configured to perform simple tasks, such as solving this puzzle, or more complex tasks such as providing a knowledge base. The programmer creates the agent and MACE chooses a processor for the agent to reside upon, thus freeing the programmer from the chore of balancing agents across processors.

The monitor and workers of Contract Net are implemented as MACE agents. They exchange messages about the announcements, bids, and contract awards via MACE communication services.

A monitor, implemented as a single MACE agent, starts with an empty board. The monitor creates four more boards and places a queen in a new row on each board. The next step is to extend each board by attempting to add a queen in the next column. The monitor advertises the work of extending each of these initial boards to the available workers, each implemented as a MACE agent. The bidding and awarding of contracts follows the normal Contract Net protocol. However, because each worker tries to extend its board recursively, each worker also acts as a monitor, attempting to subcontract out part of the work.

RESULTS

A simple comparison to a sequential version of the problem is not possible. Therefore, precise statistics are not available. The N Queens problem was previously implemented using Contract Net with MACE on a Texas Instruments Explorer™. The iPSC version runs about seven times faster than the TI Explorer version. This is a preliminary statistic and is not an accurate measurement. It does imply that distributed problem solving of the Contract Net variety has the potential for significant speedup by the use of the iPSC system and MACE.

---

1. Wirth, Nikalaus. *Algorithms + Data Structures = Programs,* Prentice-Hall, 1976.
2. Smith, Reid G., *The Contract Net Protocol: High-level Communication and Control in a Distributed Problem Solver.* IEEE Transactions on Computers C-29(12), December, 1980
3. Gasser, L. and Tenorio, M. *Rule-Agents: A Distributed, Object-Oriented Approach to Production Systems Using MACE.* Working paper, USC-DPS Group, Computer Science Department, University of Southern California, March, 1986.
4. Gasser, L, C. Braganza, N. Herman, L. Liu. *MACE Reference Manual.* USC-DPS Group, University of Southern California, July, 1986.

Application Brief

Intel Scientific Computers

Crystalline Operating System (CrOS III)

Intel Scientific Computers

INTRODUCTION

The Crystalline Operating System (CrOS III) offers very fast, nearest neighbor communications. This system is now available for the Intel iPSC and SugarCube. Versions of CrOS have been in use at Caltech for five years, and have demonstrated the benefits of concurrent computing in a wide variety of scientific applications. The primary advantages of CrOS include:

- **speed** — CrOS will run faster on many applications because of its simplicity.
- **portability** — CrOS is currently running on the Caltech Mark series of hypercubes as well as the commercial systems from Intel Corporation and NCUBE.
- **programming simplicity** — The synchronous communications model of CrOS ensures that bugs are reproducible and, thus, easy to locate.
- **applications** — A large library of applications has been implemented on CrOS. Most of these applications, covering the full range of science and mathematics, are available in source form to users of CrOS.

SOLVING PROBLEMS ON CONCURRENT PROCESSORS

Solving Problems on Concurrent Processors, by Geoffrey Fox and his associates in the Caltech Concurrent Computation Program is a comprehensive text on the design and use of the hypercube. It is based on the CrOS III operating system. The text includes a description of the hypercube architecture, the design goals and strategy of CrOS, performance considerations for hypercube applications, and numerous complete application examples drawn from many fields of science. This text will be available from Prentice-Hall in the Fall of 1987.

Examples in the text include: One-dimensional wave equation, solution of Laplace's equation by finite differences, using the finite element method to solve elliptic problems in two dimensions, conjugate gradient method, matrix algorithms, Fast Fourier Transform, Monte Carlo methods, generation of random numbers, simulated annealing as an optimization technique, Monte Carlo approach to Lattice Gauge Theory, particle dynamics, and sorting algorithms. The source code for these examples is available with CrOS for use in teaching and experimentation with concurrent programming techniques.

In addition, CrOS III is fully documented including a Primer, User's Manual, and numerous technical reports published by the Concurrent Computation Project at Caltech.

CrOS

The crystalline operating system, CrOS III, was designed at Caltech to run on concurrent systems. It is based on synchronous, nearest-neighbor communications.

The user interface to CrOS III is a library of routines for programs written in C or in FORTRAN. These routines provide low-level communications facilities such as *cread* and *cwrite*, higher-level communications to allow the user to shift, broadcast, and combine data from different nodes, communication between the control processor and the nodes, and various utilities.

CUBIX

CUBIX is the I/O system which provides every node with full UNIX™ I/O capabilities. It also has some new constructs which support parallel I/O. A C based version of CUBIX has been in use for over 18 months at Caltech. A FORTRAN based version has been developed and is entering use at Caltech. The advantages of using CUBIX with CrOS III include:

- **Sequential compatibility** — Programs written under CUBIX are very similar to sequential programs. In fact, if the program can be run on a 1-node hypercube, then it can also run without change on a sequential UNIX system.
- **No host** — CUBIX allows the user to program the hypercube without programming the host. The host computer runs a standard program which serves I/O requests sent from the cube.
- **Compatible with CrOS or NX (Intel's Operating System for the iPSC™)** — CUBIX can work with either CrOS or with NX. The CUBIX system calls are identical in both cases.

PLOTIX

PLOTIX is a parallel vector graphics system which runs on the hypercube. It provides a considerable range of utilities. The advantages of PLOTIX are:

- **parallel system** — it is the first parallel graphic system. Plotting capabilities are distributed within the hypercube.
- **inexpensive graphics workstations** — PLOTIX supports Tektronix 4105 and 4010 terminals and HP 7470 plotters as well as other terminals and plotters which emulate these devices.
- **device independence** — Internally, PLOTIX uses a device-independent representation for data. Additional graphics workstations can be supported easily.

CrOS III, CUBIX, and PLOTIX are available for the Intel standard and extended memory iPSC and SugarCube systems. For further information contact Intel Scientific Computers.

TechnicalNote
Intel Scientific Computers

Vector Concurrent Computing

Tony Anderson, Mike Ess
Intel Scientific Computers

SINGLE NODE VECTOR PERFORMANCE

The iPSC-VX™ (Vector Extension) is a vector concurrent computing system, an enhanced member of the iPSC family of "personal supercomputers." Using low-cost VLSI to harness the complementary technologies of concurrent computing and vector processing, the iPSC-VX sets a new standard in price-performance. The vector processor on each node increases vector performance by 100x and scalar performance by up to 10x.

VAST-2™, the FORTRAN precompiler for the iPSC-VX, automatically generates optimized code for the vector processor from standard FORTRAN 77 source programs. It recognizes FORTRAN DO loops that can be converted to vector operations such as the dot product, and builds execution modules with both vector and scalar operations that run entirely on the vector processor.

The first table below shows the performance of a vector processor for two standard vector operations (DDOT and DAXPY) compared with the standard iPSC node (Intel 286/287 processor). The table shows that the speed of the standard node is nearly independent of the length of the vector (i.e., it is not a vector engine). The speed of the vector processor improves as the length of the vector increases and the setup cost of the operation is divided among more numeric operations.

Single Vector Processor
Speed (in Mflops) of Some Simple Loops on the iPSC-VX

Length	STANDARD NODE (286/287)		VECTOR NODE [1]	
	DDOT	DAXPY	DDOT	DAXPY
8	.0203	.0216	.1778	.1391
64	.0216	.0229	.9846	.8258
128	.0217	.0230	1.5059	1.2190
256	.0218	.0231	2.0480	1.6254
512	.0219	.0232	2.4675	1.9883
1024	.0219	.0232	2.7676	2.2140
2048	.0219	.0232	2.9362	2.3540
4048	.0219	.0232	3.0341	2.4309

The second table (see below) shows the performance of a single vector processor executing the concurrent versions of DGEFA (factor) and DGESL (solve) of LINPACK for various double-precision (64-bit) matrix sizes. The largest number that can be supported as a single vector value is 300.

Speed (in Mflops) for Factoring/Solving on the iPSC-VX

Length	286/287		VECTOR NODE [1]	
	FACTOR	SOLVE	FACTOR	SOLVE
10	.011	.008	.067	.040
50	.020	.018	.641	.200
100	.021	.020	1.307	.367
200	.022	.021	1.998	.762
300	.023	.022	2.333	1.059

VECTOR NODE PERFORMANCE CONSIDERATIONS

At the FORTRAN level, the performance considerations for an iPSC vector processor are similar to those for larger vector machines; i.e., are property of the code that can be vectorized and the length of the vectors. The vector processor can be viewed at two levels. At the microarchitecture level, the vector processor is a highly efficient microprogrammed machine with short pipelines and random memory access. Thus, for specific functions that have characteristically short vectors, it is possible to achieve high performance with special routines for the vector processor.

CONCURRENT PERFORMANCE CONSIDERATIONS

The goal of concurrent processing is to achieve near-linear speedup in which p processors complete a job in 1/p of the time for a single processor. For a specific application, approaching linear speedup depends on several performance considerations:

- Parallel/sequential - Most of the computation can be done in parallel
- Load balancing - Computation and data can be divided evenly among the processors
- Communication/calculation - High degree of computation to communication
- Efficient parallel algorithms - Algorithm for concurrent computation is as efficient as the one for a sequential system

These considerations are discussed in the applications briefs contained in this package.

For more information:

Detailed technical information on the performance considerations for vector concurrent computing and additional benchmark results are available in a technical report from Intel Scientific Computers. Request: *Performance Considerations for Vector Concurrent Computing*.

---

1 The VAST-2 compiler for the iPSC-VX is currently under development by Pacific Sierra Research. The times for VAST-2 were obtained by hand compiling code for the vector processor based on the planned code generation strategy. The resulting code was executed on an iPSC-VX system to obtain these times.

TechnicalNote
Intel Scientific Computers

A Closer Look At Amdahl's Law

Cleve Moler
Intel Scientific Computers

AMDAHL'S LAW

Amdahl's Law, which dates from 1967 [1], says that a parallel computation cannot run any faster than its inherently sequential portion. If 5% of an algorithm cannot be parallelized, then the parallel computation cannot run more than 20 times faster than the sequential computation, no matter how many processors are used. When expressed in such terms, the law is clearly valid.

However, Amdahl's Law only talks about a single problem. A closer look at Amdahl's Law involves consideration of a sequence of problems of increasing size. We argue that the fraction of time spent in the sequential part of an algorithm may actually decrease as the problem size increases, and hence that parallel computation can become increasingly efficient.

To make these notions a little more precise, let:

$$T_p = \textit{time to solve a problem on p processors}$$

and let the speedup be

$$S_p = \frac{T_1}{T_p}$$

The goal of parallel computation is to approach linear speedup, $S_p \approx p$. Let α denote the Amdahl fraction, which is the fraction of the algorithm which is "not parallelizable." Then simple algebra produces:

$$T_p = \frac{1-\alpha}{p} T_1 + \alpha T_1$$

and

$$S_p = \frac{p}{1+(p-1)\alpha} \leq \frac{1}{\alpha} \quad \textit{for all } p$$

This is Amdahl's Law.

A NEW DEFINITION

Now consider a sequence of problems of increasing size, n. The parameter n might represent the length of a text to be analyzed, or the number of atoms in a chemical compound, or the number of cells in a finite element mesh. It should not be confused with the number of processors, p. We claim that for many algorithms, the Amdahl fraction depends upon the problem size and so we make the following definition:

An *effective* parallel algorithm has

$$\alpha(n) \rightarrow 0 \ as \ n \rightarrow \infty$$

This does not mean that the absolute time spent in the non-parallel part of the computation goes to zero, but only that it becomes negligible when compared to the overall time. Now

$$S_p = \frac{p}{1 + (p-1)\,\alpha(n)}$$

Hence, for effective algorithms, if we fix p, and let $n \to \infty$

$$S_p \to p$$

This says that for a fixed machine, linear speedup can be approached by using appropriate algorithms on larger and larger problems. Speedup is limited more by memory size than by the number of processors.

Do effective algorithms exist? In our experience with distributed memory, message passing, multiprocessors, and medium to coarse grained parallelism, we have found that most algorithms for engineering and scientific problems are effective by our definition. We offer one simple example to illustrate these ideas. Let A be a large matrix, distributed by rows across several processors, and x be a vector, stored in one processor. We wish to compute the matrix-vector product, $y = Ax$.

The following algorithm accomplishes the task.

Send x to all other processors.
All processors simultaneously:
    for i in my rows
        $y_i = A_i \cdot x$
Accumulate y in destination processor.

To analyze the computation time and the resulting speedup, let n be the order of matrix and let p be the number of processors. Furthermore, let m be the maximum number of rows in any processor. Because we are not assuming that n is exactly divisible by p, we have

$$m = \lceil n/p \rceil \leq n/p + 1$$

We must include in the Amdahl fraction all of the parallel overhead associated with distributing the data x and accumulating the result y, as well as the load imbalance which results from possibly unequal number of rows on each processor.

The amount of work is easily estimated. If we assume that the processor interconnect topology contains a hypercube, then the complexity of the communication tasks grows like log p. Other topologies would have a different dependence upon p, but would lead to the same general conclusions.

Step	Work
Send x	$n \log p$
m dot products	$m\ n$
Accumulate y	$n \log p$
Total	$O(m\ n) + O(n \log p) = O(\frac{n^2}{p}) + O(n) + O(n \log p)$

The first term in the totals, which dominates for large n, describes the parallel portion of the computation. The second term accounts for the load imbalance, and the third term describes the communication costs. It is not quite accurate to say these latter terms correspond to the "sequential" portion of the algorithm, but they do represent any losses of efficiency in the parallel computation. It is not hard to see that

$$\alpha(n) = O(\frac{1 + \log p}{n}) \to 0 \text{ as } n \to \infty$$

Hence, the distributed matrix-vector multiplication algorithm is effective by our definition. As the problem size increases, the parallel overhead becomes negligible in comparison to the total time.

---

[1] G.M. Amdahl, "*Validity of the single processor approach to achieving large scale computing capabilities*" Proc. AFIPS Comput. Conf., vol.30, 1967.

Technical Note

Intel Scientific Computers

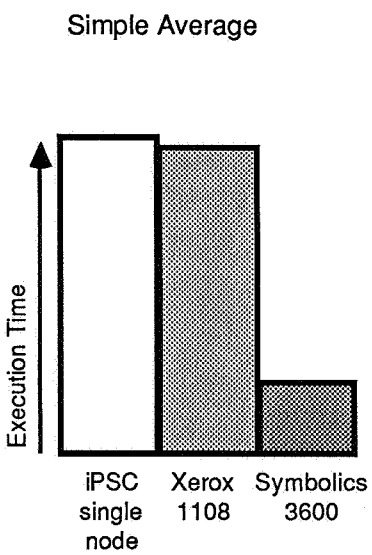
Performance of CCLISP™ on the Intel iPSC™

Intel Scientific Computers

The Gabriel Benchmarks[1] are the AI industry standard for evaluating the performance of LISP systems. Although the collection of programs do not represent a precise balance of all AI problems, selected benchmarks can be used as predictors of LISP system performance on larger, complex problems.

SINGLE-NODE PERFORMANCE

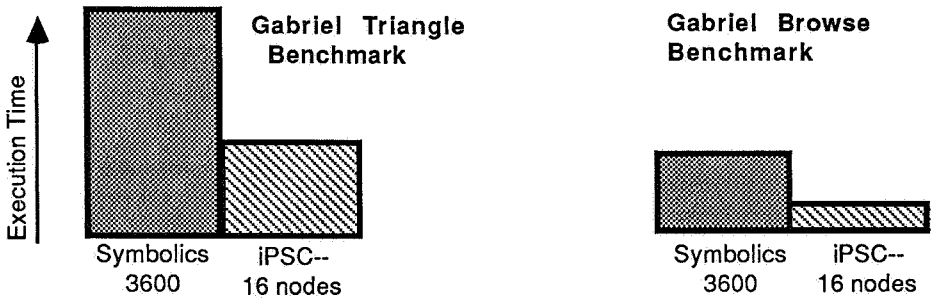
Performance of Concurrent Common LISP4 on a single iPSC™ node is roughly equivalent to a low-end LISP workstation, such as the Xerox 1108 Dandelion™. This performance is shown in the chart below:



Gabriel Benchmarks	CCLISP on a single iPSC node (compiled)	Xerox 1108 Dandelion[2]	Symbolics[3] 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 interative	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 hit	49.25	48.00	8.62
Average	**135.61**	**104.57**	**18.34**

CONCURRENT SYSTEM PERFORMANCE

The important performance criteria for evaluating concurrent computers is *speedup*. **Significant speedup can be gained from concurrent execution of certain algorithms.** Two typical simple symbolic programs yielded the following *speedup* from concurrent execution:



	CCLISP on a single iPSC node	CCLISP on a 16-node iPSC	Efficiency (use of available cpu)	Symbolics 3600
Browse	186 seconds	9.3 seconds	125%	30.8 seconds
Triangle	990 seconds	68 seconds	91%	152 seconds

---

1. R. Gabriel, *Performance and Evaluation of LISP systems.*
2. Refer to ISC Applications Brief entitled, *"Triangle, A Concurrent Version of the Gabriel Benchmark."*
3. Refer to *String Search, A concurrent String Search Example*, AI Note 104, Intel Scientific Computers.
4. Concurrent Common LISP (CCLISP) is available for the iPSC from Gold Hill Computers, Inc.
5. The results for the Xerox 1108 Dandelion were obtained from R. Gabriel, *Performance and Evaluation of LISP Systems*, 1985.
6. The results for the Symbolics 3600 were obtained from R. Gabriel, *Performance and Evaluation of LISP Systems*, 1985.

Technical Note
Intel Scientific Computers

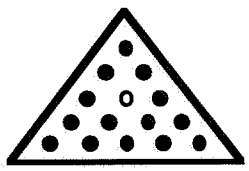
Triangle, A Concurrent Version of the Gabriel Benchmark

Intel Scientific Computers

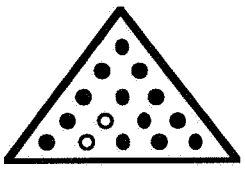
THE TRIANGLE BENCHMARK

The Gabriel Triangle[1] Benchmark predicts LISP performance on a class of AI tree search problems. The sequential algorithm has been implemented in Concurrent Common LISP (CCLISP™) for the iPSC™, a distributed memory, message-passing computer.

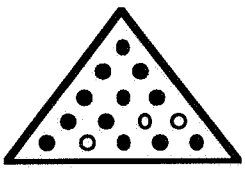
The benchmark finds all solutions to the "triangle game." This game consists of a triangular board with 15 holes. A peg is placed in every hole except the middle. The player makes a move by jumping over a peg into a vacant hole and removing the jumped peg, as in checkers. The object of the game is to remove all of the pegs but one. There are many possible sequences of moves, but only 1550 sequences result in a single remaining peg. The Gabriel version of the algorithm finds 775 solutions. The other 775 solutions are symmetrically identical (only one of the two initial moves is taken by the benchmark algorithm).



Initial Board Layout



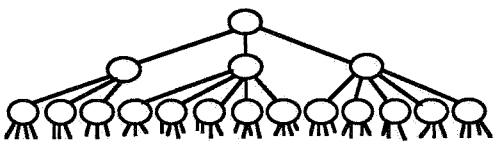
... after one move



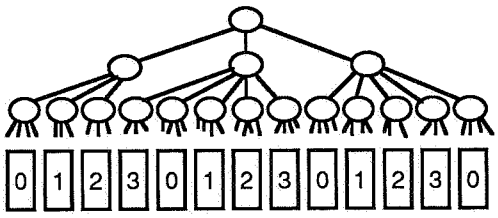
... after two moves

STATIC MAPPING OF THE SEARCH TREE

The general problem is represented as a tree of possible moves with each node of the tree representing a decision about the next possible move. The search tree is easily mapped onto a concurrent computer, such as the iPSC, by assigning subgraphs of the tree to each iPSC node. In the representation below, 13 subgraphs of the triangle search tree are selected by each of the four nodes of a dimension 2 hypercube. The work is (relatively) equally distributed. Concurrent execution (16 processors) is 10 times faster than execution on a single processor.



The search tree

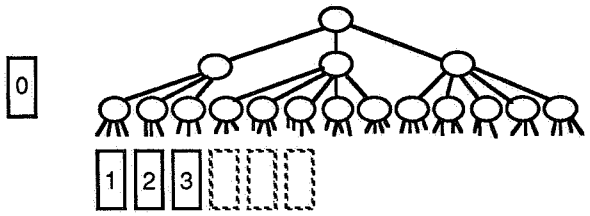


... mapped on to iPSC nodes

iPSC is a trademark of Intel Corporation
CCLISP is a trademark of Gold Hill Computers, Inc.

A SECOND TECHNIQUE: DYNAMIC LOAD BALANCING

A second method was developed to use dynamic load balancing, based upon the Problem Heap technique[2]. This method uses a single node as a manger. The manager assigns each of the other nodes subgraphs from the search tree. For 16 processors, **the resulting speedup is 14.5 times faster than a single processor.**



RESULTS

Gabriel benchmarks are measured for speed of execution. A concurrent implementation of one of the benchmarks will demonstrate *speedup* gained from concurrent execution. This is the first empirical data for concurrency speedup of a tree search in Common LISP on a commercial concurrent computer.

	CCLISP on a single iPSC node	CCLISP on a 16 node iPSC	Efficiency (use of available cpu)	CCLISP on a 32 node iPSC	Symbolics 3600
Static Mapping	990 seconds	100 seconds	62.5%	49.5 seconds	152 seconds
Problem Heap	990 seconds	68 seconds	91%		152 seconds

CONVERTING THE SEQUENTIAL BENCHMARK FOR CONCURRENT EXECUTION

The Triangle Gabriel Benchmark was implemented on the iPSC, a distributed memory, message-passing concurrent computer, by simple modifications to the Gabriel algorithm, and by the use of Concurrent Common LISP (CCLISP), a Common LISP implementation supporting message passing.

The static mapping version changed three portions of the original sequential algorithm. First, an initial "start" message is propagated to each processing node of the iPSC hypercube. This is accomplished via a minimal spanning tree broadcast[4]. Second, the program on each processing node expands the search tree four levels deep. At this level, about 120 *leaves* (or subgraphs) exist for the tree. The program, which is identical on each node, determines the identity of itself (its node number) and chooses an equal, or near equal, share of the leaves. Then each of the subgraphs from these leaves is solved using the traditional algorithm. The third change to the algorithm directs each processing node to communicate the results of its computation back to the C program running on the cube manager. This is also implemented with message passing.

For the Problem Heap version, one node is identified as the manager. It alone solves the tree to the fourth level of 120 leaves. It then distributes each of the 120 subgraphs to the remaining 15 worker nodes, assigning a subgraph to each worker node as the worker node becomes available. Each node solves the subgraph ;using the traditional sequential algorithm and then reports the results back to the manager. It then requests another subgraph. The manager continues to distribute subgraphs and receive results until results for all subgraphs have been received from worker nodes. It then communicates the results to the C program running on the cube manager.

MESSAGE PASSING WITH OTHER LANGUAGES

The version of triangle used for demonstration purposes also incorporates a color graphics display. It follows the same algorithm described above but illustrates an additional facility, C and CCLISP processes communicating with messages. Solutions computed by each processing node (in Common LISP) are communicated to the iPSC cube manager graphics program (in C) via message passing primitives.

---

1. R. Gabriel, *Performance and Evaluation of LISP systems.*
2. Moller-Nielsen, P. and J. Straunstrup. *Problem Heap: A paradigm for Multiprocessor Algorithms.*
3. Concurrent Common LISP (CCLISP) is available for the iPSC-MX from Gold Hill Computers, Inc.
4. Brandenburg, J. and Scott, D. *Embeddings of Communication Trees and Grids into Hypercubes.*

iPSC™ Concurrent Debugger

The iPSC™ Concurrent Debugger is a source level debugger designed to enhance the productivity of iPSC application developers. It provides all of the symbolic debugging capabilities found in sophisticated sequential debuggers. In addition, it has features tailored to the unique requirements of debugging large concurrent iPSC applications.

CONCURRENT DEBUGGING CHALLENGES

iPSC applications are composed of multiple processes executing concurrently on up to 128 nodes and exchanging messages with each other. Because each of the processes is an instance of a sequential program with extensions to send and receive messages, the task of debugging concurrent applications presents all of the challenges of debugging traditional sequential applications. However, it also has the added challenge of tracking the simultaneous...but independent... execution of many different processes on different nodes and monitoring the messages exchanged between processes.

The magnitude of this challenge increases with the complexity of the application. Typical applications are composed of 32 to 128 processes executing concurrently and exchanging hundreds, or even thousands, of messages. Even homogenous applications with regular message passing patterns can be very hard to debug with inadequate tools.

CONCURRENT DEBUGGER CAPABILITIES

Given the magnitude and complexity of concurrent applications, it is essential that the user: ***focus*** his attention on a given part of the application at a time, be able to quickly and easily ***change*** the focus of his attention to other parts of the application, and...in general...be able to ***zoom*** in and out of any given part. The Concurrent Debugger provides these capabilities by having a "context" associated with each debug command. This context defines the set of processes that will be affected by the debug command.

In the iPSC, each process is uniquely identified by the node id (nid) it is executing on and its assigned process id (pid). Thus, processes are specified by providing their (nid:pid) combination. A context is then defined by providing a list of one or more (nid:pid) pairs to the **context** command. For example:

- to focus on node 127 — type: context (127:1)
- to change the focus to node 0 — type: context (0:1)
- to zoom out and focus on all nodes, process 1 — type: context (nodes:1)

Another critical ability required to debug large concurrent applications is the ability to quickly and easily find any *"lost messages."* These are messages that are sent from one process to another, but, due to various programmer errors, are not received at their destination. Lost messages typically halt the progress of the computation and are very hard to find without special support from the debugging tool. The Concurrent Debugger allows the user to inspect the system message buffers where all messages arriving at a destination node are stored until they are received by the proper destination process. Any lost messages would be stored here and can be detected by using the **msgq** (for message queue) command . For example

- to check for all messages queued up on node 10 — type: msgq (10:all)
- to check for all messages queued up on all nodes — type: msgq (all:all)

The **msgq** command not only lets the user find lost messages, but also trace their cause. It does this by providing information on each message's originating process, intended destination, type, and length.

A complementary Concurrent Debugger command is **recvq**. This command lists requests to receive messages that have not yet been satisfied with the arrival of the corresponding message. For example:

- To find out which processes on node 127 have not yet received messages they expected, type: recvq(127:all).
- The response specifies which processes on node 127 are expecting messages and the message type specified in each receive request.

Together, the **msgq** and **recvq** commands allow the user to quickly and easily detect most, if not all, of the message related bugs in iPSC applications.

The Concurrent Debugger also has capabilities that are valuable in detecting bugs in iPSC program logic. These bugs can be in the purely sequential logic of a single process or in the logic of communicating and synchronizing multiple concurrent and independent processes. To help the user detect the first kind of problem, the Debugger allows the user to zoom in on any node and apply the full power of a high level source code debugger to a single process. To help with the second, more difficult problem, the Concurrent Debugger takes those capabilities found useful in the first case and extends them to the parallel environment of the iPSC. Specifically, the user can:

- symbolically inspect and modify variables across processes and nodes
- set break and trace point across processes and nodes
- single step across processes and nodes

OTHER CAPABILITIES

Some of the other features of the Concurrent Debugger let the user:

- Set conditional break and trace points based on the values of user variables...such as loop counters and variables used in algebraic equations.
- Execute debug scripts that collect a set of commonly used debug commands.
- List source code directly from the debugger.
- Execute Xenix commands from within the debugger.
- Keep a log of the debug session in a file for later review.
- Obtain On-line help which describes how to use each of the debug commands.

Product Brief

Intel Scientific Computers

iPSC™ Simulator

PARALLEL COMPUTING SIMULATOR

The iPSC™ Hypercube Simulator is a program that simulates the parallel computing environment of the Intel iPSC Hypercube System. This simulator is made available to application developers, researchers, teachers, and students in order to further the understanding of parallel programming concepts and to enable rapid checkout of programs.

Hypercubes are extremely flexible supersets of parallel topologies such as loops, trees, meshes, and toroids. The hypercube offers an ideal environment for embedding parallel applications. Coding in standard FORTRAN and C languages, developers can realize parallel constructs of their programs by "test-bedding" them on the iPSC Hypercube Simulator. The application interface is consistent between the iPSC Hypercube Simulator and the iPSC System, allowing easy migration if the performance of an iPSC System is desired. This also speeds the program development cycle by providing facilities for proving out code designs as well as locating programming errors.

FEATURES

- **FORTRAN or C** — The simulator executes hypercube programs written in either the C language or Ryan McFarland FORTRAN. It simulates the actions of the iPSC node executive routines invoked by the user's programs.
- **Modification Not Required** — Programs do not have to be modified to run on the simulator (with the exception of Ryan McFarland FORTRAN in which INTEGER*2 must be set as the default).
- **Interactive Interface** — The simulator provides an interactive interface which simulates the iPSC's cube manager commands. A programmer begins simulation by issuing these commands to the simulator. In addition, cube manager commands can be placed in a script file and directed to the simulator's standard input on the command line.
- **Interrupt Handling** — The programmer can interrupt the simulated hypercube's process (or processes), check its status, and then restart the simulator from where it was interrupted...without ever exiting the simulator.
- **Messages** — Both error and trace messages aid in detecting errors in the use of host and operating system routines.

RE-HOSTING ON OTHER OPERATING SYSTEMS

Re-hosting the simulator on other multi-tasking operating systems is possible and the simulator is delivered in source code form to support such projects. A detailed engineering design specification is supplied for the systems programmer who wishes to re-host the iPSC Simulator or enhance and customize its feature set.

COMMANDS

Most of the commands at the programmer's disposal simulate the commands which control the cube and are typed in at the iPSC's cube manager console. There are, however, a few commands that are unique to the simulator.

In addition to simulating the iPSC commands, the simulator provides other useful commands which are briefly mentioned below.

Command	Abbreviation	Description
cubelog	log	manages the simulator's log file
cubeman	m	loads programs into simulated cube manager
help	h or ?	prints summary of simulator commands and their use
quit	q or exit	terminates simulation and exits the simulator
start	s	starts the simulation after cube and host processes are loaded
status	st	checks the status of an application. It has the following switches: -a = all (message, process, request, and traffic status) -m = message status -p = process status -r = request status -t = message traffic
system	!	passes commands through the simulator to the shell
trace	t	enables or disables tracing of node operating system (NX) calls

SUPPORT

The iPSC Hypercube Simulator is available in source format for XENIX™ and UNIX™ 4.2 BSD systems including the Sun workstation and DEC VAX™. A "MAKE file" is Included with the source code. This file is a specialized "program" that allows the host machine to correctly create the machine-executable version of the simulator.

A concise manual, approximately 20 pages, presents an overview of simulator operation, program preparation, details of iPSC Simulator commands, and a listing of iPSC node operating System commands. A 150--page manual provides a description of the iPSC System, concurrent programming concepts, FORTRAN and C interface routines, a guide to developing concurrent programs, and a glossary of terms.

Product Brief

Intel Scientific Computers

VAST-2™ Fortran Vectorizer for the IPSC-VX™

VAST-2™ serves as an optimizing FORTRAN compiler for the numeric accelerators on each node of the iPSC-VX™ system. As the user interface, it accepts FORTRAN 77 and 8x vector syntax and builds optimized code for execution on each node of the iPSC-VX. VAST-2 also has diagnostic functions that aid the user in optimizing code for vector execution, and supports user directives for controlling the vectorization process at the FORTRAN level.

VECTORIZATION CAPABILITIES SUMMARY

VAST-2 is a sophisticated FORTRAN vector preprocessor, capable of vectorizing 18 of the 24 Livermore Loops (Mflops benchmark). The following operations are supported:

- Vectorizes *both* DO loops and IF loops.
- Vectorizes *outer* loops.
- Analyzes data dependencies to ensure safe translation of loops.
- Examines EQUIVALENCE statements to detect hidden recursion.
- Re-orders array references to avoid recursion, where possible.
- Uses program information from *outside* the loop to aid data dependency analysis (ambiguous subscript resolution).
- Minimizes data dependent sections of loops so that the maximum amount of computation is vectorized.
- Handles all forward-branching conditional operations, even deeply nested or complicated branches.
- Uses program information from *outside* the loop to determine whether final values of indices and of scalar temporaries are required.
- Re-rolls loops which have been "unrolled."
- Supports user directives and switches to control most aspects of the translation process.
- Supports vector common sub-expression elimination.
- Selects the best single dimension in a loop nest for vectorization.
- Collapses multiple dimensioned loops into one long loop.

FUNCTIONALITY

VAST-2 analyzes standard FORTRAN application code, detects vector constructs, and translates those vector operations into functions that can be executed on an iPSC-VX vector processor board. VAST-2 detects single, double, and triple operand operations as well as reduction operations that are supported in VecLib, the math library for the iPSC-VX. The vectorized FORTRAN output is translated into vector and scalar command operations that drive the vector processor. These command sequences can be combined into a single iPSC-VX execution module that executes entirely on the vector processor. In summary, VAST-2 automatically performs the following functions:

- translates vector constructs into iPSC-VX VecLib operations to optimize performance of node vector processor boards
- translates scalar instructions into iPSC-VX scalar operations, taking advantage of the short pipelines and direct memory access to provide balanced vector-scalar performance
- builds iPSC-VX execution modules that run as complete subroutines in a vector processor, minimizing system overheads and maximizing performance

DIAGNOSTICS AND DIRECTIVES

VAST-2 supports user diagnostics and directives that inform the user regarding loops vectorized, loops not vectorized, and reasons for rejection. The user can respond by changing the FORTRAN program, and by inserting user directives that provide information to VAST-2 about the overall program and data structure that could not be discerned from examination of an isolated program segment. Diagnostic output reports include: data dependency conflicts, syntax errors, translation problems, and possible opportunities for further optimization.

User directive commands allow the user to perform such functions as: optimizing the vectorization of an application by defining the treatment of subroutine CALLS within loops, specifying data dependency between variables, and selecting loops for vectorization in a nest of loops.

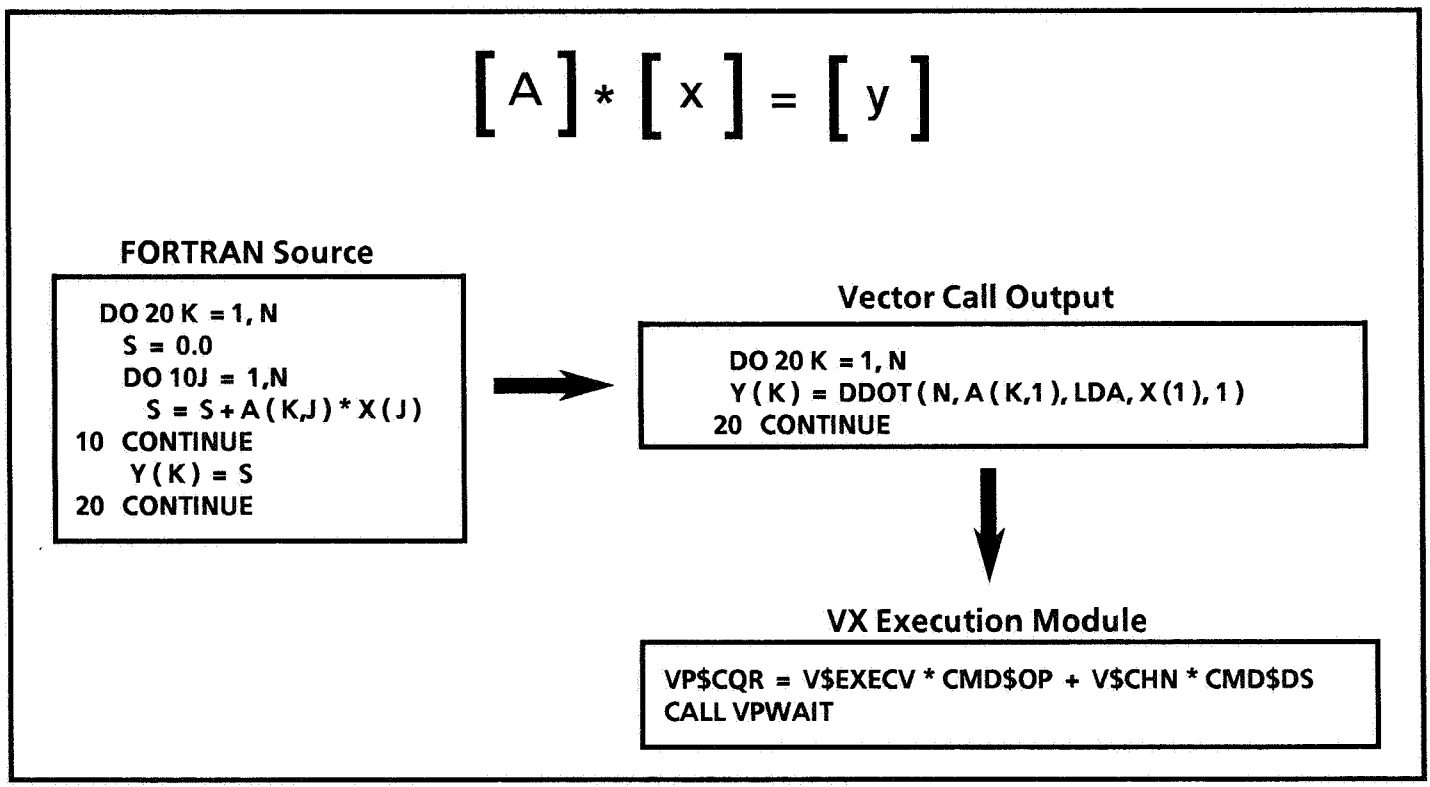
VecLib FUNCTIONS SUPPORTED BY VAST-2

VecLib functions supported by VAST-2 include:

- single variable operators such as scatter/gather and transcendental functions
- double variable operations such as vector add, multiply, divide, Ax + Y, etc.
- triple variable operations such as vector vector add, vector multiply, etc.
- reduction operations such as vector sum and dot product

VecLib includes additional functions (such as FFT, plane rotation, tridiagonal solve operators, etc.) that can be used to enhance the performance of applications through more effective use of the micro-architecture of the vector processor. Using VX tools, the programmer can also develop proprietary microcode routines for specialized applications.

VAST-2 Example - Matrix Vector Multiply



Product Brief

Intel Scientific Computers

iPSC-VX™ Vector Library (VecLib)

VecLib is the math library for the iPSC-VX™ vector processor. It contains a variety of functions and operations which have been organized to conform with the calling and naming conventions of the Basic Linear Algebra Subprograms (BLAS). Each of the routines in the library represent microcoded functions that reside in, and are executed by, the vector processor. In this sense, VecLib forms the access interface for the user to the vector processor. This access takes two forms: direct access using subroutine calls, and indirect access through the VAST-2™ FORTRAN vectorizer. The vectorized output of VAST-2 contains commands to execute VecLib routines on the vector processor.

VecLib is also supported in versions that can be used on standard iPSC nodes as well as a transportable FORTRAN version for use with the iPSC simulator. These versions of VecLib are intended for developing application programs and debugging programs ultimately targeted for the iPSC-VX.

The table shows a subset of the routines that make up VecLib. Performance numbers shown represent the peak performance or the performance which would be achieved if vector lengths were infinite ($R\infty$). Performance is shown for 32- and 64-bit operations with data residing both in the 1 Mbyte dynamic data memory and in the 16 KByte static (fast) memory.

Command	Description	32-Bit		64-Bit	
		Static	Dynamic	Static	Dynamic
xASUM	Sum absolute values	10.00 §	5.00 §	10.00	4.00
xAXPY	Constant times vector plus vector	6.67 §	3.33 §	6.67	2.67
xCLIP	Clip to interval	3.33	2.00	3.33	1.67
xCOPY	Copy vector				
xDOT	Dot product of two vectors	10.00 §	5.00 §	6.67	3.33
xFILL	Fill vector				
xGATHR	Vector gather				
xICLIP	Invert clip	3.33	2.00	3.33	1.67
xLBIDI	Solve a lower bidiagonal linear system				
xNEG	Change sign	5.00 §	2.50 §	5.00	2.00
xNRM2	Euclidean vector norm				
xRAMP	Ramp function	10.00	5.00	10.00	4.00
xROT	Apply a plane rotation				
xROTG	Construct Givens plane rotation				
xSCAL	Constant times a vector	5.00	2.50	3.33	1.33
xSCATR	Vector scatter				

x = S for single precision D for double precision

§ Denotes routines which have 2x performance for consecutive data organization (32-bit only)

Command	Description	32-Bit		64-Bit	
		Static	Dynamic	Static	Dynamic
xSUM	Vector sum	10.00	5.00	10.00	4.00
xSWAP	Swap vectors				
xTRIDI	Solve a positive definite tridiagonal linear sys.				
xUBIDI	Solve an upper bidiagnoal linear system				
xVABS	Element-wise absolute value	5.00	2.50	5.00	2.00
xVADD	Vector addition	3.33	1.67	3.33	1.33
xVAMAX	Vector element-wise maximum absolute value	3.33	1.67	3.33	1.33
xVAMIN	Vector element-wise minimum absolute value	3.33	1.67	3.33	1.33
xVATAN	Element-wise inverse tangent				
xVATN2	Element-wise inverse tangent, two operands				
xVCOS	Element-wise cosine	0.65	0.57	0.30	0.28
xVDIV	Vector division, element-by-element	0.67	0.56	0.28	0.25
xVEXP	Element-wise exponential	0.37	0.34	0.19	0.18
xVFIX	Truncate elements to integer values	5.00	2.50	5.00	2.00
xVFLOA	Convert integer to floating-point	5.00	2.50	5.00	2.00
xVLG10	Element-wise base 10 logarithm				
xVLOG	Element-wise natural logarithm	0.36	0.34	0.13	0.12
xVMAX	Vector element-wise maximum	3.33	1.67	3.33	1.33
xVMIN	Vector element-wise minimum	3.33	1.67	3.33	1.33
xVMUL	Vector multiplication, element-by-element	3.33	1.67	3.33	1.33
xVNEG	Negate vector				
xVPOLY	Vector polynomial evaluation				
xVPOW	Element-wise power function				
xVRECP	Vector reciprocal				
xVSIN	Element-wise sine	0.65	0.57	0.30	0.28
xVSQRT	Element-wise square root	0.43	0.40	0.26	0.24
xVSUB	Vector subtraction	3.33	1.67	3.33	1.33
xVVMVT	Vector minus vector quantity times vector				
xVVTVP	Vector minus vector quantity times vector				
xVVVTM	Vector minus quantity of vector times vector				
IxAMAX	Index of maximum absolute value	2.86	2.22	2.86	2.00
IxAMIN	Index of minimum absolute value	2.86	2.22	2.86	2.00
IxMAX	Index of maximum value	2.86	2.22	2.86	2.00
IxMIN	Index of maximum value	2.86	2.22	2.86	2.00
VDBLE	Convert single to double precision	5.00	2.50	5.00	2.00
VSNGL	Convert double to single precision	5.00	2.50	5.00	2.00
CFFT	Fast Fourier Transform		10.70*		
CIFFT	Inverse fast Fourier Transform		10.70*		

* 1024-point complex FFT

NEKTON™
A Computational Fluid Dynamics and Heat Transfer Package for the Intel iPSC™-VX

INTRODUCTION

NEKTON™ is a state-of-the-art fluid dynamics and heat transfer numerical simulation package developed, marketed, and supported by Nektonics, Inc. Working in cooperation with Intel Scientific Computers, Nektonics is porting NEKTON to the Intel iPSC™-VX vector concurrent supercomputer. The NEKTON product includes graphics-oriented pre- and post-processing software, which is provided on popular workstation hosts, plus an iPSC-resident NEKTON simulator. In a typical work environment, several individuals employ graphics workstations for problem definition and the presentation and interpretation of results, while sharing access to the iPSC-VX as a NEKTON compute server.

DESCRIPTION

NEKTON on the iPSC-VX provides the engineer, scientist, or designer with a cost effective, easy-to-use numerical simulation tool capable of solving a wide variety of complex 2- and 3-dimensional fluid dynamics and heat transfer problems. NEKTON solves the full incompressible, unsteady Navier-Stokes and energy equations. It is particularly well suited for detailed calculations of flows in the laminar and transitional regimes. NEKTON has already been implemented on leading supercomputers such as the Cray X-MP and has been used to solve a wide variety of problems in science and engineering. Applications for NEKTON include:

Aerospace	Thermal avionics control , drag reduction and flow control, hydraulic systems and actuators, fuel delivery systems, and gas turbine cooling systems
Automotive	Radiator design; low-speed aerodynamics; duct, manifold, and lubrication flows
Biomedical	Fluidics and fluid delivery systems, prosthetic organs, and bio-exchanger design
Electronics	Thermal management of electronic chips, components, and packages
Energy	Design of heat exchangers
Environmental	Filter and membrane studies, air quality control, and flow around buildings
HVAC	Flow distributions in rooms, duct flows, and clean air room design
Materials	Crystal growth, glass making, casting, smelting, welding, and molten metal handling
Nuclear	Heat exchanger design, fuel rod bundle thermal hydraulics, plenum thermal-hydraulics, and valve analysis
Process	Membrane, filter and separator system design; analysis of drying and coating processes; flows in reactors, piping systems, and fluidic and metering devices; and polymeric flows

NEKTON ON THE iPSC-VX

NEKTON on the iPSC-VX presents the user with a fully integrated design environment. The pre-processor, PRENEK, uses the principles of modern computer-aided design to guide the user through the process of describing the geometry and physical parameters that define a problem. Menu and mouse-driven interactive mesh input with visual color feedback enable easy, time-efficient mesh generation and boundary condition specification for both 2- and 3-dimensional geometries.

Output files from the PRENEK problem definition stage are transferred via high-speed link to the iPSC-VX for numerical simulation. This tightly coupled design environment is far more convenient and cost effective than the use of a remotely connected supercomputer and provides far more power than is available from a "mini-super."

In addition to the performance gains it provides through parallel processing, NEKTON's computational speed is further enhanced through the use of innovative numerical techniques. The spectral element method employed by NEKTON is a high-order finite element method that combines the advantages of low-order finite element techniques with the computational efficiency and accuracy of spectral methods. Based on current projections, a 32-node iPSC-VX will be able to deliver computational performance equal to a Cray X-MP at one-tenth the cost.

NEKTON's post-processor, called POSTNEK, is designed to ensure that simulation results can be readily understood. POSTNEK uses menu-driven interactive graphics to provide the user with easy access to the NEKTON-generated database. A full understanding of complex 3-dimensional simulation results is made possible by the use of full color graphics, the capability to view the flow domain at arbitrary viewing angles, and the ability to examine velocity, pressure, and temperature fields on arbitrary planes in the flow.

Support for NEKTON is provided by a staff of dedicated engineering and computer science professionals to ensure that NEKTON is applied in the most effective manner possible. A detailed product summary on the features and capabilities of NEKTON is available from Nektonics, Inc.

SUMMARY OF BENEFITS

The benefits of NEKTON on the iPSC-VX include:

- Supercomputer performance
- Cost effectiveness
- Ease of use
- Accurate numerical simulation results
- A multi-user integrated work environment
- Wide applicability to fluid dynamics and heat transfer problems

PRODUCT AVAILABILITY

NEKTON will be available on the iPSC-VX in January, 1988. For further information contact:

Nektonics, Inc.
P.O. Box 22
Bedford, MA 01730
(617) 275-4011

Intel Scientific Computers
15201 Greenbrier Parkway
Beaverton, OR 97006
(503) 629-7709

Product Brief

Intel Scientific Computers

Concurrent Artificial Intelligence

Intel Scientific Computers

ARTIFICIAL INTELLIGENCE

The more demanding real-world problems being attempted by today's artificial intelligence techniques depend on greatly increased computer power. There is a growing realization among the leaders in AI development that concurrent computing is the most effective way to satisfy its ever-increasing need for computational capability. Intel is a leader in providing computer systems to support research in concurrent AI techniques. The Sugar Cube™, recently introduced by Intel Scientific Computers, provides an ideal system to support university education and research in concurrent AI.

Artificial intelligence techniques are being applied to complex real-world problems such as robot control. Researchers are developing concurrent techniques including simulating a system of independent cooperating agents, objects, or processes. Solution of symbolic problems on a concurrent computer often seems a more natural approach than enforcing the artificial sequence of computation required by a conventional computer.

AI is currently being commercialized by integrating AI techniques into general problems. This requires a mixed environment in which LISP-based systems (agents, objects, or processes) interface with systems implemented in algebraic languages such as C and FORTRAN.

SUGAR CUBE SYSTEM

The Extended Memory Sugar Cube provides four concurrent processors each with 4.5 MBytes of memory. The cube manager is available to support networking the system via TCP/IP to other academic and research computers on campus. Concurrent Common Lisp (CCLISP™) is included along with programming environments for C and FORTRAN. A hybrid SugarCube system could be configured with two CCLISP nodes and two C or FORTRAN nodes. The FORTRAN nodes could be vector nodes.

CCLISP

Concurrent Common Lisp (CCLISP) is an implementation of Common LISP designed to run on the Intel iPSC™ concurrent computer family, including the Sugar Cube. As an emerging industry standard, Common LISP provides the foundation for a broad base of AI development tools and applications. Concurrent Common LISP extends this powerful environment to allow multiple Common LISP processes to cooperate asynchronously and spawn new processes via message passing on Intel's concurrent computer architecture. An important feature of CCLISP is the ability for Common LISP processes to communicate naturally with processes implemented in C or FORTRAN.

FCP

Flat Concurrent Prolog (FCP) is an experimental concurrent programming language which has been implemented on the Intel Personal Super Computer (iPSC). FCP is a process-oriented, logic-based language. It is an implementation language for other concurrent languages such as Concurrent Prolog, Parlog, and Guarded Horn Clauses. The language was developed at the Weissman Institute in Rehovat, Israel.

FCP differs from Prolog in that there is no backtracking control strategy. Instead, FCP views each literal in a clause as a specification of a process which may execute concurrently. The organization and simplicity of FCP make it amenable to parallel execution on the hypercube and eases the task of writing parallel programs.

FCP has an efficient sequential implementation, comparable in speed to commercially available Prolog compilers. The practicality of the language has been demonstrated on a number of non-trivial programming tasks. These tasks include: a boot-strapping compiler, sections of an operating system, a simple graphics package, and a programming environment.

FCP has proved to be an effective tool for the design of parallel algorithms and systems. It has been used as the basis for an advanced course on logic languages at Stanford University.

CONCURRENT GESBT: GENERIC EXPERT SYSTEM BUILDING TOOL

GESBT (Generic Expert System Building Tool), developed by the Artificial Intelligence and Decision Aids Division of Science Applications International Corporation (SAIC), is a tool for building simple expert systems. Concurrent GESBT is an enhanced version of GESBT that supports multiple, cooperating expert systems on a concurrent computer such as the iPSC system. The concurrent version offers a method of sharing objects and moving objects between multiple expert systems. It also enables an expert system to create another expert system.

MACE

MACE (Multi-Agent Computing Environment) is a development and execution environment for distributed agents. Agents are intelligent entities, capable of performing simple tasks. The MACE system is both a tool set and a test bed for the programming model of cooperating agents.

Distributed agents, under MACE, form the basic building blocks for a wide variety of symbolic parallel programming models. MACE is essentially a tool for programming concurrent computers in an object-oriented style. Already, demonstrations of programming models such as Contract Net, Distributed Blackboards, and Rule-Agents (Rule-based systems) have been written in MACE.

Demonstration programs for MACE include: solving the N-Queens problem and a multi-robot blocks world problem.

Product Brief

Intel Scientific Computers

SugarCube™
Introducing Concurrent Computation

Intel Scientific Computers

INTRODUCTION

The rapidly growing power of computers affects every branch of mathematics, science and engineering. In the future, this growth will be based on parallel processing. This has recently been recognized by the NSF initiative in parallel supercomputing. The development of parallel computing technology has been accomplished by a team effort between universities which have performed the research, government which has provided the funds, and industry which has transferred the research results into the commercial marketplace.

As an industrial contributor, Intel is leading the commercialization of large-scale concurrent computers. From its inception, Intel has created many of the technologies which have fueled the computer revolution. Intel supported many of the university research projects in parallel processing, including the Concurrent Computing Project at Caltech. In 1985, ISC announced and delivered the first commercial computer based on the Caltech hypercube architecture. It is now installed and in use at 30 universities.

The growing knowledge of concurrent computing techniques needs to be provided to students and faculty involved in computational science in every field. Universities are now beginning to develop and offer courses in concurrent computing as well as to provide student access to concurrent computers. For example, see the product brief, *"Introduction to Concurrent Computing"*, which gives an example course description.

Intel is continuing its leadership by providing the products and services needed to support this effort. ISC has recently introduced the SugarCube, a low-cost concurrent workstation. Available with the SugarCube are two operating environments each of which demonstrates an important alternate strategy for concurrent computation. The use of the SugarCube in education is supported by:

- Intel's factory training
- access to the experience of colleagues who have developed and presented course in concurrent computing through the Users' Group
- tested examples of concurrent programming from the Users' Group library
- comprehensive professional support from ISC customer service.

SUGAR CUBE

The SugarCube is a complete system consisting of a concurrent computer (the SugarCube) plus a multi-user development workstation, complete programming support for FORTRAN and C, and networking capability to UNIX™ system via TCP/IP. SugarCube programs are capable of running in production on larger iPSC™ systems without change.

There are four system configurations, each for a specific purpose. The standard system provides the largest degree of parallelism, 8 processors with 512 KBytes of memory each. The extended memory system provides 4 processors with 4.5MBytes of memory each. It includes CCLISP™ (Gold Hill's Concurrent Common LISP). The vector system provides 4 vector processors each capable of performing floating point operations at 100 times the rate of a standard processor. The hybrid system provides 2 extended memory processors and 2 vector processors. It includes CCLISP and is designed to support teaching and research in both numeric and symbolic computing. This configuration supports advanced research in applications which require artificial intelligence techniques to make decisions based on the results of high-speed computing.

NX

Intel's NX Operating System provides a flexible execution environment for the SugarCube. NX allows programs to send messages directly to any node, not just nearest-neighbors. It allows overlap of computation and communication. NX supports both FORTRAN and C as well as CCLISP in the Extended Memory and Hybrid systems. The NX software includes a concurrent debugger for FORTRAN. NX supports the VAST-2™ vectorizing preprocessor and VecLib for Vector and Hybrid systems.

CRYSTALLINE OPERATING SYSTEM (CrOS III)

Caltech's Crystalline Operating System (CrOS III) is available for the SugarCube (Standard and Extended Memory). CrOS III provides fast nearest-neighbor communications and a simple programming model based on synchronous communications with no overlap of computation and communications.

"Solving Problems on Concurrent Processors", a text by Geoffrey Fox and his associates at the Caltech Concurrent Computing Project, will be available in Fall 1987 from Prentice-Hall. This text describes the hypercube architecture, the design of the Crystalline Operating System, and the basic performance considerations for concurrent computing. It also includes complete examples for a wide range of scientific applications. The C source code for these examples is also available for the SugarCube.

TRAINING

Intel Scientific Computers offers a 5-day course for its iPSC™ customers including users of the SugarCube. In addition, university faculty can arrange through their local iSC representative to attend the course at no cost. Faculty who attend the course and who plan to develop a course in concurrent computing can arrange to get a free copy of the iSC course materials.

iPSC USERS' GROUP

Intel Concurrent Computing Systems are installed in 30 universities. A Users' Group has been formed. The Users' Group meets at the annual Hypercube Conference to provide a forum for the exchange ideas and methods. The Users' Group Library provides a repository for codes which demonstrate computational techniques using the iPSC. Many of the library items can be used as examples for courses in concurrent computing. In addition, members of the Users' Group are ready to work with new users to share the benefits of their experience. Some of these users have already developed and offered courses in concurrent computing. These users may be able to help with suggestions, course materials, and programming examples.

INTEL CUSTOMER SERVICE

Intel provides comprehensive service and support for its systems. The larger-scale iPSC systems are installed by a factory installation team. The Sugar Cube is installed by the user with the help of a comprehensive step-by-step manual. On-site service is available from Intel Customer Service Engineers who have been specially trained in the iPSC and the SugarCube. Telephone "hot line" help is available from the factory to respond to questions or problems in the use of the iPSC or SugarCube systems.

Introduction to Concurrent Computing

Intel Scientific Computers

The following description describes one possibility for a course in concurrent computing to be offered to graduate and advanced undergraduate students in science and engineering.

CONCURRENT COMPUTING COURSE

This course is intended for students from the fields of mathematics, science, engineering, and computer science who want to understand current developments in the computational methods for parallel computing. Prerequisites: computer organization, algorithms and data structures, calculus including differential equations, and ability to program in C or FORTRAN in a UNIX™ environment. Course includes laboratory projects using the Intel SugarCube. Projects will be based on student's major field and area of interest.

- **Parallel Architectures and Interconnect Structures**
  - **Architecture implementation and performance tradeoffs**
  - **Cost, complexity, reliability**
  - **Application strengths and weaknesses of each architecture**
  - **Implementation of various architectures on a hypercube (performance comparison)**
- **Software development techniques for concurrent systems**
  - **Parallel sorting and search algorithms**
  - **Concurrent programming methodologies**
    - **Message passing**
    - **Problem decomposition and mapping to the architecture**
    - **Performance analysis for different decompositions and functional mappings**
  - **Programming languages for concurrent/parallel systems**
- **Performance evaluation of concurrent computation**
  - **Load balancing**
    - **dynamic load balancing**
    - **'static' load balance - wrap method, strips, etc.**
    - **relationship of load balance to system performance**
- **Computation/communications ratio**
- **Performance measures for concurrent processing**
- **Computational complexity and concurrent processing**
- **Concurrent computational methods**
  - **Matrix computations**
    - **Problem decomposition**
    - **Performance analysis**
  - **Partial differential equations**
    - **Problem decomposition**
    - **Performance analysis**
  - **Statistical/approximation techniques**
    - **Simulated annealing**
    - **Monte Carlo simulation**

Product Brief

Intel Scientific Computers

Vector Concurrent System

Intel Scientific Computers

INTRODUCTION

Intel's Vector Concurrent Computers provide a dual approach to achieve maximum performance. High level parallelism speeds up performance by using multiple processors operating concurrently to solve a problem. Low-level parallelism is provided by a pipelined arithmetic accelerator at each node which supports high-performance computations for vector and matrix applications.

Intel Scientific Computers (ISC) has introduced the Vector SugarCube™, a solution to the need for a low-cost system to support research and education in parallel and vector computing techniques. SugarCube systems are provided with the same development environment and programming tools used with the larger iPSC™ systems. Consequently, programs developed on a SugarCube can be used to solve larger production problems on full-scale iPSC systems with true supercomputer performance.

The performance of LINCUBE, Intel's parallel version of LINPACK on the SugarCube, is an example of the benefits of this dual approach. The routines DGEFA and DGESL from LINPACK combine to factor and solve a system of linear equations in double precision (64-bit). On a 300 by 300 matrix, a single vector node achieves 2.3 MFLOPS on DGEFA and 1.06 MFLOPS on DGESL. A Vector SugarCube with four such vector nodes executing concurrently can factor and solve a 600 by 600 matrix at a rate of 7 MFLOPS. The vectorization and concurrency each contribute to the overall performance.

The peak performance for a single vector processor is 6.67 MFLOPS for a double-precision (64-bit) operand and 20 MFLOPS for a single-precision (32-bit) operand. This gives a combined peak performance for a vector SugarCube of 23 MFLOPS and 80 MFLOPS respectively.

VECTOR CONCURRENT PROGRAMMING

The first step in developing an application for a vector concurrent computer is to define a concurrent programming strategy. The objective is to divide the problem evenly so that data and computation are balanced between the processors. The result should be a program which can handle a range of problem sizes with larger problems being processed on larger dimension iPSC systems as efficiently as smaller problems are handled by the SugarCube.

The program which executes on the node processors is a conventional sequential FORTRAN program. Concurrent processing is controlled by messages transferred between the nodes. Performance of the node program can be greatly improved by using vector processing techniques, as much as 100 times for vector floating point operations. The vector processor can also speed up scalar operations by a factor of 10.

The tools that support vector programming are:

- **VAST-2** preprocessor for vectorizing FORTRAN
- **VecLib** library of FORTRAN callable vector routines
- **Microcode** tools for writing microcode for the vector processor

VAST

VAST-2 serves as an optimizing FORTRAN compiler for the numeric accelerators on each node of the iPSC-VX and Vector SugarCube systems. At the user interface, it accepts FORTRAN 77 and 8x vector syntax and builds optimized code for execution on each node. Speedups by a factor of 10 (scalar operations) and by a factor of 100 (vector operations) are possible by using VAST-2.

VAST-2 analyzes standard FORTRAN application code, detects vector constructs, and translates those vector operations into functions that can be executed on an iPSC-VX vector processor board. VAST-2 detects arithmetic operations with one, two, or three operands as well as reduction operations, like vector sum or dot product, that are supported in VecLib. The vectorized FORTRAN output is translated into vector and scalar command operations that drive the vector processor. These command sequences can be combined into a single iPSC-VX execution module that executes entirely on the vector processor. VAST-2 is a sophisticated, modern vectorizing compiler. For example, VAST-2 will:

- Interchange nested DO loops
- Re-roll DO loops
- Promote common sub-expressions
- Collapse nested DO loops
- Split DO loops into vectorizable and scalar DO loops

VECTOR LIBRARY - VecLib

VecLib is a math library for the iPSC-VX vector processor. It is an expanding library of useful routines that can be called from any FORTRAN program. It contains the Basic Linear Algebra Subroutines (BLAS) plus a variety of additional mathematical functions, e.g., a 2-dimensional Fast Fourier Transform operator and a tridiagonal matrix solver. All of the functions and operations have been organized to conform to the BLAS calling and naming conventions to ensure consistency. It supports both single and double precision operations and a full complement of integer and logical routines. VecLib operations achieve maximum performance because each of the routines in the library are microcoded functions that reside in the vector processor.

VecLib is also available in a version for standard iPSC nodes and a version in transportable FORTRAN for use with the iPSC Simulator. These versions of VecLib allow the use of standard iPSC and SugarCube systems for developing application programs and debugging programs ultimately targeted for the iPSC-VX.

MICROCODE

In some applications, a small kernel of code or a single DO loop will be responsible for the majority of work. There are tools available for writing this portion in microcode to take full advantage of the vector node. For example, a 15-line microcode program improved the performance of a second-order difference operator by a factor of four. The ultimate control of the vector processor rests in the microcode executing on the board. Included with the Vector system is a set of tools including an assembler, linker/loader, library editor, and debugger. With these tools, custom routines can be developed for the vector processor that achieve maximum performance.

SUMMARY

The Intel iPSC-VX and the Vector SugarCube, combine two approaches to maximum computer power. High-level parallelism can be exploited to obtain multiples of the performance of a single processor. The Vector Processor provides an opportunity to increase the performance of each processor one or two orders of magnitude by using low-level parallelism found in the mathematics of the computation. VAST-2, VecLib, and microcode provide a hierarchy of techniques for reaching the maximum performance of the vector processor. The combination, vector concurrent computing, results in an order of magnitude improvement in cost/performance ($/MFLOP) over today's supercomputers.

Functional & Logic Programming with Large-Scale Concurrent Computers

Intel Scientific Computers

The following papers concern the user of logic and functional languages with large-scale concurrent computers.

Notes on the Complexity of Systolic Programs	Hellerstein, L., Shapiro, E. Safra, S. and Taylor, S.	Weizmann Institute of Science, Nov. 1985
The Design and Implementation of Flat Concurrent Prolog	Mierowsky, Taylor, Shapiro, Levy, and Safra	Weizmann Institute of Science Rehovot, Israel Technical Report CS85-09, July 1985
Systolic Programming: A Paradigm of Parallel Processing	Shapiro, E.	Weizmann Institute of Science Rehovot, Israel Technical Report CS84-16, January 1985
Concurrent Prolog A Progress Report	Shapiro, E.	Computer, Aug., 1986: Vol. 19, No. 8.
Parallel Logic Programming Languages	Takeuchi, A. and K. Furukawa	Proceedings of the Third International Logic Conference, July, 1986. ICOT Research Center.
A Layered Method for Process and Code Mapping	Taylor, S., Av-ron, and E. Shapiro	Department of Computer Science Weizmann Institute of Science Rehovot, Israel Technical Report CS86-17, March, 1986

Distributed Memory Architectures for Artificial Intelligence

Intel Scientific Computers

The following papers concern distributed-memory and message-passing computer architectures for artificial intelligence research and applications.

Concurrent Programming Using Actors: Exploiting Large-Scale Parallelism	Agha, G., and Hewitt, C.	MIT Press A.I. Memo No. 865, October 1985
The Architecture of the FAIM-1 Symbolic Processing System	Davis, A., and Robinson, S.	Proceedings of the Ninth International Conference on Artificial Intelligence, August 1985
Rule Agents: A Distributed, Object-Oriented Approach to Production Systems Using MACE	Gasser, L. and Tenorio, M.	Working paper, USC-DPS Group Computer Science Department University of Southern California, March, 1985
Open Systems	Hewitt, C., and de Jong, P.	MIT Press A.I. Memo No. 691, December, 1982
The Apiary Network Architecture for Knowledgeable Systems	Hewitt, C.	Conference Record of the 1980 Lisp Conference Stanford University, August, 1980
Communicating Sequential Processes	Hoare, C.A.R.	Communications of the ACM, August, 1978, Vol. 21, No. 8, pp 666-677
On "Hot Spot" Contention	Lee, R.	Computer Architecture News Vol. 13, No. 5, December 1985
Architecture and Applications of DADO: A Large-Scale Parallel Computer for Artificial Intelligence	Miranker, D., Shaw, D.E., and Stolfo, S.	Columbia University
Problem-heap: A Paradigm for Multiprocessor Algorithms	Moller-Nielsen, P. and J. Straunstrup	
"Hot Spots" Contention and Combining in Multistage Interconnection Networks	Pfister, G.F., and Norton, V.A.	IEEE Transactions on Computers, Vol. C-34, No. 10, Oct. 1985, pp 943-948
Systolic Programming: A Paradigm of Parallel Processing	Shapiro, E.	Weizmann Institute of Science Rehovot, Israel Technical Report CS84-16, January 1985
A Variable Supply Model For Distributing Deductions	Singh, V. and Genesereth, M.	Proceedings of the Ninth International Conference on Artificial Intelligence, August, 1985

Concurrent Applications on the Intel iPSC™

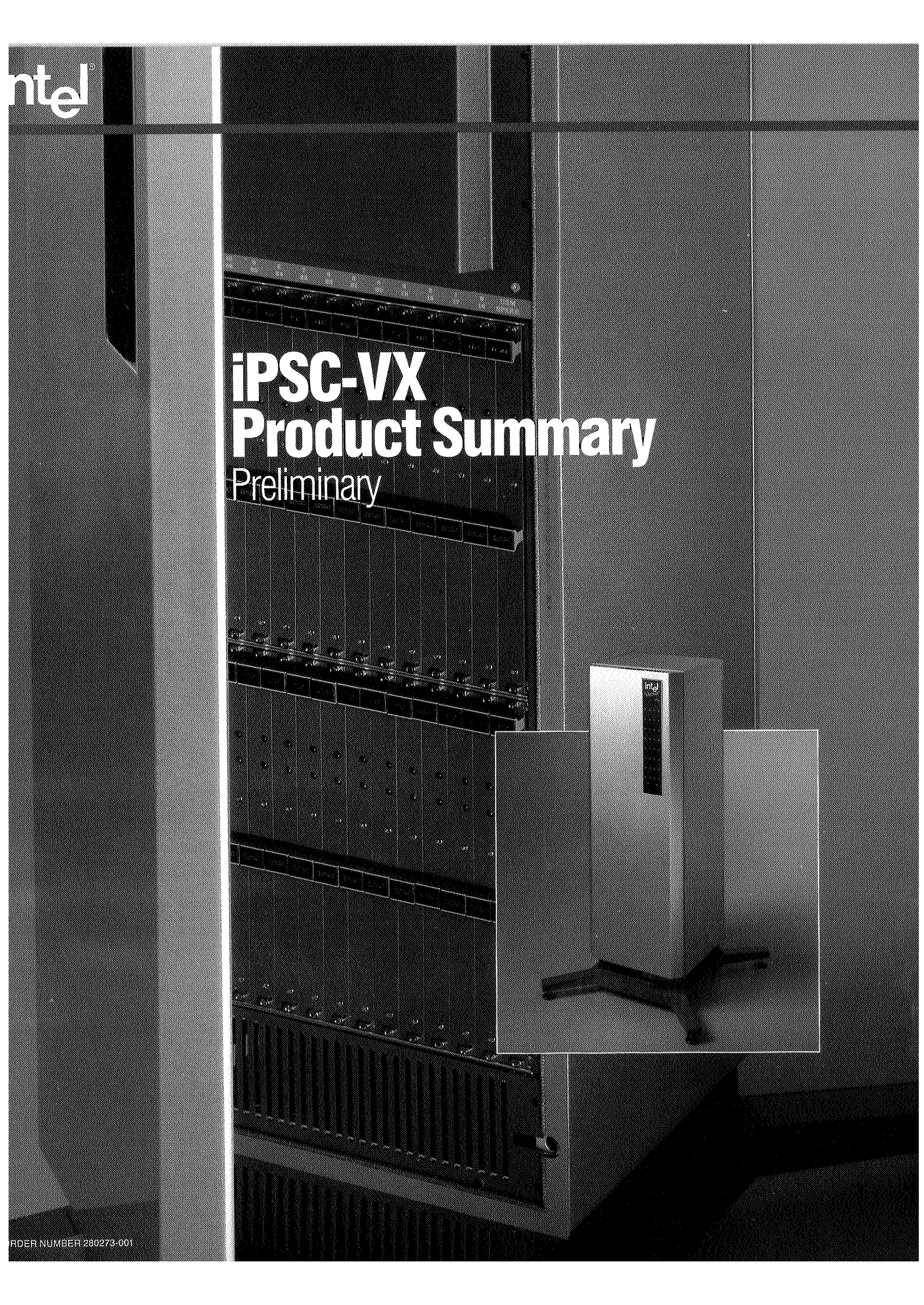
Intel Scientific Computers

The following papers describe the concurrent computing strategies for a wide range of applications for Intel's iPSC™ computer.

Numerical Simulation of the Navier-Stokes Equations Using the Intel Hypercube	Bruno, J.	Department of Computer Science University of California at Santa Barbara
Concurrent Stochastic Methods for Global Optimization	Byrd, R., Dert, C., Kan A., Schnabel, R.	Technical Report Department of Computer Science University of Colorado, Boulder
Multigrid Algorithms on the Hypercube Multiprocessor	Chan, T.F. and Saad, Y.	Department of Computer Science Yale University
Parallel PISCES	Lucas, R. and Blank, T.	AEL 231 D Integrated Circuits Laboratory Stanford University
Performance of a Parallel Algorithm For Standard Cell Placement on the Intel Hypercube	Jones, M. Banerjee, P.	Computer Systems Group Coordinated Science Laboratory University of Illinois at Urbana-Champaign
Parallel Computers for Finite Element Analysis	Ercal, F., Sadayappan, P., Schwan, K., Weide, B., Aykanat, C., Doraivelu, S.	Universal Energy Systems, Inc. Air Force Materials Laboratory Wright Patterson Air Force Base Dayton, Ohio

iPSC is a trademark of Intel Corporation

Dynamic Load Balancing of a Vortex Calculation Running on Multiprocessors	Baden, S.	Computer Science Division and Lawrence Berkeley Laboratory University of California at Berkeley
Multiprocessing Monte Carlo Codes on Distributed and Common Memory Computer Systems	McKinney, G.	Department of Nuclear Engineering University of Washington
Using Hypercube Multiprocessors to Determine Geometric Properties of Digitized Pictures	Miller, R. and Miller, S.E.	Department of Computer Science State University of New York Buffalo, New York
Adaptation of a Large-Scale Computational Chemistry Program to the Intel iPSC Concurrent Computer	Larrabee, A.R.	Oregon Graduate Center
Details Concerning the Implementation of the Fast Fourier Transform on the Intel iPSC Hypercube	Chu, C.Y.	
Solving Elliptic Partial Differential Equations on the Hypercube Multiprocessor	Chan, T., Saad, Y., Schultz	Computer Science Department Yale University



iPSC-VX Product Summary

Preliminary

ORDER NUMBER 280273-001

The iPSC-VX Vector Concurrent Supercomputer

The iPSC™-VX (vector extension) is a vector concurrent computer system, an enhanced member of the iPSC family of "personal supercomputers." The iPSC-VX offers true supercomputer performance for the individual scientist or research team, and provides the tools for developing efficient, high-performance applications in a large-scale parallel computing environment.

The cost of owning and maintaining a conventional supercomputer has made high-end scientific computing inaccessible to all but a privileged few. The iPSC-VX changes this. By using low cost VLSI to harness the complementary technologies of concurrent and vector processing, the iPSC-VX sets a new price-performance standard. The result is supercomputer performance at the price of a supermini.

The iPSC-VX family builds upon the basic architecture of the iPSC concurrent computer. By coupling a high-performance vector processor to each of the iPSC processing nodes, the vector enhancement results in dramatically improved computational performance for both vector and scalar operations. Optimized to meet the mathematical requirements of scientific computation, the iPSC-VX family is ideal for such applications as circuit simulation, structural analysis, fluid dynamics, and oil reservoir modeling.

An iPSC-VX system (Figure 1) consists of 16, 32, or 64 computational nodes. Each node consists of an independent microcomputer with its own CPU, communications control, local memory, and dedicated vector processor. In keeping with the basic iPSC architecture, the processing nodes in a system are interconnected using a hypercube topology where connected nodes are supported with reliable point-to-point message delivery service. The multiple nodes which comprise a system are collectively known as the "Cube" and may be housed in one, two, or four computational units. The Cube Manager, an Intel System 310 supermicrocomputer, provides a gateway to the system and serves as a convenient software development station.

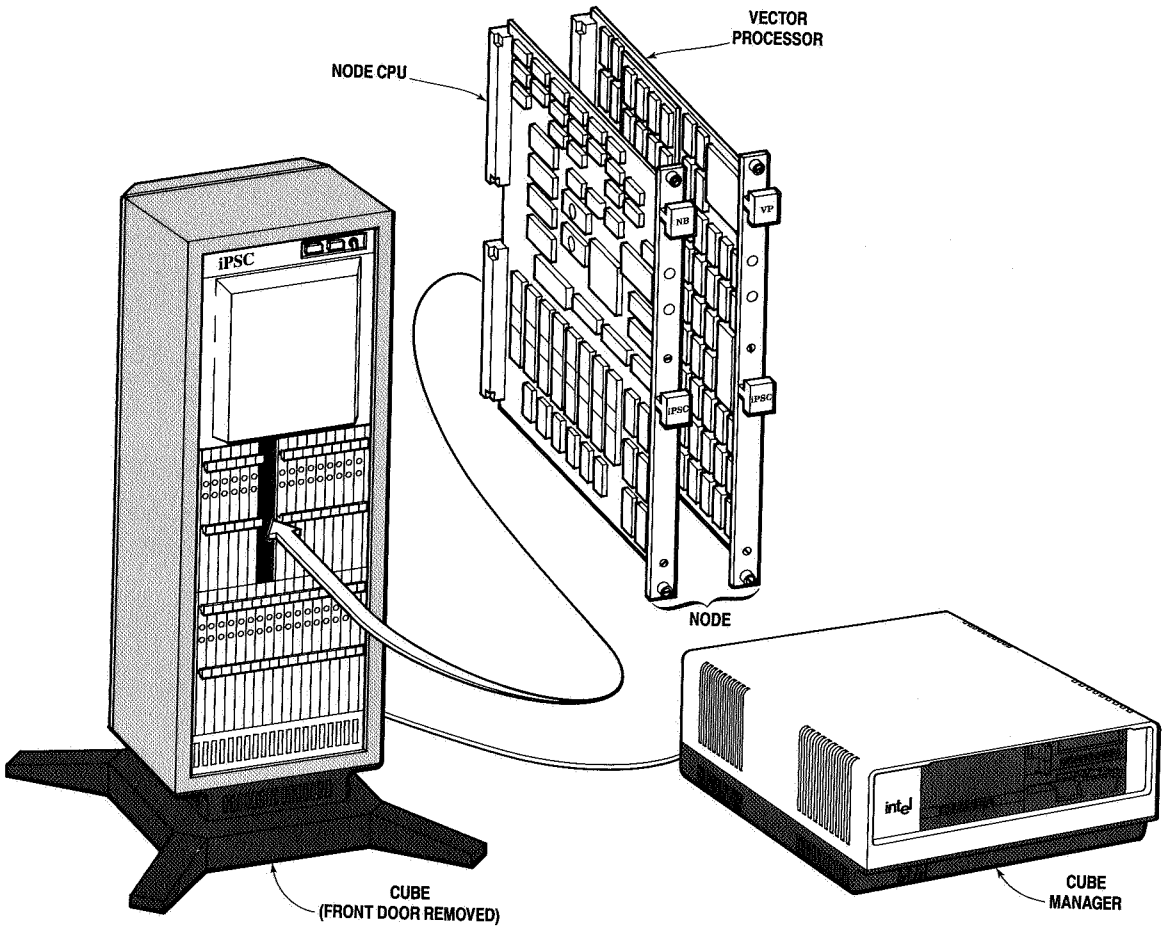


Figure 1—iPSC-VX System

iPSC-VX FEATURES

Vector Concurrent Architecture	Uses low-cost VLSI to capture the benefits of both vector and concurrent processing to ensure the lowest possible cost per calculation.
Supercomputer Performance	A high-performance vector processor is tightly coupled to each of the system's computational nodes, boosting (64-bit) peak performance to 424 MFLOPS on a 64-node iPSC-VX/d6.
Large Memory	1.5 MBytes per processing node meets the memory requirements for efficient computation. Total system capacity ranges from 24 MBytes to 96 MBytes, depending on system size.
Field Upgradable and Expandable	A 16-node iPSC-VX is field upgradable to 32 or 64 nodes. Standard iPSC systems can also be upgraded to any of the iPSC-VX system configurations.
UNIX-Based Development Environment	An enhanced System 310 Cube Manager with its XENIX 3.0 operating system is the primary host for supporting system diagnostics and program development. Its standard configuration includes 3 MBytes of system RAM, a 140 MByte hard disk, a floppy disk, and a 45 MByte cartridge tape drive.
Professional FORTRAN	Opimizing FORTRAN compiler from Ryan McFarland offers complete GSA certified implementation of the FORTRAN-77 standard plus popular VAX and VS extensions.
Vector Programming Support	Standard FORTRAN programs can be automatically converted to vector routines using the VXP vectorizing preprocessor. Explicit vector programming is also supported through a powerful library of matrix, vector and scalar math functions.
Network Support	Intel's MULTIBUS®-based System 310 provides a convenient gateway to users of the Cube from other systems and networks via TCP/IP protocols.
Performance Monitoring Display	Convenient front-panel monitor can be used to display processing, communications, and computational activities for each node. Provides the user with instant feedback on system runtime behavior, load balancing, and fault diagnosis.
Lab Environmental Design	Compact packaging and a quiet air-cooled design allows convenient placement in laboratory environments.
Low Mean Time-To-Repair	Systems are provided with spare vector and/or node processor boards to reduce mean time-to-repair.

SYSTEM OVERVIEW

The iPSC concurrent computer was designed to accommodate a wide variety of enhancement options. Within the backplane of each iPSC computational unit, adjacent odd and even card slots have been connected to form a high speed local bus. This MULTIBUS II iLBX™ bus allows each of the iPSC-VX node processors to be tightly coupled to a high performance vector processor. The resulting two board node is capable of 100 times the (64-bit) vector performance and 10 times the (64-bit) scalar performance of a standard iPSC node.

Figure 2 illustrates the basic architecture of an iPSC-VX computational node. The 80286-based node processor board (shown on top) serves as a general purpose microcomputer. It contains 512 KBytes of local memory and hosts a small message-based operating system called MBOS. The node processor with its operating system are primarily responsible for coordinating message traffic into and out of the node, for scheduling and executing user processes, and for controlling its companion vector processor. The vector processor (VP) is a highly optimized arithmetic processing unit that can perform complex mathematical functions in tandem with node CPU operations.

Message SEND and RECV operations, which are supported by MBOS, provide the iPSC family with its basic means for interprocess communications. Figure 2 illustrates the steps this message delivery service uses to bring computational data into a node for processing by the VP:

1. A message sent from an adjacent node is received over one of the serial communication ports and is automatically deposited in a message buffer.
2. If a request for the message is pending (or is made some time later), MBOS will then transfer the data from the message buffer to the requesting user process. (A user process requests its message by executing a CALL RECV procedure which specifies a message ID and a destination address in the VP's data memory.)
3. Once the data is available to the user process, it issues commands to the vector board to initiate computation and to store results at the intended location.

   (These computational results can be passed to other nodes following a similar sequence of events by using the analogous CALL SEND procedure.)

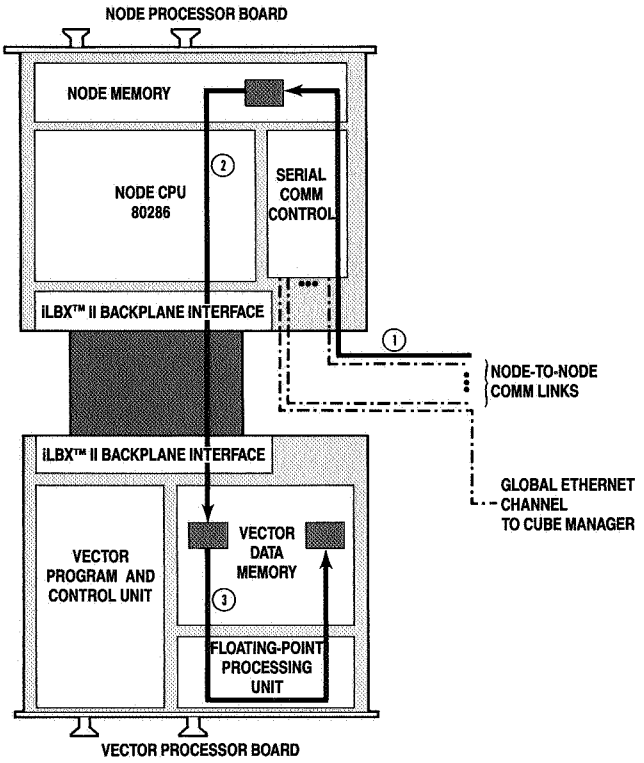


Figure 2—iPSC-VX Computational Node

A key feature of the iPSC-VX node architecture is the dual ported access to vector memory, which is shared between the node CPU and the vector processor (Figure 3). The system's linker/loader ensures that all user data is placed on the vector board where it is accessible to both the 80286 and the VP. This allows the vector processor to perform in-place calculations on applications data. The convenience of this tightly-coupled shared memory architecture contrasts sharply with the approach used in "attached" floating-point processors. Attached processors typically require explicit programmer intervention to queue up data for vector processing. The iPSC-VX provides this service as an automatic system-level function without incurring additional overhead.

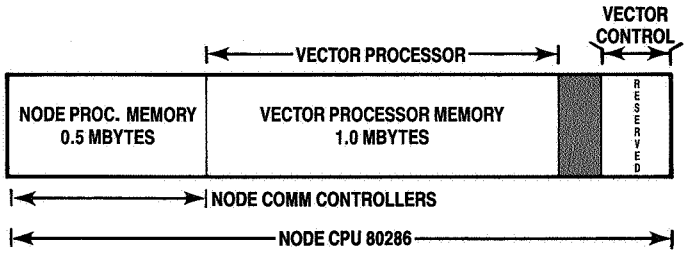


Figure 3—VP/Node Board Memory Access

VECTOR PROCESSOR DESCRIPTION

The iPSC-VX vector processor (VP) board has a synchronous microprogrammed architecture with a 100 nanosecond cycle time. The VP board's three main functional components are a floating-point Arithmetic Unit, a Data Unit, and a Control Unit (Figure 4).

Arithmetic Unit

The arithmetic unit is a pipelined processor composed of an adder and a multiplier, each with the performance capability shown below. The arithmetic unit also contains twelve registers which are used to store intermediate results and to stage data into the arithmetic elements. Except for gradual underflow, all floating-point operations conform to the IEEE-754 standard. The arithmetic unit also supports 32-bit fixed point and logical operations.

Floating Point Throughput

	32-bit	64-bit
Adder (elapsed/pipelined)	300/100 ns	300/100 ns
Multiplier	300/100 ns	500/300 ns

Data Unit

The VP data unit is composed of a 32-bit address Arthmetic Logic Unit (ALU) and 1 MByte of data RAM. Data RAM is mapped into the address space of the node processor bringing its total memory capacity to 1.5 MBytes. This memory space may be used by the 80286 to store user code as well as data. Computational operands for the VP must, however, reside in the one megabyte of VP data memory.

The vector processor's local data RAM supports 64-bit accesses in a cycle time of 250 nanoseconds. This memory is augmented by a 16 KByte section of fast (static) RAM, which is capable of 64-bit accesses in 100 nanoseconds. Fast RAM is partitioned into a 4 KByte user-accessible scratch pad and 12 KBytes of high performance storage for intermediate computational results and table functions. The fast scratch pad memory is accessible to the programmer through named FORTRAN COMMON statements. When properly allocated to frequently used scalar or short vector operands, the use of scratch pad memory can significantly improve execution time.

Control Unit

The control unit contains a microsequencer and a 32 KByte program RAM that is directly loadable from the 80286 node CPU. Program RAM contains instructions for the microsequencer that allow it to coordinate operation of the VP's data and arithmetic units. These microcoded routines include a runtime monitor and VecLib, an optimized library of vector, scalar, and logical operations.

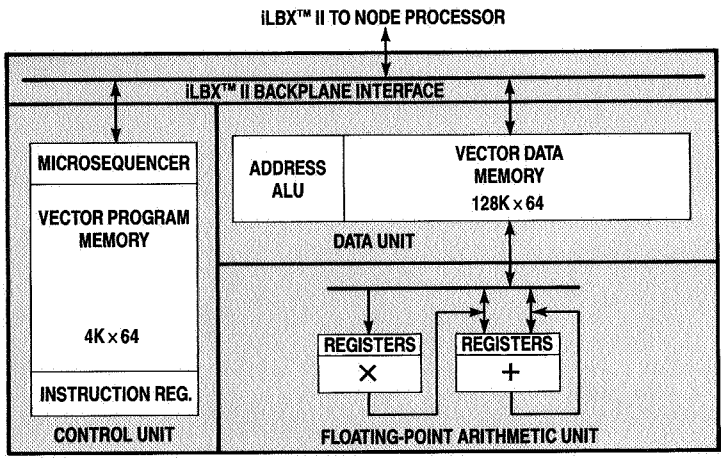


Figure 4—Vector Processor Organization

PERFORMANCE

By operating from fast data RAM, the vector processor board is capable of reaching peak performance levels of 6.67 MFLOPS (64-bit) and 20 MFLOPS (32-bit). This brings the 64-bit peak system performance to 106 MFLOPS for a 16 node iPSC-VX/d4, 212 MFLOPS for a 32 node iPSC-VX/d5, and 424 MFLOPS for the high-end iPSC-VX/d6. Peak performance at 32-bit precision ranges from 320 MFLOPS to 1280 MFLOPS.

Table 1. Node Computational Performance

Node Vector Performance for operands in standard/fast RAM

Operation	Integer (MOPS)	32-bit FP (MFLOPS)	64-bit FP (MFLOPS)
ADD/SUB/MLT	2.6/6.2	2.6/6.2	1.3/3.2
DIVIDE	0.52/0.52	0.52/0.52	0.28/0.28
SIN/COS	0.47/0.47	0.47/0.47	0.24/0.24
DAXPY[1]	8.8	8.8	4.4
DOT PRODUCT	9.1/18.2	9.1/18.2	3.3/6.7
COMPLEX (1024) FFT[1]	10.0	10.0	N/A

Node Scalar Performance for operands in fast data RAM

Operation	Integer (MOPS)	32-bit FP (MFLOPS)	64-bit FP (MFLOPS)
ADD/SUB	.35	.35	.35
MULT	.35	.35	.33
DIVIDE	.13	.13	.10
SIN/COS	.07	.07	.07

MOPS (millions of operations per second)
MFLOPS (millions of floating-point operations per second)

[1]For operands in both standard and fast data RAM.

PROGRAM DEVELOPMENT AND SOFTWARE

The program development environment for the iPSC-VX series is identical to that of the original iPSC family, combining FORTRAN development capabilities with simulation and debug facilities in a XENIX operating environment. Thus, a programmer can apply familiar, well-developed programming techniques when building applications for the entire iPSC family. The complete iPSC-VX program development package includes iPSC development tools, an optimizing FORTRAN compiler, a FORTRAN Vectorizing Preprocessor, a simulator, a microcode development tool kit, and the VecLIB math function library.

iPSC	iPSC-VX
1. Develop Program	1. Develop Program
2. Compile Program	2. Vectorize with VXP
3. Load Object	3. Compile Program
4. Run Program	4. Load Object
	5. Run Program

FORTRAN Vectorizing Preprocessor

Harnessing the vector capability of the iPSC-VX series requires a simple addition to the normal software development process: FORTRAN code vectorization. The FORTRAN Vectorizing Preprocessor (VXP) makes vector programming automatic. Starting from a standard FORTRAN-77 program, VXP will extract the computational portions of the problem and partition them off to the vector processor. This is accomplished by substituting vector operations for vectorizable code segments. Mathematical statements and routines that appear in the original source code are "commented out," and directives are substituted which call upon the vector processor to perform the work.

VXP supports the following performance accelerating capabilities:

1. **DO Loop Vectorization**

Actual Source	Equivalent Conversion[2]
DO 10 I = 1,1024 C(I) = A(I)*B(I) 10 CONTINUE	CALL VMUL(A,1,B,1,C,1,1024)

VXP can recognize standard DO loop constructions and convert them to equivalent vector operations.

2. **Scalar Assignment**

Scalar operations can also be converted into directives for the VP, for example:

X = A * (B–C)/SQRT(D)

The VP board is capable of ten times the scalar performance of the node board's 80287 math coprocessor, and so the scalar conversion is usually desirable.

3. **Explicit CALL statements to VecLib**

VecLib is the VP's internally microcoded mathematical function library. It contains approximately 100 microcoded mathematical routines which perform optimized scalar, vector, and matrix operations. VecLib is made accessible to the user to allow "hand tuning" of critical computational algorithms. Its various mathematical functions may be envoked through standard FORTRAN subroutine calls, for example:

CALL VSQRT(A,1,B,1,1024)

4. **Command Subroutines[3]**

VXP can also be used to take entire computational segments, including non-vectorizable loops, IF statements and GOTO statements, and convert them into a sequence of VP directives called a Command Subroutine. The node program can then trigger execution of this entire "subroutine" by issuing a single command word to the VP. Command subroutines make maximum use of the VP's computational hardware. In the extreme, the node processor may perform little more than the functions of a dedicated I/O processor.

[2]VXP would actually express the CALL VMUL procedure as a series of VP directives.
[3]Available Q4, 1986

FORTRAN Compilation

Since FORTRAN is the language of choice for high-end scientific computing, the iPSC-VX provides a complete FORTRAN support package. The Ryan-McFarland RM/FORTRAN™ compiler is supplied with each system. RM/FORTRAN is a full implementation of the FORTRAN-77 standard including all of the popular VAX and VS extensions. It imposes no limits on the size of executable programs or arrays and includes several levels of optimization to ensure fast execution. Program development is facilitated by the availability of an interactive symbolic debugger (on the Cube Manager) and by an extensive set of compiler diagnostics. RM/FORTRAN has been certified by the Government Services Administration (GSA) as being complete and error free.

Simulation

The iPSC simulator is available on both the iPSC Cube Manager and Unix 4.2 BSD systems. It provides an iPSC node execution environment where carefully controlled program debugging can take place prior to execution on the Cube. The iPSC simulator has been extended with a library of routines to implement the VP's VecLib functions.

Microcode Development[3]

Intel provides microcode development tools as an option to allow the user to add customized routines to the VP microcode library. This capability may be desirable for proprietary applications that must run with very high efficiency or which are characterized by short vectors.

[3]Available Q4, 1986

SYSTEM SPECIFICATIONS

Cube	iPSC-VX/d4	iPSC-VX/d5	iPSC-VX/d6
Number of Nodes	16	32	64
Peak System Performance (MFLOPS)			
64-bit	106	212	424
32-bit	320	640	1280
Total System Memory (MBytes)	24	48	96
Internode Communication Links	32	80	192
Spare Boards			
Node Processors	0	1	2
Vector Processors	1	1	2
Cube Footprint			
(in)	26.75 x 26.75	42.7 x 26.75	74.5 x 26.75
(cm)	67.96 x 67.95	108.46 x 67.96	189.23 x 67.95

Cube Manager

Central Processor (CPU)	Intel 80286
Numeric Processor (NPU)	Intel 80287 32-, 64-, 80-bit floating point (IEEE 764) 32-, 64-bit integer 18-digit BCD operands
Memory	3 MByte memory with ECC, expandable to 5 MBytes
Mass Storage	140 MByte 5¼″ Winchester disk 360 KByte 5¼″ floppy disk (DS, DD) 45 MByte cartridge tape drive
MULTIBUS Expansion Slots (IEEE 796)	3 available slots at 0.6″ spacing 1 reserved for extra memory, 2 for standard MULTIBUS I cards 93 watts available power
Printer Port	Centronics compatible
User Processes Supported	50 (XENIX configuration parameter)
Global Channel	Ethernet (IEEE 802.3)

Console Terminal

WYSE Model 75	See iPSC System Product Summary for details

Node Processor (NB) Board

Node CPU	Intel 80286 16 MBytes physical addressing memory management and protection
Numeric Coprocessor	Intel 80287 32-, 64-, 80-bit floating-point (IEEE 754) 32-, 64-bit integer 18-digit BCD operands

Node Processor (NB) Board (cont.)

Memory	512 KBytes dual-port RAM with byte parity 64 KBytes PROM monitor (expandable)
Communications Channels	Eight per node. Six to nearest neighbor nodes, 1 Ethernet channel to the Cube Manager and 1 reserved
Communications Bandwidth	Hardware: 20 Mbits/sec. maximum
iLBX II Port	Provides 80286 node CPU with 1 wait state memory-mapped access to VP program and data RAM. Supports 8- and 16-bit access
Indicators	System directed or user programmable red and green LEDs are viewable from board's face plate or from the system's front panel display
Size	2 x 4 Eurocard (9.2″ x 11″) with two 96-pin male DIN connectors

Vector Processor (VP) Board

Arithmetic Unit	ALU: 100 nsec cycle time, pipelined operation 32-bit/64-bit floating-point (ADD,SUB) 32-bit integer (ADD,SUB,AND,OR,XOR) Multiplier: 100 nsec (32-bit) cycle time or 300 nsec (64-bit) cycle time, pipelined operation 32-bit/64-bit floating-point MULTIPLY
Program Memory	32 KBytes (4K x 64), 50 nsec static RAM Directly loadable through iLBX II bus by 80286 node CPU Supports VecLib function library and runtime monitor
Data Memory	1.0 MBytes (256K x 32), 250 nsec cycle time dynamic RAM Dual ported, can be directly addressed by the VP's dedicated 32-bit Address ALU, or through the iLBX II bus by the 80286 node CPU (on a cycle stealing basis). Extends the 80286 node memory space to 1.5 MBytes. Used as a shared data workspace for the node CPU and vector processor. Also available to the node CPU for user applications code.
Fast Data Memory	16 KBytes (4K x 32), 100 nsec cycle time static RAM Dual ported, can be directly addressed by the VP's dedicated 32-bit Address ALU, or through the iLBX II bus by the 80286 node CPU (on a cycle stealing basis). Used as high-speed data storage for table functions and intermediate computational results. 4 KBytes are reserved for a user scratch pad.
Data Types	IEEE P754 floating-point (except gradual underflow) Single precision real: 32-bit Double precision real: 64-bit Single precision complex: 2*32-bit Double precision complex: 2*64-bit Integer/Logical: Supports 32-bit format of the 80286
iLBX II Port	Provides 80286 node CPU with 1 wait state memory mapped access to VP program and data RAM. Supports 8-, 16- and 32-bit accesses.
Indicators	System directed or user programmable red and green LEDs are viewable from the board's face plate or from the system's front panel display.
Size	2 x 4 Eurocard (9.2″ by 11″) with two 96-pin male DIN connectors. Includes data memory daughter board.

Physical	16-Node Cube	Cube Manager	Terminal	Keyboard
Height (in/cm)	49.0/124.5	6.5/16.5	12.0/30.5	2.25/5.7
Width (in/cm)	16.0/40.6	17.0/43.2	12.3/31.2	17.25/43.8
Depth (in/cm)	16.0/40.6	20.0/50.8	13.0/33.0	7.6/19.3
Weight (lbs/kg)	205.0/93.2	40.0/18.2	12.0/5.5	2.0/0.9
Footprint				
(in)	26.75 x 26.75	17.0 x 20.0	12.0 x 12.3	17.25 x 7.6
(cm)	67.9 x 67.9	43.2 x 50.8	30.5 x 31.2	43.8 x 19.3

Electrical	16-Node Cube	Cube Manager	Terminal/Keyboard
AC Voltage	230 VAC ± 15%	115/230 VAC ± 10%	115/230 VAC ± 10%
AC Current	14.4 amps	5.8/2.9 amps	2/1 amps
Frequency	50/60 Hz ± 5%	50/60 Hz ± 5%	50/60 Hz ± 5%
Power	3000 watts 10236 btu/hr	367 watts 1320 btu/hr	45 watts 154 btu/hr
Safety/RFI/EMI (designed to meet)	UL 478 CSA C22.2 No. 154 VDE 0806 VDE 0871 IEC 380 FCC 47 CFRJ Class A	UL 114 CSA 22.2 VDE 0871 IEC 435 FCC Docket 20780	UL

Environmental	16-Node Cube	Cube Manager	Terminal/Keyboard
Operating Temperature	10°-30° C	10°-35° C	0°-50° C
Humidity	85% max non-condensing	20-80%	10-90%
Altitude			
(feet)	0-10,000	0-8000	0-15,000
(meters)	0-3048	0-2438	0-4572
Acoustical	50 dBA, max		

SOFTWARE TOOLS SUMMARY

Professional FORTRAN

Mainframe-quality FORTRAN compiler from Ryan McFarland offers complete GSA certified implementation of FORTRAN-77 standard plus popular VAX and VS extensions. Supports arrays extending to the full 1 MByte capacity of vector memory and produces object code optimized for very fast execution.

FORTRAN Vectorizer

VXP translates standard FORTRAN source code into directives which control the vector processor. Capabilities include:

- Explicit use of the VecLib math function library
- Automatic Do Loop vectorization
- Automatic processing of scalar assignments
- Command subroutine chaining of complete program segments

Math Function Library

The math function library (VecLib) consists of over 100 microprogrammed vector, scalar, and logical routines. Each of these are available to the user as a FORTRAN subroutine call.

Simulator Debug Tool

The iPSC simulator is available on both the iPSC Cube Manager and Berkley 4.2 Unix systems. It provides an iPSC-VX node execution environment in which controlled program debugging can take place prior to execution on the actual Cube.

Performance Monitoring Display

Convenient front panel monitor can be used to display processing, communicating, and computational activities for each node. Provides instant feedback on system runtime behavior, load balancing, and fault diagnosis.

Microcode Development Tool Kit

Includes micro-assembler, debugger, and linker to support user microcode development.

Support Software

Provides software necessary to initialize, diagnose, load, and drive the vector processor from its adjacent node processor board.

iPSC USER TRAINING

Product training for one customer staff member is included with the purchase of each iPSC-VX system. The complete course sequence consists of five segments spanning a two-week period. Courses in week 1 teach all the basic skills required to use the iPSC-VX. The second advanced block of courses may be taken either immediately or at a later date and focuses on advanced concurrent programming topics and on microcode development (see Figure 5).

Each course of study consists of lectures, guest lectures, examples, case studies, hands-on programming experience, lab assistance, and class discussions.

Training is typically held at Intel Scientific Computers for two consecutive weeks each month and can be provided to additional staff members for a fee. Please phone the training administrator at Intel Scientific Computers at (503) 629-7629 to arrange training dates.

Course Modules

Title	Objectives
Introduction to the iPSC Concurrent Computer (one day)	Familiarize students with the architecture and functionality of Intel's iPSC family of concurrent computers.
Programming Concurrent Computers (two days)	Provide students with the fundamental knowledge, strategies, skills, and tools needed to design and implement software for Intel's iPSC family of concurrent computers.
Vector Concurrent Programming (two days)	Teach students how to use the vector processing capabilities of the iPSC-VX. Students are instructed on the use of the FORTRAN Vector Preprocessor, and the VecLib math function library.
Advanced Concurrent Programming (two days)	Familiarize students with advanced concurrent programming techniques useful to solving large scientific and engineering problems.
Microcode Development for the iPSC-VX (three days)	Provide students with the skills to develop proprietary microcode for the iPSC-VX vector processor boards.

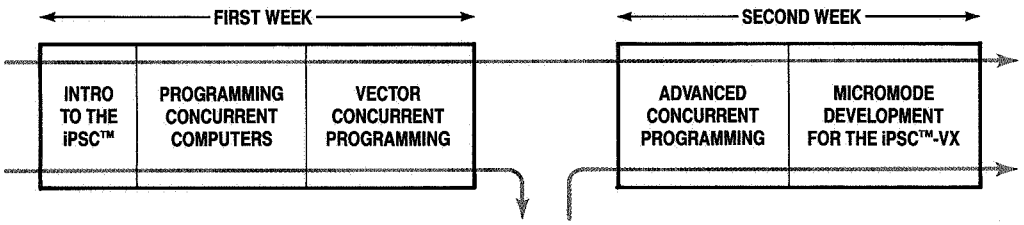


Figure 5—Course Sequence

iPSC DOCUMENTATION

The iPSC-VX Documentation Set is composed of the following reference manuals. Each one is also available separately under the order numbers shown.

Title	Order Number
iPSC-VX User's Guide	310238-001
iPSC-VX VecLib User's Manual	310239-001
iPSC-VX Diagnostics Manual	310240-001
iPSC Simulator Manual	310104-001
iPSC Dynamic Loader Manual	310103-001
iPSC Site Preparation Guide	280112-002

WARRANTY and SERVICE

Includes software support and on-site hardware maintenance services. For details see iPSC System Product Summary

ORDERING INFORMATION

iPSC-VX System Configurations*

Product Code	Description
iPSC/D4VX	iPSC-VX/d4: A complete 16-node vector concurrent system
iPSC/D5VX	iPSC-VX/d5: A complete 32-node vector concurrent system
iPSC/D6VX	iPSC-VX/d6: A complete 64-node vector concurrent system

*Each system includes system software and languages, a 1-, 2-, or 4-computational unit hypercube, a Cube Manager, an alphanumeric terminal, transceivers, and necessary cables.

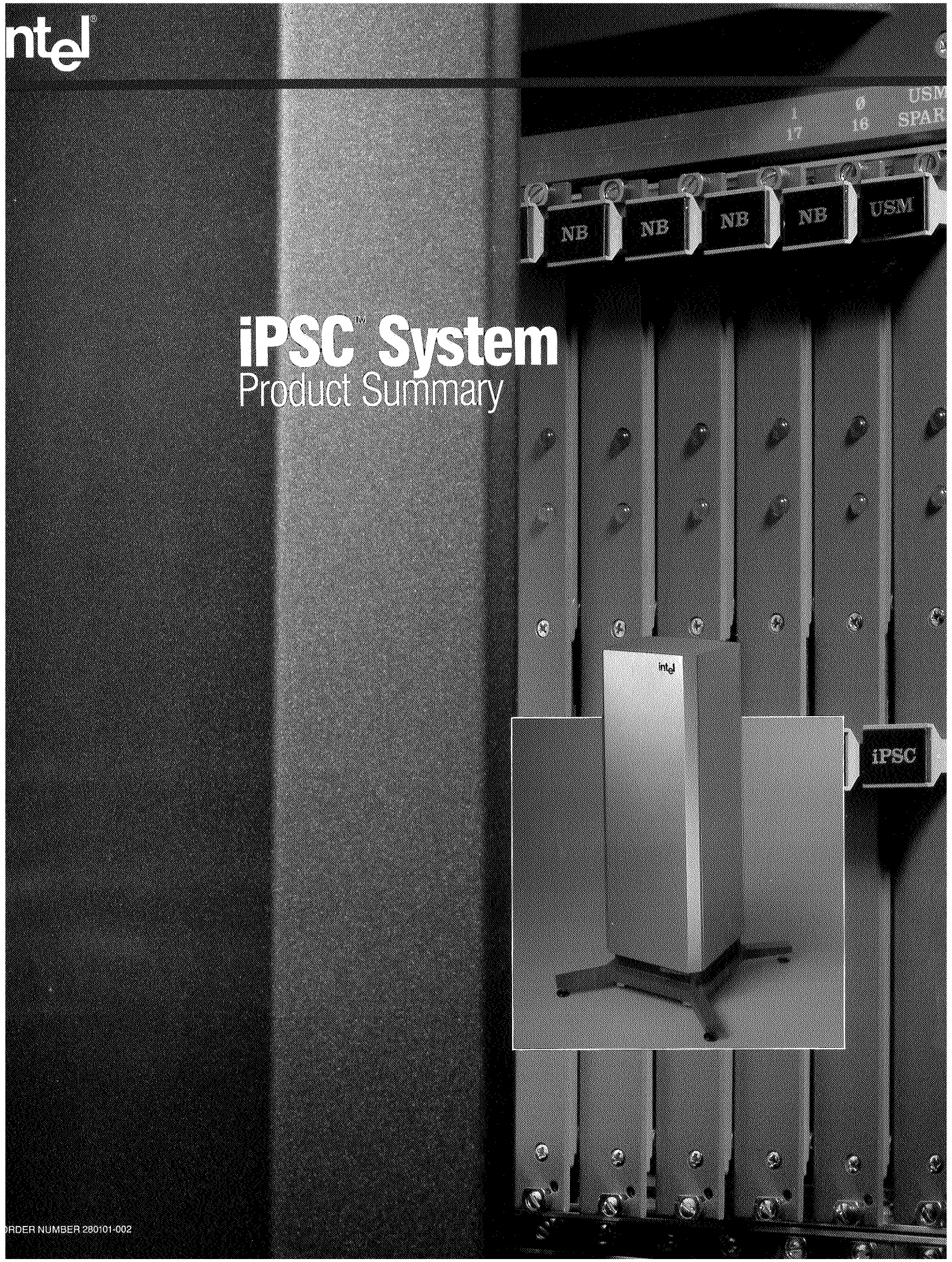
iPSC to iPSC-VX Upgrades*

Product Code	Description
UD5/D4VX	iPSC/d5 to iPSC-VX/d4 conversion
UD5/D5VX	iPSC/d5 to iPSC-VX/d5 conversion
UD6/D6VX	iPSC /d6 to iPSC-VX/d6 conversion
UD7/2D6VX	Conversion of one iPSC/d7 to two iPSC-VX/d6's

*Each upgrade option includes a complete iPSC-VX software upgrade plus additional vector boards, cabinets and cabling as required.

INTEL SCIENTIFIC COMPUTERS
15201 N.W. Greenbrier Parkway
Beaverton, Oregon 97006
(503) 629-7629

Printed in USA/SC6016/0486/2K/LC/VJC
SCIENTIFIC COMPUTERS



iPSC™ System
Product Summary

ORDER NUMBER 280101-002

iPSC

The iPSC™ is the first family of expandable concurrent computers. It is designed to be a flexible base of hardware and software upon which to build concurrent programming tools and application programs.

The iPSC system consists of one, two, or four computational units. Each unit contains up to 32 high-performance microcomputers, each with its own numeric processing unit and local memory. The microcomputers are interconnected in a hypercube topology. A resident node operating system, in conjunction with communication coprocessors, provides the user application with process-to-process message delivery capabilities.

The iPSC also includes Intel's MULTIBUS®-based System 286/310 microcomputer which is connected to each node by an Ethernet communication channel. The System 310 serves as Cube Manager, providing the user interface to the Cube as well as hosting the programming tools and system diagnostics.

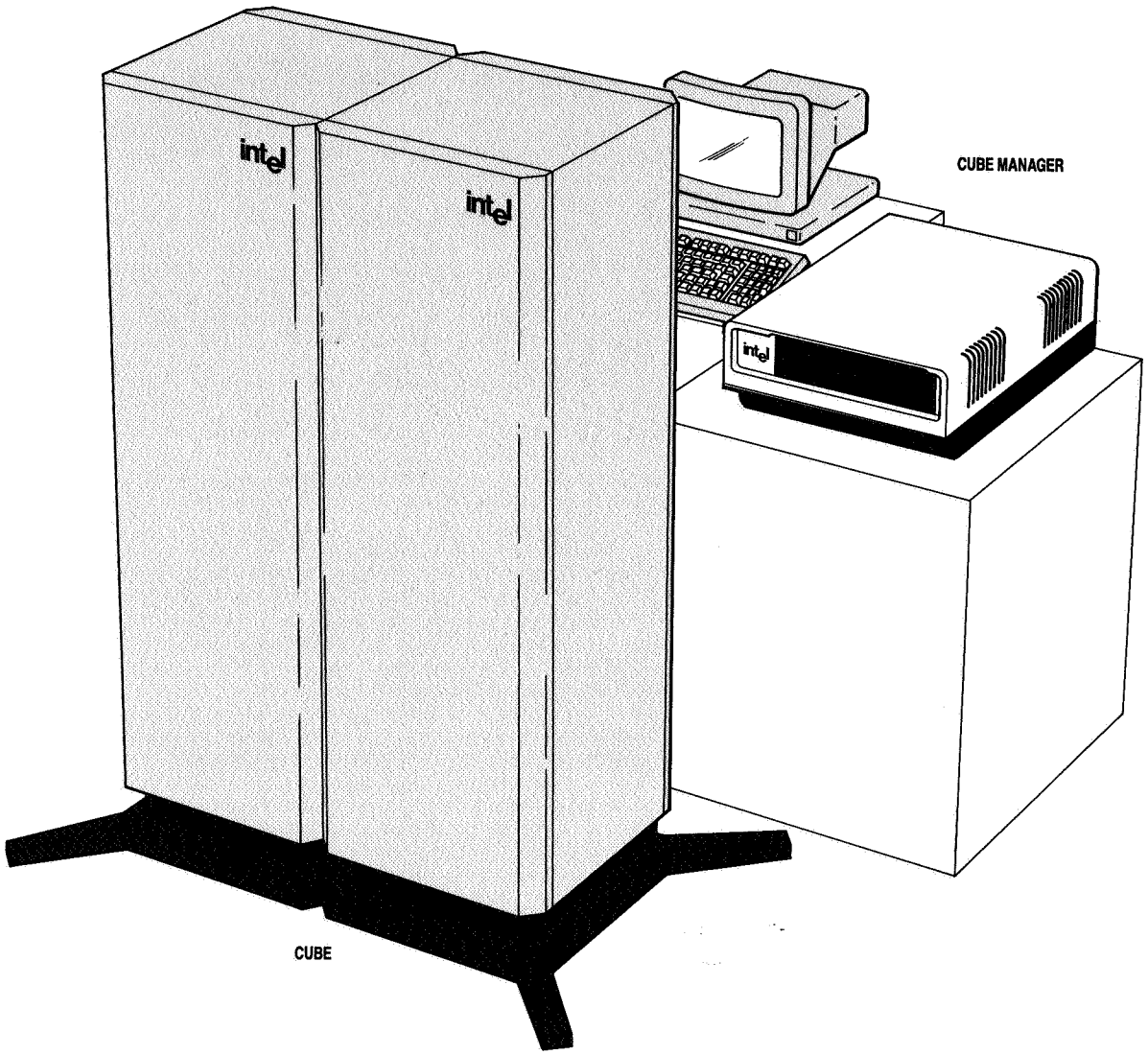


Figure 1-iPSC

FEATURES

Cube

Concurrent Architecture	Multiple, high-performance microcomputers work simultaneously, but independently on parts of a larger problem.
Versatile Hypercube Interconnect Topology	Efficient communication topology which is easily adaptable to a wide range of applications.
Reliable High-Speed Node-to-Node Communication	Bandwidth of 10 Mbit/second point-to-point internode serial communication channels with reliable message delivery between nearest neighbors.
Protected Software Environment	Intel 80286 central processing unit provides hardware memory protection between user programs and operating system.
High-Performance Numeric Computation	Intel 80287 numeric processing unit on each node. Supports 32-, 64-, and 80-bit floating point arithmetic operations (IEEE 754).
Large High-Speed Local Memory	512 KBytes of RAM per node, expandable to 4.5 MBytes.
Efficient Node Operating System	Multiple processes per node with cube-wide process-to-process message delivery; dynamic loading, execution, and termination of processes.
iLBX II Node Expansion Capability	Provides the facilities to extend memory or processing capacity with the addition of enhancement boards in adjacent node slots.
Field Expandable Computational Units	Base 32-node Cube is field upgradable to 64 or 128-node systems; expandable to 4.5 MBytes per node.
Low Mean-Time-To-Repair	A spare node board in each 32-node computational unit reduces mean-time-to-repair.
Office/Lab Environmental Design	Compact packaging, low acoustical noise, and an air-cooled design allow convenient placement.

Cube Manager

Supermicro System	Intel's powerful System 286/310 microcomputer with 80286/80287 and 2 MBytes of RAM.
Integrated Mass Storage	140 MByte Winchester disk, 360 KByte floppy disk, and 45 MByte cartridge tape drive integrated into one compact unit.
UNIX-Based Programming Environment	XENIX 3.0 operating system plus FORTRAN, C, and assembler languages.
High-Speed Manager-to-Cube Communication Channel	10 Mbit/second Ethernet channel bandwidth provides rapid direct access to every node with reliable message delivery.
Powerful Diagnostic Tools	Comprehensive confidence and diagnostic tests ensure system integrity and rapid fault isolation.
Flexible MULTIBUS Expansion	3 additional slots are available for system expansion from a choice of over 1500 MULTIBUS boards: 1 slot for high-speed memory expansion, 2 slots for other functions.
TCP/IP Network Access	Optional Ethernet, TCP/IP interface enables access to remote network resources.
Expandable Memory	Optional addition of DRAM memory board for a total of 5 MBytes of main memory.
Expandable Terminal Ports	Optional 8-port serial expansion board allows Cube Manager to support a total of 9 terminals.

SYSTEM OVERVIEW

The iPSC provides users with an expandable concurrent processing system for computationally intensive scientific applications. It consists of multiple high-performance microcomputers that each work concurrently on parts of a larger problem. This architecture enables users to exploit the inherent concurrency present in many scientific problems, and obtain high computational performance at the relatively low costs made possible by VLSI technology.

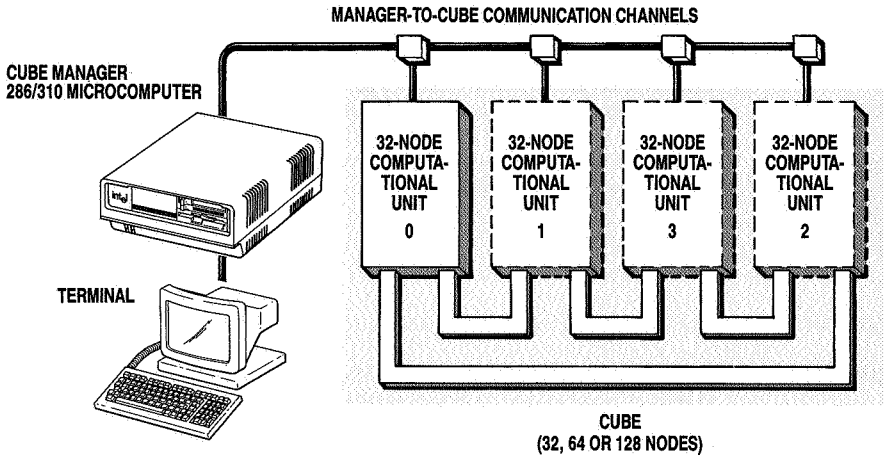


Figure 2-iPSC System Architecture

The iPSC consists of two major functional elements: the ***Cube*** and the ***Cube Manager.***

Cube

Cube hardware consists of an ensemble of one, two, or four computational units consisting of 32, 64, or 128 microcomputers or nodes. Nodes contain high-performance Intel 80286 CPU's coupled with 80287 numeric processing units and 512 KBytes of local memory. Nodes are interconnected by high-speed communication channels in a hypercube topology.

Cube software consists of a monitor and kernel residing on each node board. The monitor is in PROM and the kernel is loaded into node RAM after successful initialization. The kernel provides user application processes with a cube-wide process-to-process message passing service.

Cube Manager

Cube Manager hardware consists of an Intel System 286/310 microcomputer which is linked to each node over a global Ethernet communication channel.

Users interact with the System 310 via an ANSI-compatible alphanumeric terminal. MULTIBUS-based boards for the System 310 provide an optional Ethernet, TCP/IP LAN capability which enables networking to remote host environments.

Cube Manager software consists of a UNIX-based programming and development environment with FORTRAN, C, Assembler, cube control utilities and communications, and complete system diagnostics.

CUBE DESCRIPTION

Cube Hardware

The term Cube refers to the ensemble of microcomputers (nodes) connected in a concurrent hypercube architecture. The Cube may consist of one, two, or four 32-node computational units. The major hardware elements are described below:

Topology

The interconnection scheme, or topology, for the iPSC™ is a ***"binary n-cube"*** or ***"hypercube."*** This is an *n*-dimensional cube where *n* represents the number of directly-connected nodes that establish the cube's dimension. The "dimension" is equal to the power of two, corresponding to the number of nodes in the cube. A 32-node system is a 5-dimensional cube with each node connected to its five nearest neighbors (i.e., d5). A 64-node system is a 6-dimensional cube and a 128-node system is a 7-dimensional cube (i.e., d6 and d7 respectively). Examples of this topology are shown in Figure 3. For simplicity, this diagram also shows a 4-dimensional (16-node) and 5-dimensional (32-node) hypercube.

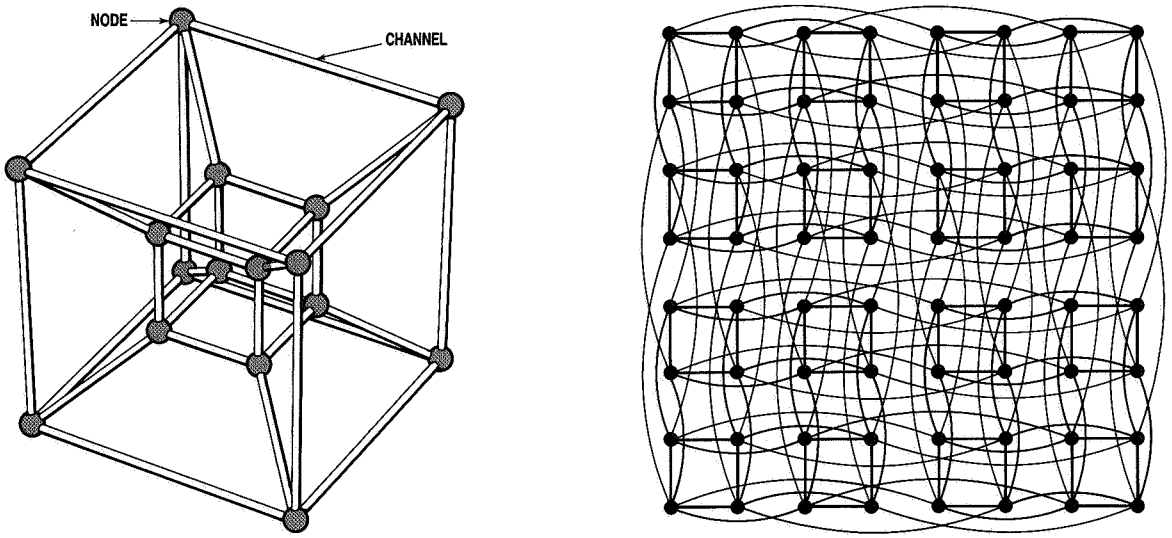


Figure 3–Hypercube Topology

The various iPSC models are designated according to the dimension of the associated hypercube.

The available models are:

iPSC-d5	5-dimensional cube	(2^5 = 32 nodes)
iPSC-d6	6-dimensional cube	(2^6 = 64 nodes)
iPSC-d7	7-dimensional cube	(2^7 = 128 nodes)

Memory is local to each node, and interprocess communication is accomplished by message passing, eliminating the bus contention problems associated with shared memory systems.

The hypercube provides a nearly ideal balance between performance and communication overhead. Any node in the cube can be reached in a small number of message transmissions. Internode delays increase proportionally to the dimension of the cube. For example, in the 32-node system the maximum internode delay is 5.0 node-to-node transmission periods with an average delay of only 2.5, assuming uniformly distributed message distances. Typical application delays are much shorter.

Because of the versatility of the architecture, other topologies...such as rings, trees, etc.... can be mapped onto the system because the hypercube is a superset of these other topologies. Specifically, these topologies can be realized by appropriate programming. This multi-dimensional structure also makes the hypercube well suited to both homogeneous and heterogeneous computational problems.

The hypercube interconnection is physically implemented via the backplane within each computational unit. Cables running between units implement the interconnect for d6 and larger dimension cubes (see Figure 2).

Nodes

Each node in the Cube is an independent, single-board computer. The node contains a high-performance Intel 80286 central processing unit and its companion 80287 numeric processing unit which support 32-, 64-, and 80-bit floating-point formats. The node also contains 512 KBytes of dynamic RAM, with byte parity. An initialization and self-test monitor is contained in 64 KBytes of PROM (see Figure 4).

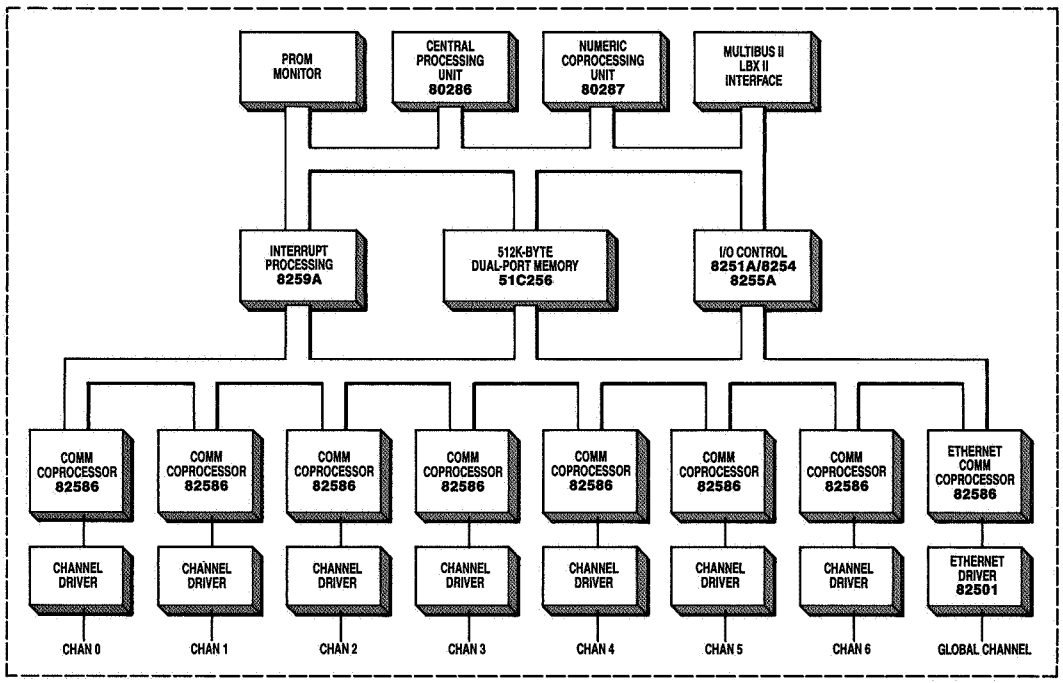


Figure 4-Computational Node

Each node also contains eight bidirectional communication channels managed by dedicated 82586 communication coprocessors. Seven of these channels physically link the nodes together and serve as dedicated point-to-point communication channels. The eighth channel is a global Ethernet channel that provides direct access to and from the Cube Manager for program loading, data input/output, and diagnostics.

Each node board has red and green LED indicators which are used during initialization and reset for diagnostic purposes. The LED indicators may also be controlled by the user's node program during execution for monitoring and debugging.

Node Expansion Port

For enhancement purposes, the local processor memory bus on each node is accessible via a standard MULTIBUS II iLBX™ high-speed bus interface. The iLBX-II interface reaches the backplane via one of the two 96-pin DIN connectors. On the backplane, the iLBX-II bus is routed from each even-numbered board slot to the adjacent odd-numbered board slot. The odd-numbered slots may be populated by boards that extend the memory or processing capacity of the nodes. The node memory option expands each node's memory through use of the iLBX-II bus.

Manager-to-Cube Communication Channel

The Manager-to-Cube communication channels are shown in Figure 5. There are two channels: global and diagnostics.

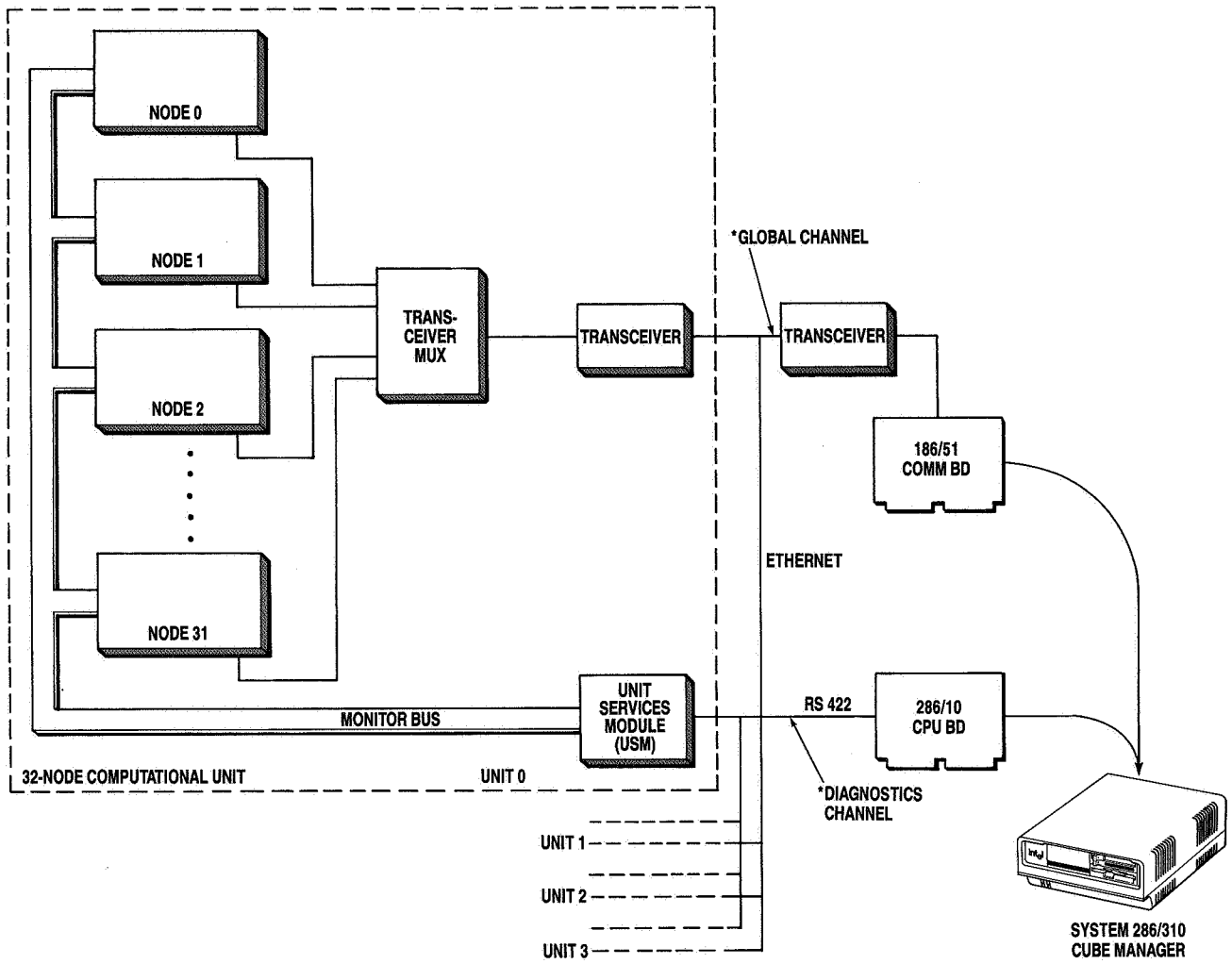


Figure 5–Manager-to-Cube Communication Channels

Channels

The global channel enables Cube Manager processes to communicate with the node processes. This channel physically links the Cube Manager to each node from the iSBC 186/51 communications board to the global communications coprocessor chip set on each node board. This channel complies with the standard Ethernet specification (IEEE 802.3).

The diagnostics channel is a separate path for communicating with the nodes. This channel links the Cube Manager to the unit services module (USM) through an RS 422 port. The USM manages the multidrop monitor bus supplying all nodes in a single unit with communication, interrupt, and reset lines. In the event of a failure, this alternate path is used to determine if the fault is within a node or within the global communication channel to the nodes. It is used to reset the nodes, initiate on-board diagnostics, and monitor the results, thus providing a more fundamental level of communication than is possible over the global channel.

Cube Enclosure

Each 32-node computational unit is a free-standing enclosure which is 49″ high and 16″ square with a 26.75″ x 26.75″ footprint. Each enclosure contains 32-node boards, one spare node board, and the USM board in one 34-slot card cage. Communications transceivers, multiplexer, power supplies, air cooling system, and associated cables are also housed in the enclosure (see Figure 6).

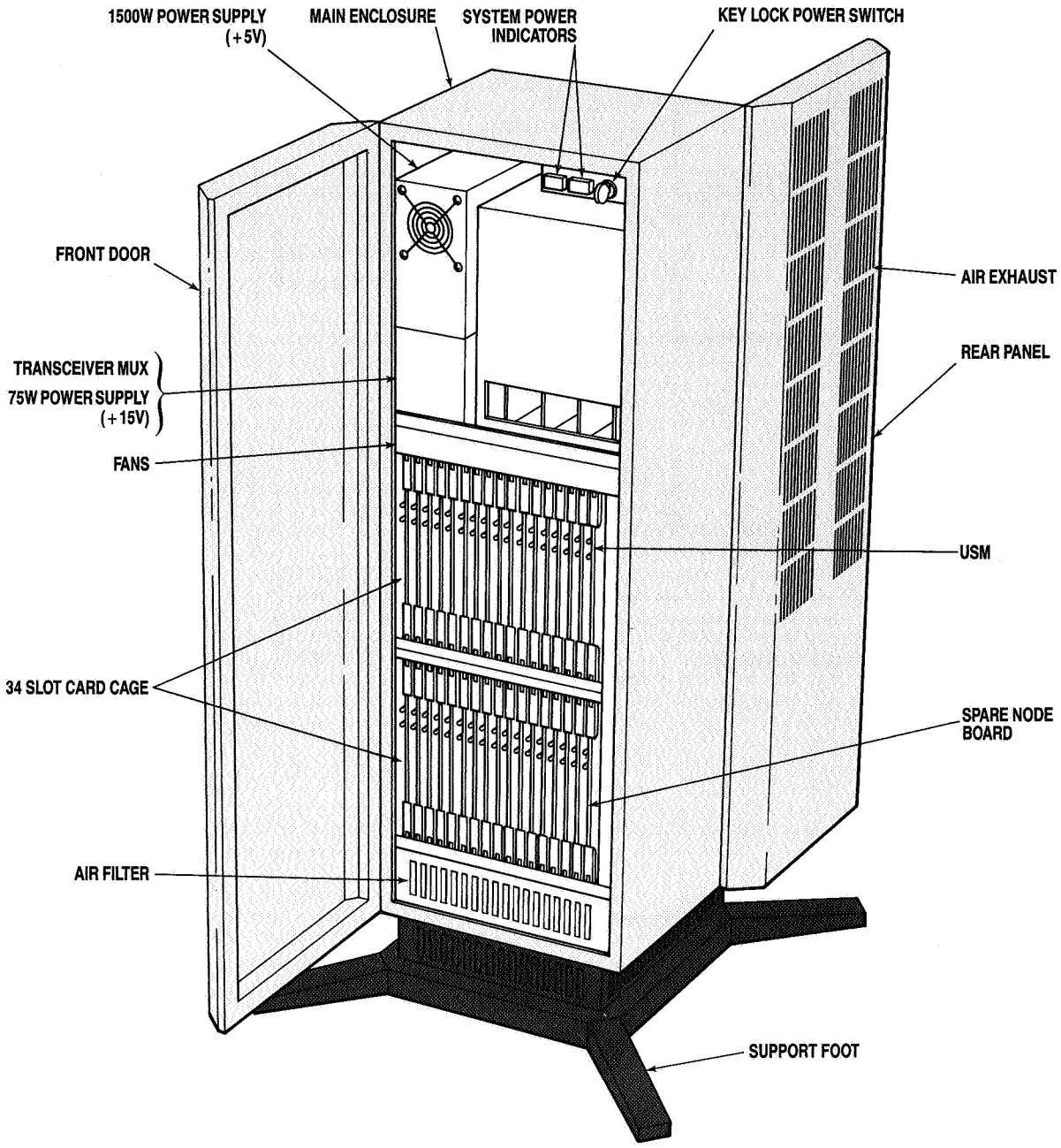


Figure 6–Cube Enclosure

Power Distribution System

A removable key, in the switch on the front panel, turns on the cooling fans and +15-volt, and +5-volt power. Front-face lamps indicate AC on and DC on in that sequence. A power module at the base of the unit houses a filter and circuit breaker which can be externally reset from the rear. An automatic thermal shutdown circuit in the +5-volt power supply protects the system from catastrophic heat failure (see Figure 7).

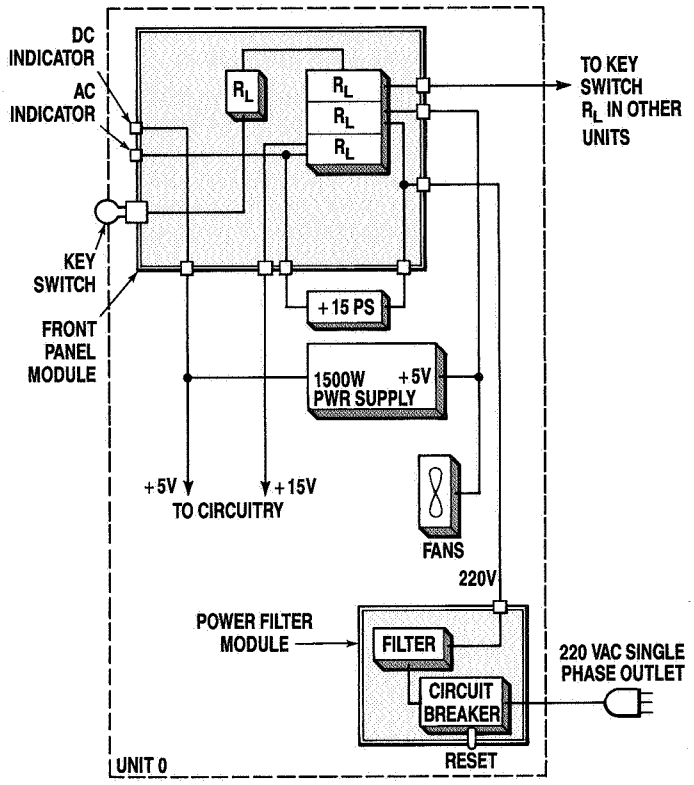


Figure 7–Power Distribution System

In a multi-unit system, Unit 0 enables the power-up sequence for each of the other units.

Cube Software

Cube software consists of a monitor and a kernel operating system residing on each node.

Node Monitor

The node monitor is responsible for initializing each node at power-up or system reset, verifying node operability, and loading the node operating system. It is stored in PROM on each node board.

Initialization

Initializes the system by resetting and enabling the node memory, communication controllers, I/O controller, interrupt controller, and CPU. Node identity is also set by reading the slot ID from the backplane.

Confidence Testing

Verifies the node board by running the node confidence test (*"NCT"*) on the RAM, peripheral devices, and the communications controllers.

Static Loading

Loads the node operating system and, optionally, a statically bound application program.

Node Operating System

The node operating system provides the application programmer with the necessary set of software services for dynamically loading programs, managing multiple processes, and delivering variable length messages between processes, all within a protected, large-model, multiple address space environment.

Dynamic Loading

Allows multiple application programs to be loaded into node memory, placed in execution, and subsequently stopped or killed, all under user program control. Eliminates the need to reload the node operating system with each application.

Interprocess Communication

The operating system provides users with a flexible set of capabilities that help optimize communication in a concurrent processing environment. The communication interface is consistent whether communicating with other processes in the same node, to remote nodes, or to processes in the Cube Manager. Sending and receiving can be synchronous or asynchronous. Messages are automatically routed from node to node, if necessary, to reach the destination process (see Figure 8).

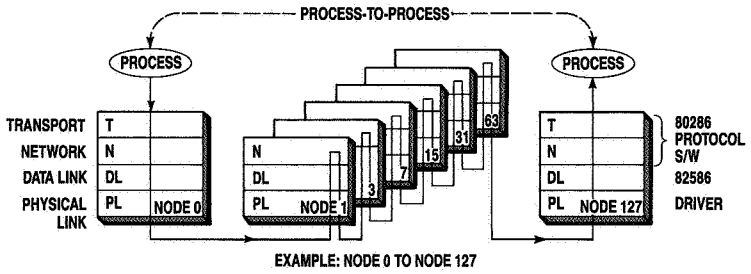


Figure 8–Automatic Process-to-Process Message Routing

Reliable message delivery service is provided between nearest node neighbors (see Figure 9). Data message length can vary from 0 to 16 KBytes. Messages larger than 1 KByte are automatically fragmented and reassembled at the destination node, transparently to the user process.

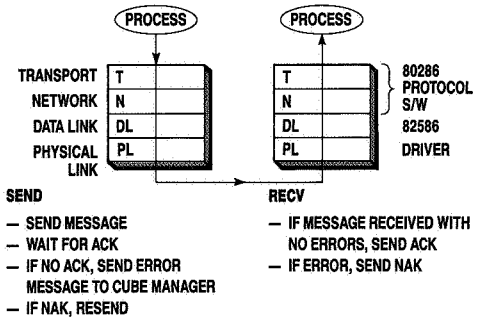


Figure 9–Nearest Neighbor Communication Protocol

	To send or receive messages, the user, via FORTRAN or C interface libraries (refer to Table 1), simply opens a channel. To send a message, the user invokes the SEND library routine with the appropriate message parameters. To receive a message, the user calls RECV with the appropriate parameters.
Process Management	Controls process scheduling by allowing each process to run for a specified interval in a round-robin fashion. The number of processes supported is limited only by available memory.
Physical Memory Management	Provides memory space for each process on the node as well as message buffering. Memory management may be controlled by the user, if desired.
Protected Address Spaces	Local descriptor tables (one for each process) provide a "shield" between the operating system and user space and between multiple user process spaces. This is hardware enforced so that the user cannot corrupt the operating system software.

Table 1. Node Operating System Services

Library Name	Function Performed
COPEN	Creates a channel for node process communication. A descriptor block for message parameters is set up in the operating system and a channel identifier (CI) is assigned and returned to the user process.
CCLOSE	Destroys the communication channel created by the COPEN call.
SEND	Initiates transmission of a message to another process. The return to the calling process acknowledges the SEND request. STATUS must be used to determine if the user message descriptor is available for reuse.
SENDW	Initiates transmission of a message to another process. The return to the calling process indicates that the user message descriptor is available for reuse.
RECV	Initiates the receipt of a message. The return to the calling process acknowledges receipt of the request by the operating system.
RECVW	Initiates the receipt of a message. The calling process is blocked until the message has been received.
STATUS	Informs the calling process as to the availability of the user message descriptor for another call.
PROBE	Provides the calling process with the ability to determine if messages of a specified type have been received on a channel prior to invoking a RECV call. It returns the length of the received message to the calling process if a message has been received.
FLICK	Relinquishes the CPU to other processes prior to completion of the scheduled time interval.
MYNODE	Returns the node number of the process that initiated the call.
CUBEDIM	Returns the dimension of the cube to the calling process. (For example, "6" for a 64-node cube.)
CLOCK	Returns a snapshot value of elapsed time, in seconds, that started at initialization of the node. Maximum period is approximately 49 days.
GREENLED	Allows process to turn on or off the node's green LED.
REDLED	Allows process to turn on or off the node's red LED.
MYPID	Returns the process ID of the calling process.

Optional Cube Software

Source Code

The sources for iPSC node communication libraries, dynamic loader, node operating system, and XENIX communication driver may be purchased. A PL/M compiler is provided with the source code to facilitate additions and changes. The product is distributed on 5.25″ floppy diskette, cartridge tape and on half-inch 9-track tape.

CUBE MANAGER DESCRIPTION

Cube Manager Hardware

The Cube Manager hardware consists of a microcomputer system, related options, and an alphanumeric terminal.

Intel System 310 Microcomputer

The System 310 is a MUTLIBUS-based microcomputer built with Intel's 80286 CPU and 80287 numeric processing unit (see Figure 10). It contains a 5¼" 140 MByte Winchester disk, a 360 KByte floppy disk, and 2 MByte ECC RAM memory. The 310 also has an integrated Ethernet interface for communicating with the Cube.

Internally, the 310 has a 7-slot card cage powered by a 360-watt power supply with the following MULTIBUS boards integrated into four slots:

iSBC 286/12 CPU board
iSBC 010CEX iLBX memory board
iSBC 214 disk controller board
iSBC 186/51 communications board

The communication board is connected to an external Ethernet transceiver which is cabled to the cube. An RS 422 link runs from the CPU board to the cube to provide the diagnostic channel.

Expansion Options

Printers and other peripherals can easily be added to the Cube Manager via a Centronics compatible printer port and three spare MULTIBUS expansion slots. One slot is reserved for iLBX boards and the other two are available for standard MULTIBUS boards (see Figure 11).

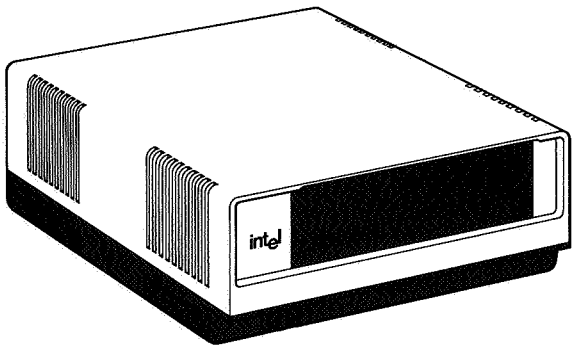


Figure 10-System 286/310 Microcomputer

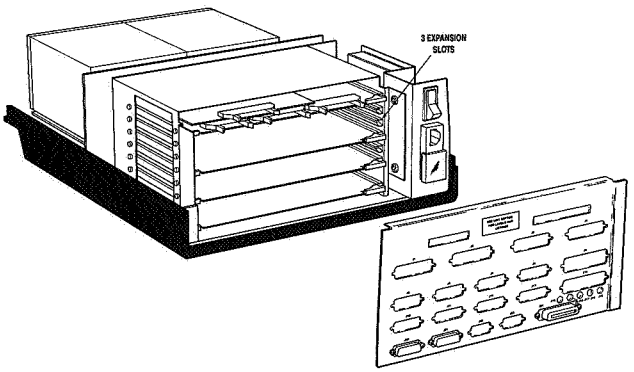


Figure 11-System 310 Expansion Capability

Expansion options include:

Ethernet, TCP/IP network option–Enables access to remote host environment resources such as mass storage, workstations, or printers. It consists of an Ethernet front-end processor and associated software. Provides protocol package conforming to latest DOD ARPANET TCP/IP specifications. Compatible with UC Berkeley Version 4.2.

4 MByte memory expansion option–Expands memory capacity to 5 MBytes. It consists of a 2048 KByte memory board with dual port capability (MULTIBUS and iLBX interface). Includes double-bit error detection and single-bit error correction logic. An error status register provides error logging to the host CPU board.

Serial port option–Provides eight additional serial ports. It is an intelligent communications controller which functions as a single board controller or as an intelligent slave for multi-terminal communications expansion. The on-board iAPX 188 CPU provides communications control and buffer management for up to eight programmable synchronous/asynchronous channels. The board includes 64 KBytes of dual-ported parity RAM buffer space to handle messages at data rates up to 19.2 K Baud.

Over 1500 other board options–available from Intel and other MULTIBUS vendors.

Note: For more details, refer to Reference Literature section.

Terminal

The terminal is an alphanumeric terminal which meets ANSI X3.64 specifications (VT100 software compatible). It features high resolution characters (upper/lower case) on a 14″ non-glare green phosphor tilt display with detachable keyboard. The 101 keys include a numeric pad and function keys in addition to a standard typewriter keypad (see Figure 12).

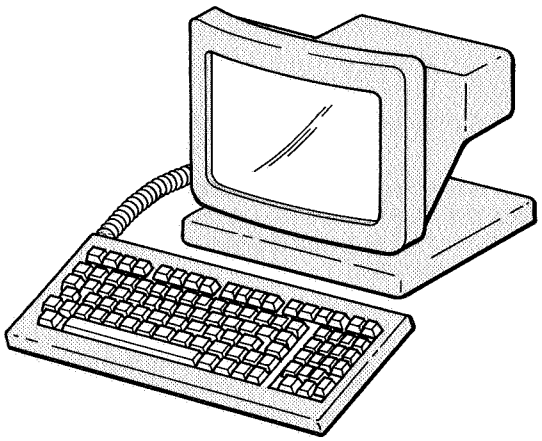


Figure 12–Cube Manager Terminal

Optional Development Station

iPSC Development Station

Providing an environment identical to the Cube Manager, the iPSC Development Station offers additional access to program development services. The station provides simulators, compilers, and libraries for creating executable load modules. A TCP/IP network interface is also provided for easily copying code modules to or from remote Cube Managers.

Cube Manager/Development Station Software

The Cube Manager software is divided into three basic categories: programming and development software, cube control software, and diagnostics.

Programming & Development Software

Operating System

XENIX 3.0 a fully-licensed derivative of UNIX System III. It includes enhancements from University of California at Berkeley as well as Microsoft and Intel. It is compatible with UNIX Version 7.

XENIX 3.0 includes:

UNIX System III–kernel, utilities, libraries, tools.

Berkeley Enhancements–curses, termcaps, C shell, VI editor, various utilities.

Microsoft Enhancements–large model C compiler, UNIX version 7 compatibility, visual shell, MS DOS file transfer features, shared data, file locking, keyed access to files.

Intel Enhancements–performance tuning to the system 286/310, device drivers for: 4- and 8-channel intelligent I/O controllers, Ethernet controller, Winchester, floppy, and tape controllers, RAM disk, console, and line printer.

FORTRAN Compiler

Meets ANSI FORTRAN 77 subset language specifications and supports real math IEEE floating-point standard, transcendentals, complex numbers, interrupt procedures, and run-time exception handling.

This is a large-model FORTRAN compiler and handles very large arrays.

Macro Assembler

Native 286 Assembler ("ASM 286") plus macro extension.

Tools

Linker
Loader

Simulator

The iPSC Simulator is included with the purchase of each iPSC System, but also is available as a separate product.

The iPSC Simulator is a software program which simulates the actions of the Intel iPSC system. Use of the simulator significantly speeds up the program developent cycle and provides facilities for locating program errors. Large cube programs will run much more slowly on the Simulator than the actual Cube, however.

The Simulator package provides documentation, simulator program, and a set of libraries. Documentation consists of installation instructions, Users Manual, Internal Design Specification, and iPSC System Overview. The package is distributed in its source code form on both half-inch 9-track tape and 5.25″ floppy and is easily installed on Cube Managers, Development Stations, and Unix BSD 4.2 or Xenix 3.0 machines. A Simulator Source Code License is required.

Note: For more details, refer to Reference Literature section.

Cube Control Software

Node Facilities

Support multiple application processes and provides reliable message delivery services to any node in the cube via the shortest route. Consistent with the Cube Manager interface for sending and receiving messages. (Refer to Table 2 for a list of FORTRAN and C services.)

Cube Manager

During initialization, each node is queried for node confidence test ("NCT") status. Control is deferred to user diagnostic procedures if an error is indicated.

After node initialization, several utilities exist (refer to Table 3) to load, start, kill, or investigate processes on the nodes. These utilities are available as a collection of XENIX programs which may be invoked from the terminal using standard XENIX shells, or executed by XENIX programs.

Diagnostics

The iPSC diagnostics consist of a set of software confidence and diagnostic tests.

The confidence tests are used to verify overall system integrity prior to normal usage. The diagnostic tests are targeted to individual boards or system modules and are used to isolate faults to this level.

System Confidence Testing

At power up, or upon reset, PROM-based confidence tests in the Cube Manager and Cube verify individual module integrity and basic system functionality. Immediately following, during the boot-up sequence, XENIX verifies the system configuration. If an error occurs during initialization confidence testing, the user can invoke system diagnostic tests as appropriate.

System Diagnostic Testing

Under user control, individual floppy-based diagnostic tests can be invoked to facilitate isolation of faults to the board/system module level. The diagnostics strategy is to verify the functionality of the Cube Manager first, Cube Manager-to-cube communications next, and then targets the Cube. Test clusters target such module elements as the central processing unit, numeric processing unit, RAM, I/O peripheral devices, Winchester, and floppy disk.

Table 2. Cube Manager Communication Services

Library Name	Function Performed
COPEN	Creates a channel for Cube Manager process communication. A descriptor for message parameters is set up and a channel identifier (C1) is assigned and returned to the user process.
CCLOSE	Destroys the communication channel created by the COPEN call.
SEND MSG	Initiates transmission of a message to another process. The return to the calling process indicates that the user descriptor is available for reuse.
RECVMSG	Initiates the receipt of a message from a designated process (node and process ID specified). The calling process is blocked until the message has been received.
CUBEDIM	Returns the dimension of the cube to the calling process.

Table 3. Cube Manager Commands

Command Name	Function Performed
LOAD	Loads a file into a node or group of nodes. Checks for sufficient memory, valid process ID, valid node ID, and valid file record.
LOADSTART	Starts execution of a file in a node or group of nodes. Checks for sufficient memory, valid process ID, valid node ID, and valid file record.
LKILL	Kills the process specified by process ID. Checks for sufficient memory, valid process ID, valid node ID, and valid file record.
LWAIT	Waits for a process (or processes) to complete on a node (or group of nodes). Checks for sufficient memory, valid process ID, valid node ID, valid file record, and active process for which to wait.
CUBELOG	Instructs the system to perform various manipulations on the system logfile; such as, emptying the contents or changing the logfile name.

SYSTEM SPECIFICATIONS

Cube	/d5	/d6	/d7
Number of Nodes	32	64	128
Number of Spare Nodes	1	2	4
Total RAM Memory	16 MBytes (expandable)	32 MBytes (expandable)	64 MBytes
Total Node-to-Node Communication Channels	80	192	448
Cube Footprint	26.75″ x 26.75″	42.7″ x 26.75″	74.5″ x 26.75″

Node

Central Processor (CPU)	Intel 80286 16 MBytes physical addressing Memory management and protection
Numeric Processor (NPU)	Intel 80287 32-, 64-, 80-bit floating point (IEEE 754) 32-, 64-bit integer 18-digit BCD operands
Memory	512 KBytes dual-port RAM with byte parity, expandable to 4.5 MBytes of RAM 64 KBytes PROM monitor (expandable)
Communication Channels	Eight total. Seven to nearest neighbor nodes via Intel's 82586 communication coprocessors and 1 Ethernet channel to Cube Manager via 82586/82501
Communication Bandwidth	H/W: 20 Mbit/sec max.
Control I/O	Reset, interrupt
iLBX II Port	Bus bandwidth: 8 MByte/sec max. 16 MByte addressing 8- and 16-bit data transfers over 32-bit path Even numbered slots are masters; odd are slaves
Indicators	Red LED Green LED
Size	2 x 4 Eurocard (9.2″ by 11″) with two 96-pin male DIN connectors

Cube Manager

Central Processor (CPU)	Intel 80286
Numeric Processor (NPU)	Intel 80287 32-, 64-, 80-bit floating point (IEEE 754) 32-, 64-bit integer 18-digit BCD operands
Memory	2 MByte iLBX memory with ECC expandable to 5 MBytes
Mass Storage	140 MByte 5¼″ Winchester disk 360 KByte 5¼″ floppy disk (DS,DD) 45 MByte cartridge tape drive
MULTIBUS Expansion Slots (IEEE 796)	3 available slots at 0.6″ spacing 1 reserved for iLBX memory, 2 for standard MULTIBUS 93 watts available power
Printer Port	Centronics compatible
User Processes Supported	50 (XENIX configuration parameter)
Global Channel	Ethernet (IEEE 802.3)

Terminal — WYSE 75

Compatibility	VT100, ANSI X 3.64
Keyboard	Detachable, 101 keys Typewriter, numeric, and function key pads 2-position tilt 16 programmable function keys (32 combinations)
Display	14″ swivel and tilt display Non-glare green phosphor 24 rows by 80 or 132 columns
Character Set	7x13 matrix in 10x13 cell 128 ASCII character set Upper/lower case with line drawing graphics Cursor block or underline selectable, with or without blinking

Physical

	32-Node Unit	Cube Manager	Terminal	Keyboard
Height	49″	6.5″	12″	2.25″
Width	16″	17″	12.3″	17.25″
Depth	16″	20″	13″	7.6″
Weight	200 lbs	40 lbs	12 lbs	2 lbs
Footprint	26.75″ × 26.75″	17″ × 20″	12″ × 12.3″	17.25″ × 7.6″

Electrical & Environmental

	32-Node Unit	Cube Manager	Terminal/Keyboard
ELECTRICAL			
AC Voltage	230 VAC ± 15%	115/230 VAC ± 10%	115/230 VAC ± 10%
AC Current	13.8 amps	5.8/2.9 amps	2/1 amps
Frequency	50/60 Hz ± 5%	50/60 Hz ± 5%	50/60 Hz ± 5%
Power	2704 watts 9226 btu/hr	367 watts 1320 btu/hr	45 watts 154 btu/hr
SAFETY/RFI/EMI	UL 478	UL 114	UL
(designed to meet)	CSA C22.2 No. 154	CSA 22.2	
	VDE 0806	FCC Docket 20780	
	VDE 0871		
	IEC 380	IEC 435	
	FCC 47 CFRJ Class A	VDE 0871	
ENVIRONMENTAL			
Operating Temp.	10-35°C	10-35°C	0-50°C
Humidity	85%, max. non-condensing	20-80%	10-90%
Altitude	0-10,000 ft	0-8000 ft	0-15,000 ft
Acoustical	50 dBA, max		

iPSC DOCUMENTATION

The iPSC Documentation Set is composed of the following reference manuals. Each one is also available separately under the order numbers shown.

iPSC Documents

Title	Order Number	Content
System Overview Manual	175278-001	Introduces the user to the iPSC system and explains the general concepts...stressing those associated with concurrent processing.
User's Guide	175279-001	Contains all of the material normally required to use the system. Includes both hardware and software information. Explains how to start up the system and develop and execute programs. Contains general housekeeping procedures as well as customer preventive and corrective maintenance. Also includes various applications program examples.
Dynamic Loader Manual	310103-001	Describes the facilities of the dynamic loader for programmers. Details the procedure for loading, starting, waiting on, and killing processes on the nodes
Simulator Manual	310104-001	Describes the facilities of the iPSC Simulator for programmers.
iPSC Diagnostics User's Guide	175306-001	Describes the diagnostic facilities available to the service engineer
iPSC Site Preparation Guide	280112-002	Provides information concerning space, power, cooling, and cabling for the iPSC system.

Reference Literature

Additional information is available from the following Intel documents:

Title	Order Number	Content
Systems Handbook	210941-003	MULTIBUS board and system product information.
Introduction to the System 310 Microcomputer	173202-002	System 310 and XENIX technical product overview/reference document.
FORTRAN 286 User's Guide	122196-000	Technical product information.

iPSC USER TRAINING

Training for one customer staff member is included with the purchase of each iPSC System. This 5-day course titled "Programming Concurrent Computers" provides students with the fundamental knowledge, strategies, skills, and tools needed to design and implement large-scale concurrent software for the iPSC.

The course is held approximately once a month at Intel Scientific Computers in Beaverton, Oregon. Study consists of lectures, guest lectures, examples and case studies, hands-on programming experience, lab assistance, and class discussions.

Training for other customer staff members is available for an additional fee. Please phone Intel Scientific Computers at (503) 629-7629 to arrange training dates and fees.

iPSC WARRANTY/SERVICE

Included in the purchase price of any base iPSC configuration are the following:

Pre-Installation Site Planning

An Intel service consultant will provide assistance regarding site and environmental requirements, including power and air conditioning needs. Intel will arrange an installation schedule that allows adequate time for site preparation and that insures a successful and timely installation. (Refer to Site Preparation Guide)

Installation

Intel field engineers will arrive on site on the designated installation date. The equipment will be assembled and system integrity verified. Basic system orientation training will also be provided.

Hardware Warranty

Intel warrants iPSC related products to conform to Intel-published specifications and to be free from defects in material and workmanship for a period of one year from date of shipment. During the warranty period, service is provided on site for the first 90 days and factory Return Receipt Authorization (RRA) is provided for the remaining nine months of the year.

Software Support

Intel provides software support for a period of one year from date of shipment. During the warranty period, service includes software updates, technical reports, and software problem reporting service. Software support also includes a HOTLINE telephone number for priority assistance. (Refer to Software Support information below for more details)

Software Support

Intel provides the following software support during the warranty period and under annual software support contract after warranty. Further details are described in iSC's Terms & Conditions of Services.

Technical Reports

Periodic publication containing solutions to known problems, advance notice of software updates, programming tips, iPSC training schedule, and listings of latest available manuals/documentation. Manual updates are automatically distributed as additions, changes, and clarifications are published.

Software Problem Reporting (SPR) Service

Intel will respond to written questions and/or problems (submitted on a standard SPR form) on system software or documentation. Intel will verify receipt of the SPR within 48 hours and will respond within 3 weeks in writing. Intel does not guarantee a resolution will always be available.

Technical Information Phone Service (TIPS)

Intel will provide telephone assistance to software related problems during normal working hours (8:00 a.m. to 5:00 p.m., Pacific Time, Monday through Friday).

Software Updates

Intel will provide, at no additional charge, software updates or new releases to existing software products. Updates include such items as problem fixes and performance improvements. Documentation is included as well as installation assistance. Individual software support items are available on an annual basis. Contact your iSC sales representative.

To be eligible for software support, customer's system software and hardware must be at the currently-supported revision level.

On-Site Hardware Maintenance Service

On-site service is available for hardware under an annual service agreement or on a per-call basis. Refer to iSC Service Agreement for complete details.

Hardware Maintenance Agreement

Intel offers a hardware maintenance agreement that allows budgeting of all hardware service costs, guarantees priority response, and insures that the hardware remains current.

Support Includes:

Unlimited service calls during contract hours
Unlimited problem determination and resolution on a priority basis
All replacement parts necessary are provided on an exchange basis
Installation of all necessary field change orders
Scheduled preventive maintenance

Per-Call Service

Field service is also available on a time and materials basis. Normal response is on a best-effort basis. Travel time, repair time, expenses, and parts are billed on an as-used basis:

Hourly rates:

8:00 a.m. to 5:00 p.m., Monday through Friday—$100/hr

Minimum charges:

2 hrs within a service zone
3 hrs outside a service zone
Parts charges are at prevailing rates

ORDERING INFORMATION

Base Configurations*

Part Number	Description
iPSC/d5	Complete 32-Node iPSC system
iPSC/d6	Complete 64-Node iPSC system
iPSC/d7	Complete 128-Node iPSC system

Cube Options*

Part Number	Description
iPSC/d4M	Complete 16-Node iPSC system with 4.5 MBytes of RAM per node
iPSC/d5M	Complete 32-Node iPSC system with 4.5 MBytes of RAM per node
iPSC/d6M	Complete 64-Node iPSC system with 4.5 MBytes of RAM per node
iPSC/d5E	Field Upgrade for existing systems. Upgrades d5 to d6 Hypercube
iPSC/d6E	Field Upgrade for existing systems. Upgrades d6 to d7 Hypercube

Cube Manager Options

Part Number	Description
iPSC/DEV	iPSC Software Development Station
iPSC/SX	Serial Expansion Kit. Adds 8 serial RS232 lines for terminal support
iPSC/MX	2 MBytes additional RAM memory for Cube Manager or Development Station
iPSC/NET	Ethernet, TCP/IP network interface
iPSC/ETC	Ethernet Transceiver & Cable hardware for use with iPSC/NET option

Optional Software

Part Number	Description
iPSC/SRC	iPSC Software Source Code. Requires iPSC Source License Agreement
iPSC/SIM	iPSC Simulator for execution on VAX/BSD4.2 systems or Intel Cube Managers/Development Stations. Simulator source code included for those who wish to re-host on alternate systems. Requires simulator license agreement.

*Each base system includes software and languages, a 1-, 2-, or 4-computational unit hypercube, a Cube Manager, alphanumeric terminal, transceivers, and necessary cables.

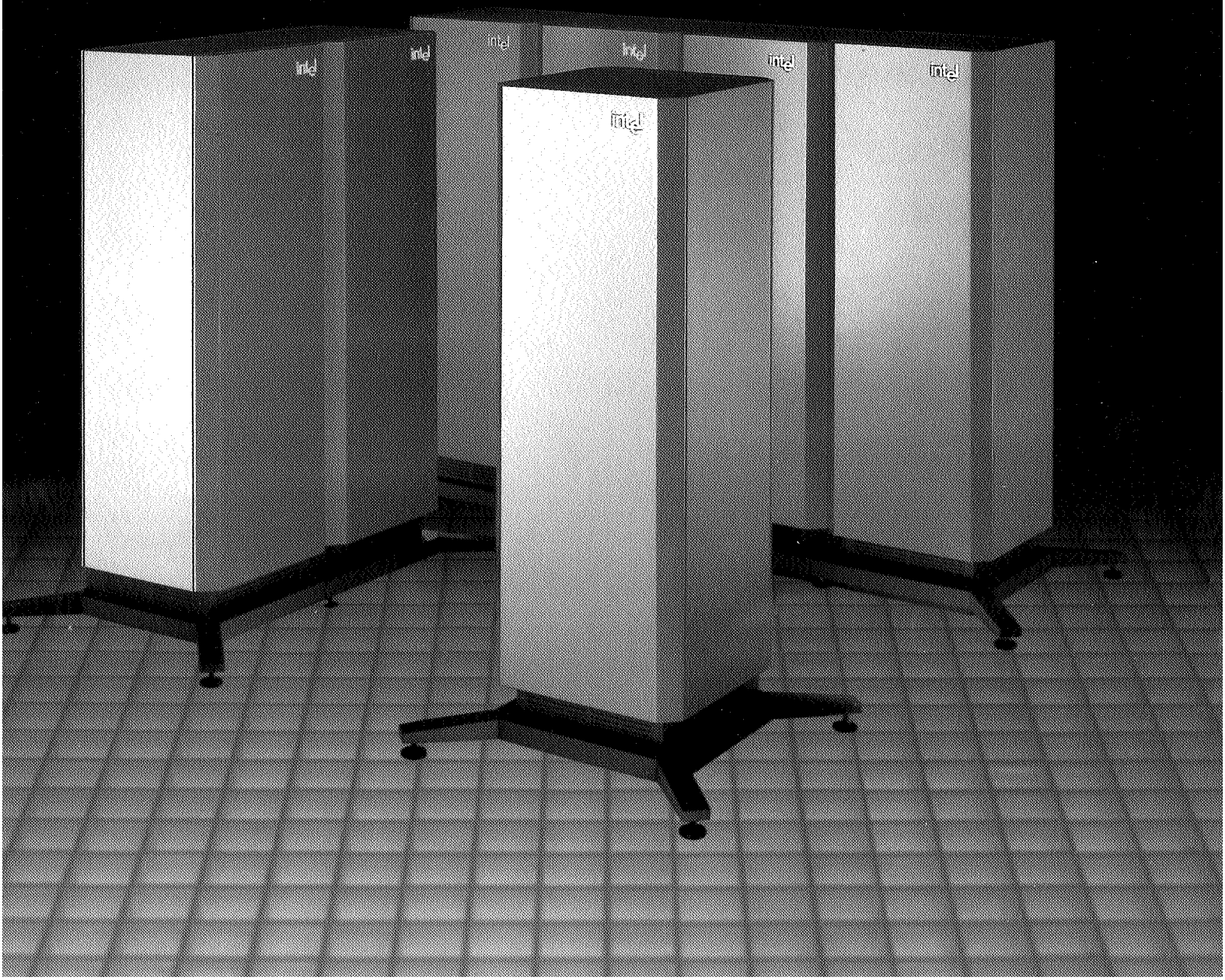
INTEL SCIENTIFIC COMPUTERS
15201 N.W. Greenbrier Parkway
Beaverton, Oregon 97006
(503) 629-7629

Printed in USA/SC6007/5K/LC/VJC
SCIENTIFIC COMPUTERS

intel®

iPSC™

A NEW DIRECTION IN SCIENTIFIC COMPUTING



iPSC: The First Family of Concurrent Supercomputers

In the world of scientific research, there is a clear need for affordable computational power — for a family of supercomputers priced in the range of superminis. And, with current supercomputer technology beginning to reach its performance limits, new architectural approaches are needed that will allow scientific computers to keep pace with the computational problems they'll have to solve in the years to come.

Today, Intel is meeting these needs. The new iPSC™ family of concurrent computing systems puts supercomputer performance in the hands of the individual researcher.

The iPSC is a multiprocessor system that combines a proven parallel processing architecture with Intel's strength in VLSI technology. The concurrent operation of as many as 128 independent processing units ensures high performance, while the use of standard VLSI microcomputers guarantees low cost and high reliability. These processing units, or nodes, are connected using a hypercube architecture.

The Hypercube Architecture

The hypercube is a versatile concurrent architecture. It provides the high-performance environment needed for a broad range of research applications. The multidimensional structure makes the hypercube well suited to both homogeneous and heterogeneous computational problems. Its communication properties of high data bandwidth and low message latency provide the computational efficiency needed to ensure high performance.

Three Compatible, Upgradable Models

Three iPSC models are available, consisting of 32, 64, or 128 processing nodes. Each system can be field upgraded to larger, higher-performance models by modularly expanding the number of processing nodes. System performance and memory capacity double as the number of nodes doubles.

The iPSC System Organization

Each iPSC system consists of two major functional elements: the **Cube** and the **Cube Manager.**

The **Cube** provides the multiprocessor computational power of the iPSC system. It can be configured in 32-, 64-, or 128-node versions to meet expanding performance requirements.

Each node of the Cube is an independent board-level processor supporting 34-, 64-, and 80-bit IEEE floating-point formats with high-speed versions of the 80286/80287 microcomputer chip set and 512K bytes of local RAM.

Resident in each node is a message-passing, kernel operating system. The kernel supports multiple processes per node, automatic routing of messages between non-adjacent nodes, and dynamic message buffer management.

The Cube nodes are connected by means of point-to-point dedicated high performance communication channels in the hypercube topology. To support high-performance interprocessor communications, each board contains eight independent VLSI communication coprocessors.

The **Cube Manager** provides a high-level interface for system users. It serves as the local host for the Cube, supporting the programming environment, communication/control software, and the system diagnostic facility.

The Cube Manager is an Intel System 310 microcomputer system running the XENIX* 3.0 operating system. The Cube Manager provides a user-friendly software environment for developing and using cube applications.

Cube management functions include compilation, program loading, input/output, and error handling. To maximize computational efficiency, the Cube Manager treats the Cube as a single-user device.

System diagnostics, which allow isolation of failures to the board level, are also controlled by the Cube Manager.

System Communications between the Cube Manager and the Cube occur over the high speed Global Channel, connecting each node in the system. This channel serves as a two-way highway for distribution of programs, data, and control operations, as well as for communicating computational results, requests for data, and status.

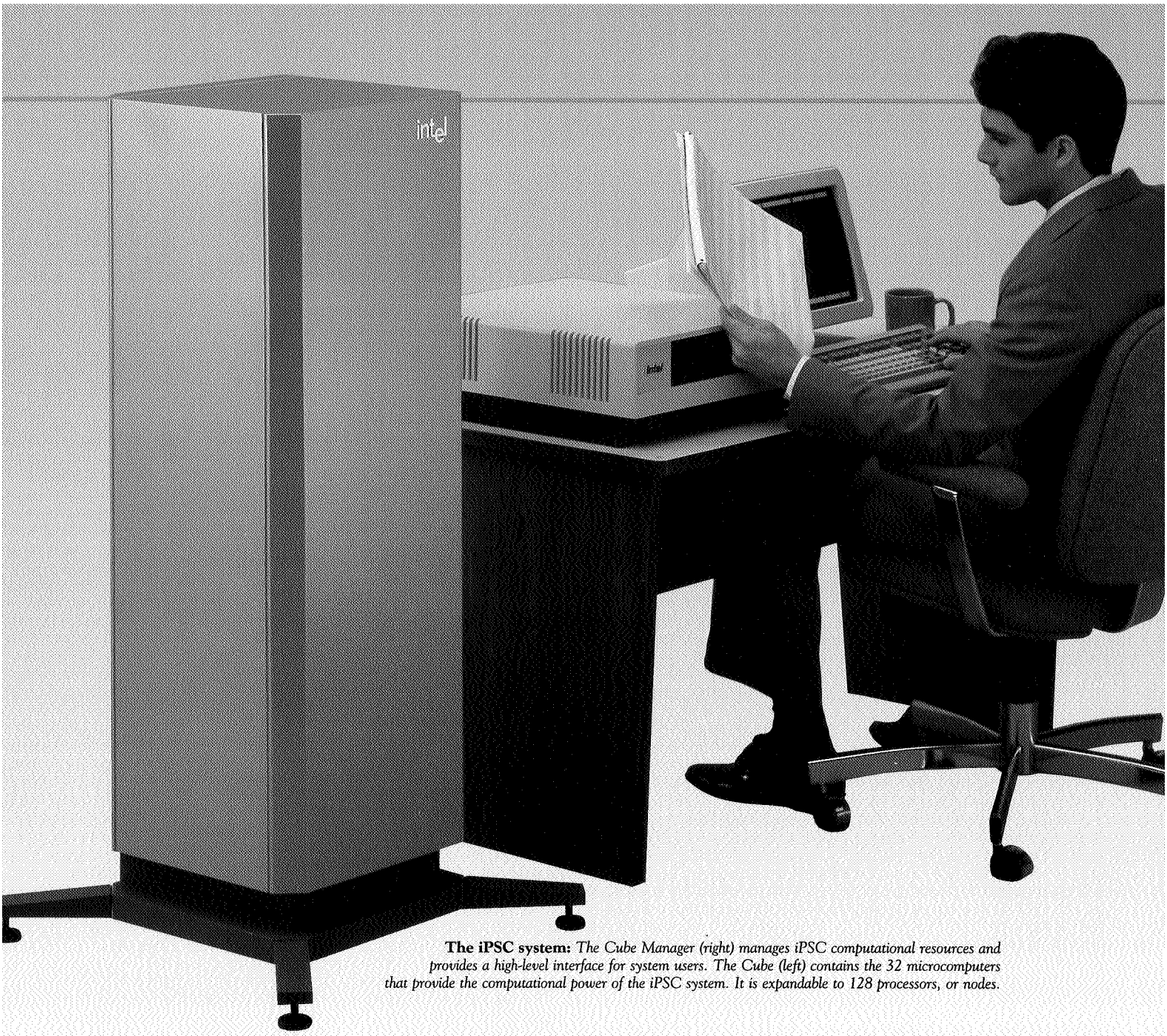
The iPSC system *achieves high performance using a multiprocessor hypercube architecture. The hypercube is a multi-dimensional interconnect topology that can be expanded to meet the needs of the computational problem. Nearest neighbor communication efficiency is optimized, to conform to the requirements of real numerical modeling problems.*

THE HYPERCUBE TOPOLOGY

Dimensions	Nodes	Channels	
d0	1	0	
d1	2	1	
d2	4	4	
d3	8	12	
d4	16	32	
d5	32	80	

On The Cover: *The iPSC is available in three models. The iPSC/d5 (foreground) consists of 32 processing nodes. The iPSC/d6 (background, left) has 64 processing nodes, while the iPSC/d7 (background, right) has 128 processors. Each node is an independent board-level processor with high-speed versions of the 80286/80287 microcomputer set with local RAM memory.*

iPSC systems are modular and expandable up to 128 nodes. Expanding the system increases system performance proportionally, while maintaining software compatibility.



The iPSC system: *The Cube Manager (right) manages iPSC computational resources and provides a high-level interface for system users. The Cube (left) contains the 32 microcomputers that provide the computational power of the iPSC system. It is expandable to 128 processors, or nodes.*

THE iPSC FAMILY

	iPSC MODEL		
	d5	d6	d7
Processing Nodes	32	64	128
Memory, MBytes	16	32	64
Communication Channels	80	192	448
MIPS*	25	50	100
MFLOPS*	2	4	8
Whetstones, KWIPS*	5,750	11,500	23,000
Aggregate Memory Bandwidths (Mbytes/sec)	512	1,024	2,048
Aggregate Communication Bandwidth (Mbytes/sec)	64	128	256
Aggregate Messages (K messages/Sec)	180	360	720

*Based on 80% computational efficiency relative to message traffic overhead.

Applications

Concurrency is an inherent aspect of virtually all scientific problems. The architecture of the iPSC takes advantage of this concurrency to provide high-performance for a variety of applications.

Computational Algorithms

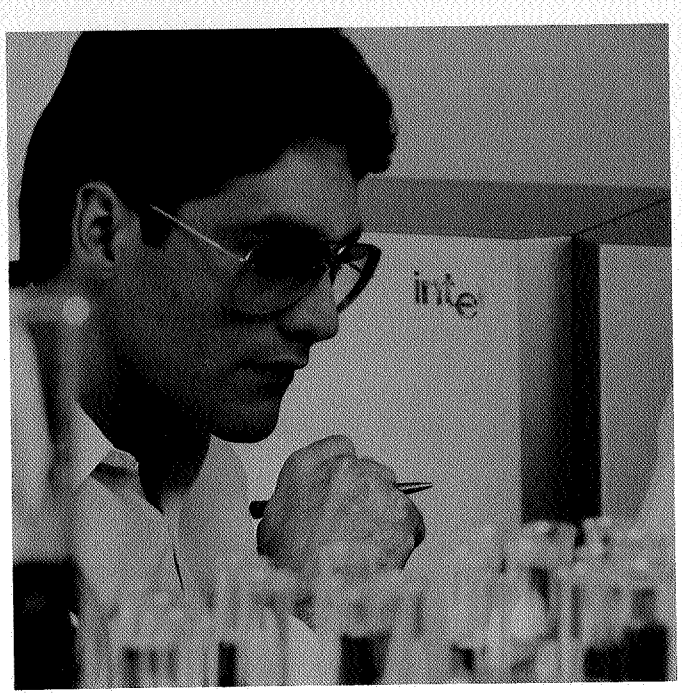
The versatility of a computer architecture can be measured by the range of computational algorithms which efficiently use the machine. The effectiveness of the hypercube architecture has been demonstrated on a diverse range of algorithms and mathematical processes, including:

- FFT and Filter Techniques
- Finite Difference Equations
- Finite Element Modeling
- Monte Carlo Simulations
- Ordinary Differential Equations
- Partial Differential Equations
- Ray Tracing
- Relaxation Techniques
- Sparse Matrix Techniques

Computer Science

In addition, to help computer scientists advance the field of concurrent processing, the iPSC can function as an experimental vehicle for further research into concurrent computing, in such areas as:

- Algorithms
- Artificial Intelligence
- Languages
- Operating Systems



The affordability of the iPSC makes supercomputer power accessible to researchers directly in the lab — almost like having a personal supercomputer.

Scientific and Engineering Applications

Scientific applications have a common denominator in the mathematics which describe the physical phenomena. The iPSC is well suited to these computational needs — both as a computational engine for direct solution of the mathematical representation of the problem, and as a physical analog of the system being modeled. Among the applications for which the iPSC is particularly well-suited are:

- **Physical Sciences**

Biochemistry	Genetic Engineering Pharmaceutical Research
Chemistry	Molecular Dynamics Molecular Mechanics Quantum Chemistry Quantum Mechanics Statistical Mechanics
Geophysics	Granular Motion Mantle Dynamics Oil Reservoir Modeling 3-D Seismic
Physics	Astrophysics Fluid Mechanics High Energy Physics Meteorology Quantum Field Theory

- **Engineering**

Aerospace	Aerodynamics Flight Simulation Structural Mechanics
Electrical	Linear Circuit Simulation Logic Simulation Rules Checking
Image Processing	Animation Graphics Image Enhancement Image Synthesis
Mechanical	Fluid Dynamics Structural Analysis

The iPSC System in Brief

Concurrent architecture	High performance from as many as 128 microcomputers working concurrently. Modular upgradability — expandable from 32 to 64 to 128 nodes with software compatibility.
Hypercube interconnect	Efficient communication — point-to-point path between each node and its neighbors. Flexibility — adaptable to various topologies (ring, tree, 2D, 3D, etc.) to suit the application.
Open system	Flexibility — Cube Manager is expandable according to application requirements — with a choice of over 1500 MULTIBUS® products.
Network Interface	Convenient integration into existing TCP/IP Ethernet networks providing immediate iPSC access to network users via remote log-in and file transfer.
XENIX* 3.0 operating system	Licensed UNIX System III and Version 7 compatible system software — supporting FORTRAN 77, C, and 286 ASM languages for ease of developing and maintaining software.
80286/80287 processors	High performance scalar and floating point capability at an affordable cost using commercially available VLSI.
Large, distributed system memory	Large physical memory space — up to 64 MBytes of distributed memory — for handling large-scale problems without memory contention and disk transfer overheads. Incrementally expands with the number of nodes.
iLBX™ II Interface	Node expandability — provides a facility to extend memory or processing capabilities with the addition of enhancement boards in slots adjacent to node (processor) boards.

Service and Support

Intel is committed to providing complete post-sale support to iPSC customers. On-site hardware maintenance is provided for part of the one-year warranty period, and after that is available on a contract basis from over 60 Intel product service offices in the continental U.S. Dedicated factory-based customer support resources are in place to provide complete software and applications support.

Services included with the purchase of any iPSC

Installation	On-site installation by Intel personnel is included with system purchase. Intel also provides site preparation assistance to help insure a successful installation.
Training	Training is included to instruct customers in the use and programming of the iPSC.
Service/Support	A one-year warranty is provided for hardware. Software support is also provided for one year.

Hardware service is provided through on-site maintenance by trained Intel product service personnel.

Software support includes updates, technical reports, software problem reporting, and telephone HOTLINE assistance.

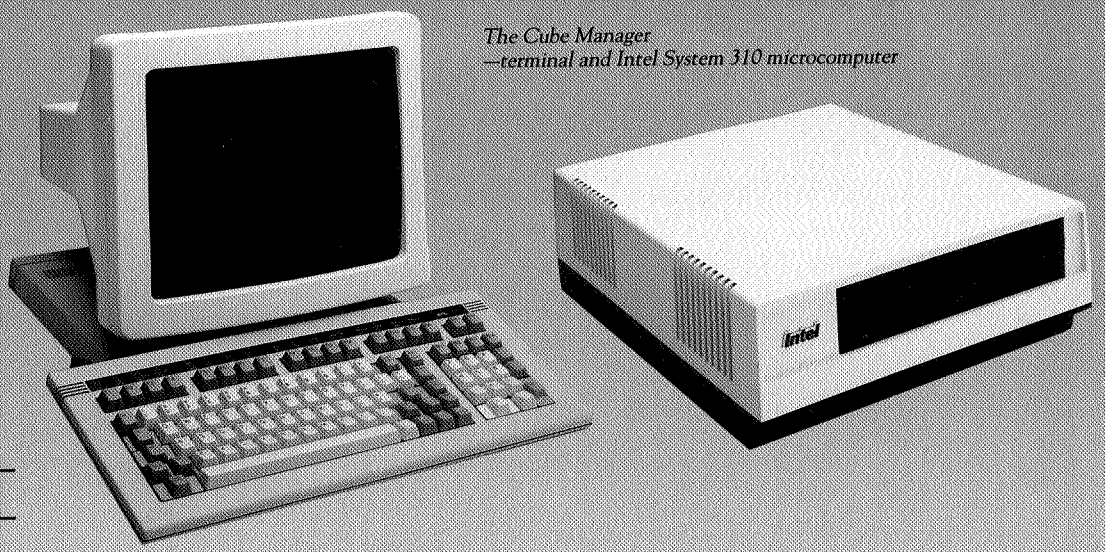
Post-Warranty Services

Hardware Maintenance	On-site maintenance is available under an annual service agreement or on a per-call basis. Factory exchange service is also available.
Software Support	Factory-based software support is available under an annual service agreement.

Applications Support

Part of our long-term commitment to scientific computing customers includes a team of applications specialists who are dedicated to the development of concurrent software technology.

iPSC Specifications



The Cube Manager
—terminal and Intel System 310 microcomputer

iPSC System Software

Programming and Development Software

- XENIX 3.0 Operating System
  - AT&T System III UNIX**
  - UNIX Version 7 compatible
  - Berkeley UNIX features
  - Microsoft enhancements
- Languages
  - FORTRAN 286 — FORTRAN 77 with extensions
  - C — Native language for XENIX
  - ASM286 — Assembly language for 286/287
- Tools and Utilities
  - Binder/Builder
  - Loader
  - System configurator

Cube Control Services

- Initialization
- Loading
- System status reporting
- Cube access control

Cube Communications Services

- High-level access to Cube
- Message generation and control
- Cube communications

Diagnostic Services

- System Diagnostic Testing
  - Cube Manager
  - Communications
  - Nodes
- System Confidence Testing
  - Global channel
  - Node channel
  - Arithmetic computation

Node Services

- Node Monitor
  - Self-test
  - Initialization
  - Loading
- Node Kernel
  - Interprocess Communications
  - Process Management
  - Physical Memory Management
  - Address Space Protection
  - Error Reporting and Exception Handling

Cube Manager Hardware

Architecture

- High Performance
- "Open System" Support (IEEE 796 MULTIBUS)
- Expandable – 3 slots

Processing Elements

- 80286 Central Processing Unit
  - Virtual address: 1 GByte
  - Direct address: 16 MBytes
- 80287 Numeric Processing Unit
  - High Performance Numerics
  - IEEE Floating-Point Formats

System Memory Capacity

- 2 MByte with ECC
- 4 MBytes w/expansion option

Mass Storage Capacity

- 40 MByte 5¼" Winchester
- 320 KByte 5¼" Floppy Disk

Input/Output Capability

- Terminal: Alpha-numeric
  - ANSI X3.64 (VT 100) Compatible
  - CRT: 14" Diagonal
  - 24 rows×80 or 132 columns
  - Serial Printer Port
- Parallel Printer Port
  - IEEE Standard (Centronics)
- Multi-Line Serial Interface Option
  - 8 Lines Up to 19.2 Kbps
- TCP/IP Ethernet Interface Option
  - Remote Log-in
  - Remote File Transfer
  - Mail

Cube Hardware

Hypercube Architecture

- Communications
  - Dedicated point-to-point channels
  - High system bandwidth
  - Low communication latency
- Flexible
  - Multidimensional Topology
  - Supports meshes of lower dimension (ring, tree, 2D mesh, 3D mesh, etc.)
  - Expandable (from 32 nodes to 64 nodes to 128 nodes)
- Connections per node

32-node (d5)	5
64-node (d6)	6
128-node (d7)	7

Node Processors

- 80286 Central Processing Unit
  - High performance integer processing
  - Memory Protection
- 80287 Numeric Processing Unit
  - High performance floating-point processing
  - IEEE 32-, 64-, 80-bit precision arithmetic (IEEE 754)
- Memory
  - 512 KBytes Dual Port RAM per node with parity
  - 64 KBytes PROM (expandable)
- Internode communication
  - 7 bi-directional channels/node
  - 10 MBits/sec per channel (hardware limit)
  - Reliable Message Delivery to Nearest Neighbor
- I/O Ports
  - Global Link
  - iLBX™ II Expansion Port
  - RS422 Diagnostic Port

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 79 HILLSBORO, OREGON

POSTAGE WILL BE PAID BY:

intel®

Marketing Department
Intel Scientific Computers CO-01
5200 N.E. Elam Young Parkway
Hillsboro, OR 97124-9990

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

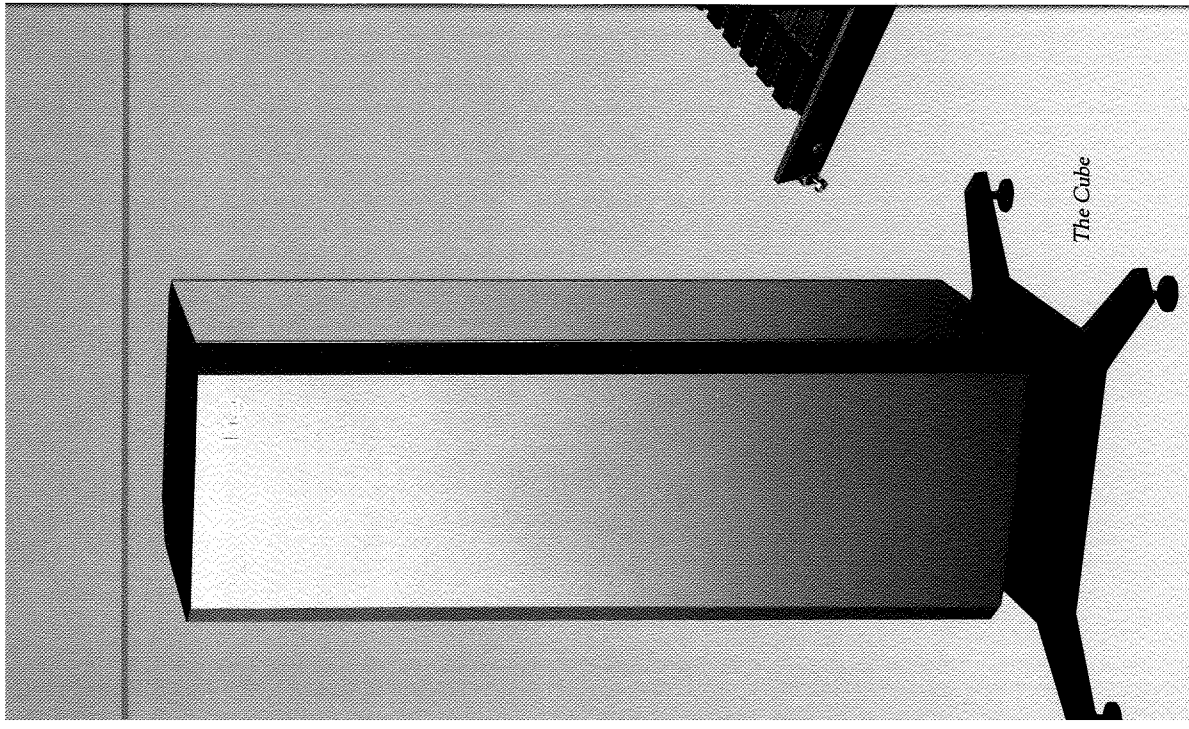
BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 79 HILLSBORO, OREGON

POSTAGE WILL BE PAID BY:

intel®

Marketing Department
Intel Scientific Computers CO-01
5200 N.E. Elam Young Parkway
Hillsboro, OR 97124-9990



The Cube

Environmen

Acoustical No
- Appropriate environmer

Safety & Emn
- Designed an safety speci CSA
- Designed an conducted specificatio

Power (per 32-
- 2700 watts
- 240 VAC 1
- 208 VAC 13
- 10,000 BTU

System Memory Capacity

■ 32 Node (d5):	16 MBytes
■ 64 Node (d6):	32 MBytes
■ 128 Node (d7):	64 MBytes

Input/Output
- Global Channel
  - CSMA/CD Baseband (Ethernet 802.3)
  - Access to all nodes
  - 10 Mbits/sec (Hardware limit)
  - Variable block size- 1 Byte to 16 KBytes
  - Reliable Message Delivery
- Diagnostic Channel
  - RS422 Protocol
  - Dedicated access to all nodes via Unit Services Modules

Data Formats

■ Integer	8, 16, 32, 64-bits	
■ BCD:	18-digits	
■ Floating-point, IEEE formats		
32-bits	7 digits	$10^{\pm 38}$
64-bits	16 digits	$10^{\pm 308}$
80-bits	19 digits	$10^{\pm 4392}$

intel®

Intel Corporation
Founded in 1968, Intel Corporation has become a recognized leader in technological innovation. The iPSC family continues that tradition. By combining Intel's VLSI leadership in microprocessors, memory, and communications with a proven parallel processing architecture, the iPSC represents a new direction in supercomputer technology. With this strategic combination of leading-edge technologies, Intel delivers the cost reductions that will make concurrent supercomputing the future of large-scale computation.

For more information on the iPSC, write to:

Intel Scientific Computers
Director of Marketing CO-01
15201 N.W. Greenbrier Parkway
Beaverton, OR 97006

Or call:
(503) 629-7629

Printed in U.S.A./iSC5-53/SWW/685/20K/VJ

Order #280093-002

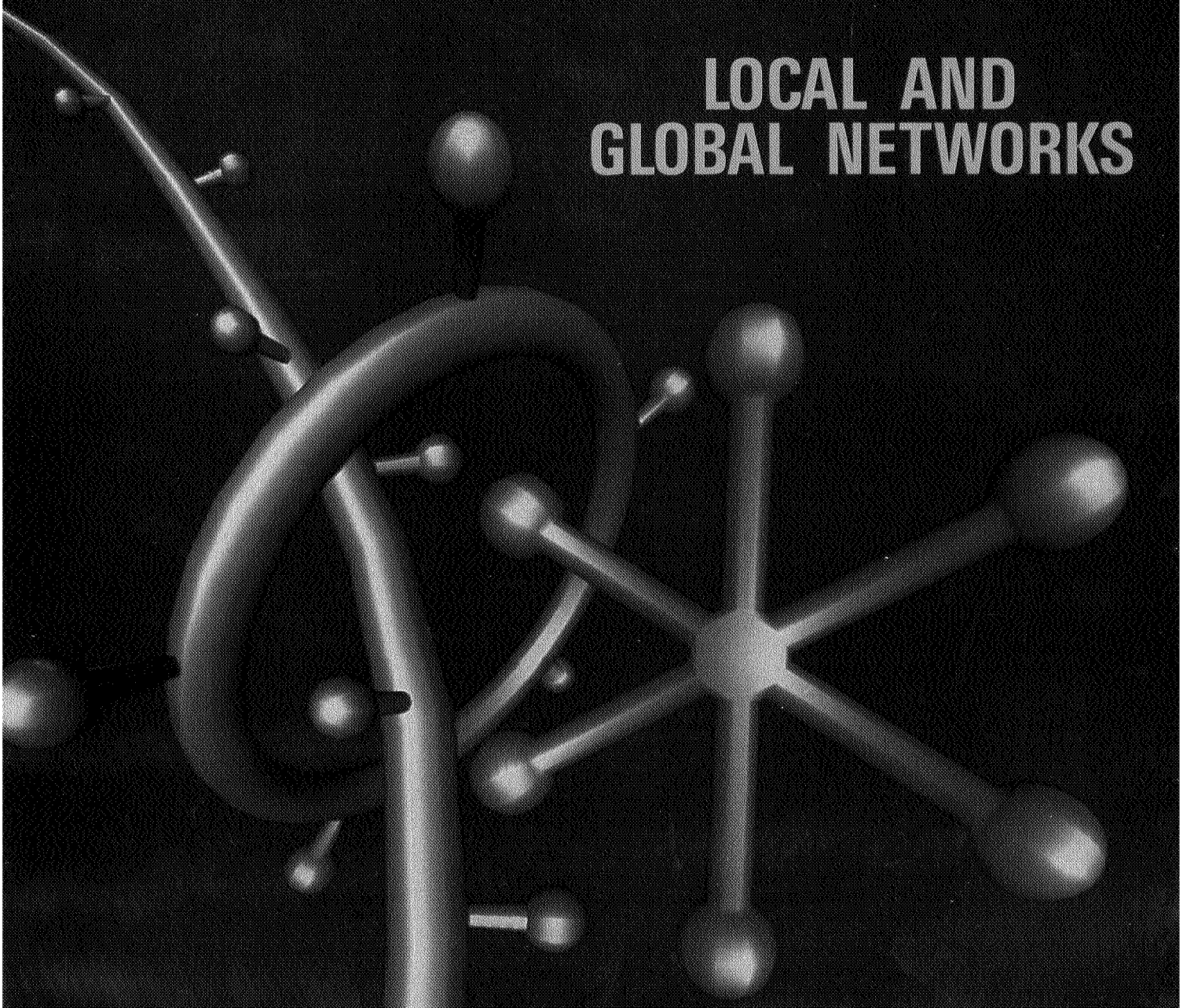
A PennWell Publication

SEPTEMBER 1, 1985

COMPUTER DESIGN

AR-416

LOCAL AND GLOBAL NETWORKS



LOGIC ANALYZERS OFFER NEW CHOICES IN PERFORMANCE AND PRICE

DUAL-POINTER FIFO ELIMINATES BUS ARBITRATION

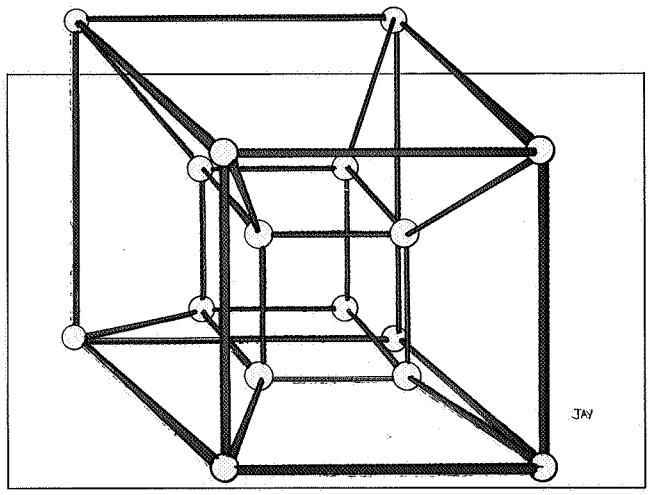
SPECIFY WISELY TO LOWER POWER SUPPLY COSTS

Order Number: 280197-001

CONCURRENT COMPUTERS IDEAL FOR INHERENTLY PARALLEL PROBLEMS

Processors and nodes mimic scientific problem geometry with expandable, parallel processing architecture.

by Ray Asbury,
Steven G. Frison, and
Thomas Roth



Concurrent processing represents the most promising long-term approach to achieving affordable, accessible supercomputing now that the limitations of supercomputers—especially those handling combined scalar and vector operations—are becoming evident. Concurrency is a high-level, or global, form of parallelism that denotes independent operation of a collection of simultaneous computing activities, rather than the lockstep connotation of the more familiar term "parallelism." Concurrency is essentially an interactive parallelism that allows asynchronous operation of processors in a multiprocessor system.

Solving problems on a concurrent machine requires partitioning them into a number of segments that can run independently on more than one processor. Many applications, particularly in scientific computing, lend themselves to this partitioning. Intel's parallel architecture computer, the hypercube, is well suited for these partitioned applications.

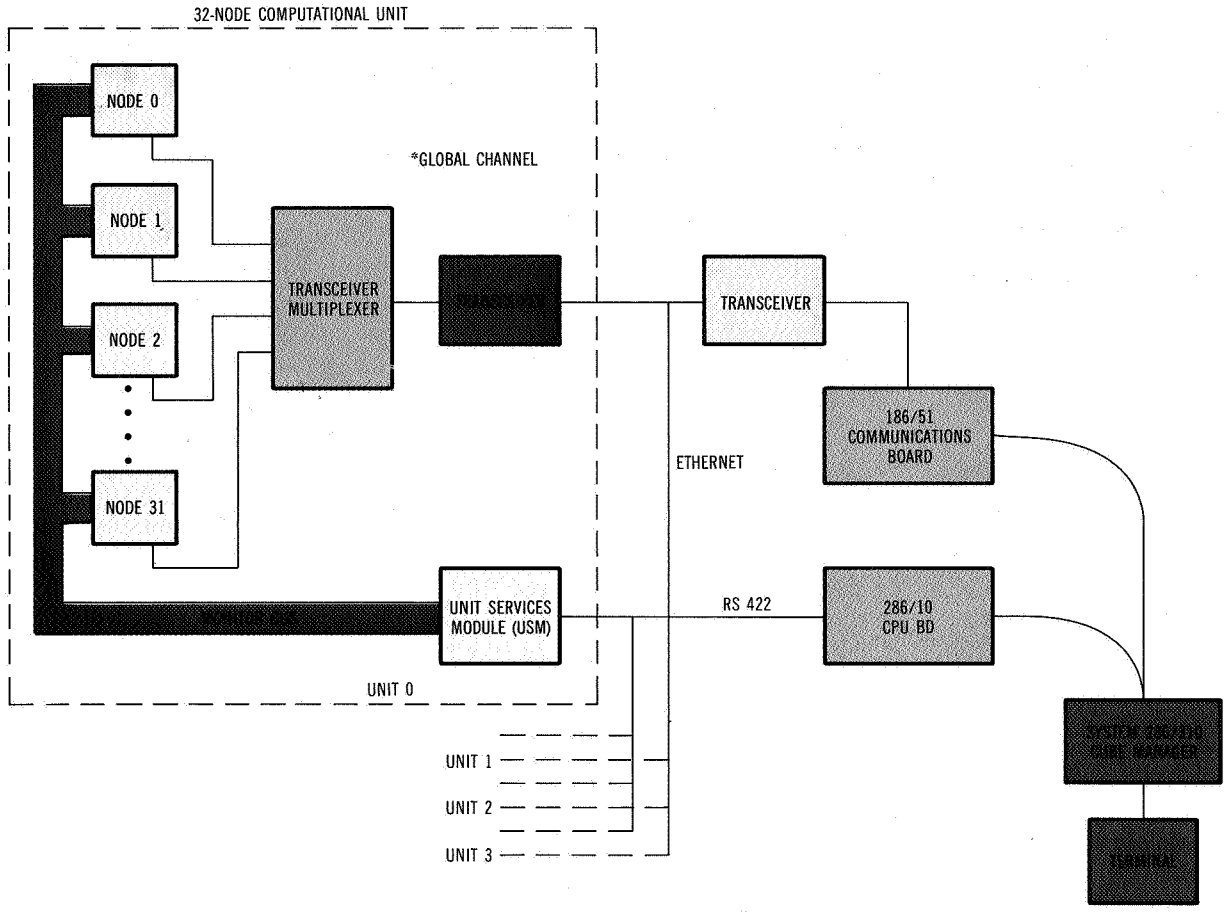
The iPSC hypercube achieves concurrency through an ensemble of loosely coupled, independent processors executing portions of a larger computational problem simultaneously. The hypercube consists of 32, 64, or 128 microcomputers connected to each other via point-to-point communications channels. Each processor is connected directly to a local host processor—the cube manager—via a global communications channel. The cube manager supports the programming environment and serves as the cube's system manager.

Hypercube architecture originated from research at the California Institute of Technology sponsored by the Defense Advanced Research Projects Agency (DARPA). This architecture uses individual microprocessors and associated local memories to form computational nodes that are tied together in a network. For a hypercube of dimension n, 2^n nodes are

Ray Asbury is an applications specialist at Intel Corp (Beaverton, OR). He holds an MS in physics from the University of New Mexico.

Steve Frison is system software manager at Intel Corp. He holds a BS in mathematics and computer science from Portland State University.

Tom Roth is a senior systems engineer at Intel Corp. He holds a BS in applied science and engineering from Portland State University.



Sporting a cube-managing 286/310 microcomputer and up to four 32-node computational units connected by global and diagnostics channels, the iPSC computer systems depend on the Xenix operating system for computer management, and Fortran and C for most of its application programs. An Ethernet network allows local communications.

interconnected by point-to-point communications channels. Hypercube architecture calls for individual nodes to operate independently on a subsection of a larger problem. These individual nodes work independently, according to resident process instructions, on data resident in a specific processor's memory. This data can come through a message-passing system from processes resident in other nodes, or from the cube manager. Process code is written in ordinary sequential languages. Fortran, for example, is used with operating system primitives provided by the cube's operating system. These primitives allow programmers to direct message sending and receiving.

Why hypercube?

The hypercube topology presents several advantages. First, for large systems consisting of hundreds, or even thousands of nodes, the scale of the architecture can be increased by expanding the dimensionality or size, of the cube. Second, its distributed memory (the memory resident at each node) avoids the problems of contention between many processors for a shared memory. Third, its communication properties of high data bandwidth (which grows as $N\log_2 N$), and low message latency (the worst case being $\log_2 N$) help hypercube-based systems achieve computational efficiency. Finally, the robust interconnect structure makes the hypercube adaptable to other topologies, such as ring, tree, and two- and three-dimensional meshes.

Application programs are developed and compiled on the cube manager and then downloaded to the cube nodes. The cube manager is an Intel 286/310 multi-user supermicrocomputer. This supermicro

runs the Xenix 3.0 operating system, which is Intel's version of Unix, as well as associated Fortran and C compilers.

Unlike many parallel processing architectures, the iPSC is a multiple-instruction, multiple-data (MIMD) stream machine that uses message passing rather than shared storage for communication between nodes. Message passing is used to perform the basic unit of cube computation—the "process." A process is defined as a sequential program including system calls that cause messages to be sent and received. In fact, processes communicate only by sending and receiving messages. This architecture provides increased overhead efficiency compared to memory-sharing schemes.

Computing via multiple processes

A single node may contain many processes that perform computations. Computations are distributed through the computer and executed concurrently, either by virtue of being in different physical nodes, or by being interleaved in execution within a single node.

The hardware model of the iPSC is quite similar to the process model of computation. As a result, instead of formulating a problem to fit on nodes, and on the physical communication channels that exist only between certain pairs of nodes, the software developer may formulate a problem in terms of processes and virtual communication channels that connect all processes. This abstraction requires the cube message system to route messages efficiently from any one process to any other process.

The code for a process may be written in any combination of Fortran, C, or ASM286, the 80286's assembly language. It is linked to the node application library using the 80286's binder utility (BND286). Finally, the user process is bound to the kernel with a utility (BLD286) to create a loadable object module.

Each process has a unique 32-bit identifier in which two 16-bit fields are defined. These are the processor (node) number and the process number (process ID) within the node. Process IDs in the range 32,768 to 65,535 are reserved for the kernel. If the process number field is treated as a signed number, user process IDs are positive and reserved process IDs are negative. The size of available memory within the node limits the number of processes per node. The node number ranges from zero to the number of nodes in the cube minus one. In a 32-node cube, for example, the node numbers range from 0 to 31.

The node kernel provides basic services, such as interprocess communication, process management, physical memory management, and protected address space. In addition, the node kernel forms a foundation for other operating system functions.

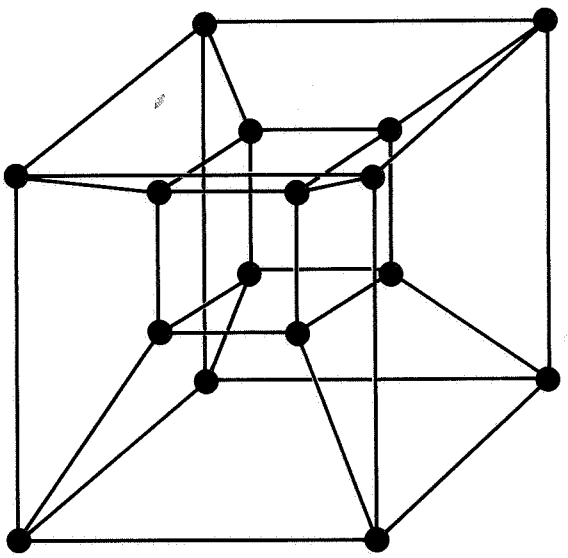
Next, a copy of the node kernel and lined application processes is loaded into each node after initialization and confidence testing have been successfully completed.

Node communication

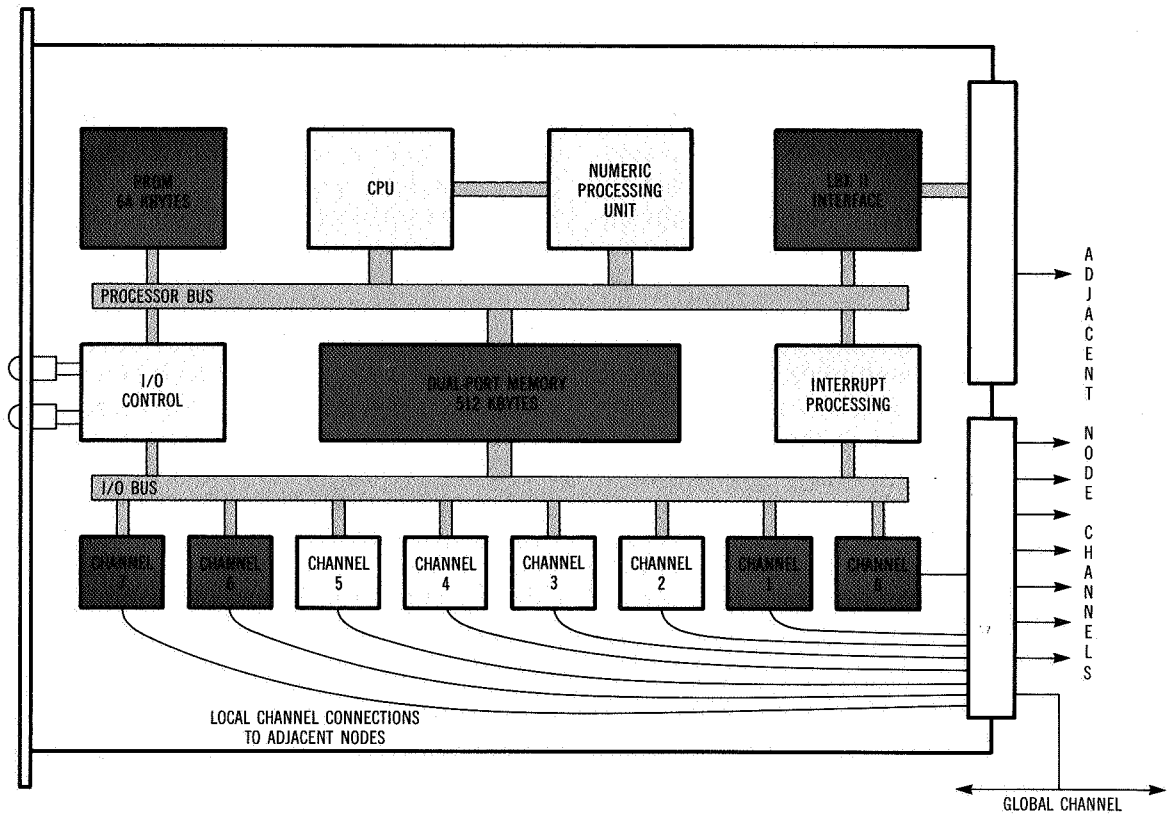
The node operating system provides user processes with a flexible set of communication primitives. The communication mechanisms are the same whether the processes are in the same node, mother nodes, or in the cube manager. The node operating system, in conjunction with the 80286, also provides protection for processes executing within the nodes. This design prevents errors in processes from being transferred to other processes.

The node operating system uses a common message format to support message passing functions. One possible message format is for messages to be passed "by value." This means that a copy of the message from the sender to the receiver is always made, even if the processes are on the same node. Messages may be queued in transit, but message order is preserved between any pair of processes. Message routing in the cube is entirely a function of the node operating system. Shared memory is never used as a communication mechanism.

Interprocess communication is performed through "channels." A channel is a system construct used



A hypercube (binary cube or Boolean n-cube) connects $N = 2^n$ microcomputers (nodes) through point-to-point communication channels. A 16-node (n = 4) cube has each node physically connected to its four nearest neighbors.



There are $\log_2 N$ channels per node. An actual iPSC node fits on a 9- by 11-in. Eurocard and contains a self-sufficient microcomputer.

to manage a process's message requests. There is no synchronization between processes when a channel is established. One process communicates with another process simply by opening up the channel and initiating send and receive requests.

Because message passing between processes is asynchronous, there is no guarantee that a process has made a receive request before the node receives the message. To ensure proper message reception, messages that are received by a node prior to a receive request are buffered by the node operating system until the request is made. The message is then transferred into the process message buffer.

A process performs message sending and receiving by invoking system calls. Send and receive merely enable a message transmission or reception. If the action is not satisfied, a request is created that remains pending until it is satisfied. Program execution may continue concurrently with one or more pending communication requests.

Once a channel has been opened, a send and its corresponding receive may occur in either order. It is generally most efficient if the receive is executed before a message arrives at a node. The receive is, therefore, delivered to the process rapidly, without occupying a node operating-system buffer for an extended period.

A programmer can construct the application to incorporated checkpoints, if system reliability demands it. Additional flexibility is possible because a single message may often contain up to 16 kbytes. Messages larger than 1 kbyte are automatically fragmented and reassembled at the destination node in a process transparent to the user.

Point-to-point communications between adjacent nodes perform the bulk of message transfers. Moreover, reliable message delivery is guaranteed between a node and its nearest neighbors. To reduce communications overhead, end-to-end message acknowledgement is not used between a node and more distant nodes (non-neighbors). This feature can be provided at the applications level, if needed.

Concurrent or vector?

When an application is to be programmed for the hypercube, algorithms must be developed to define the discrete component processes of real physical problems. The process descriptions are then topologically assigned to nodes by the programmer to model events as closely as possible. Many types of

concurrent applications can be mapped to the cube. These applications include geophysics, astrophysics, network simulation, image processing, statistics, or any research where the problems exhibit concurrency.

The program to solve a problem that is inherently concurrent in nature can be separated into two components. First, there is a scalar portion, called the loop, or problem space, which mathematically describes the problem boundary. Then there is a vector portion (inner loop), which is a software model of the interaction between the individual problem elements. A common problem in molecular dynamics—the diffusion of one group of molecules through a medium—is a good example of how concurrent processing and vector processing differ in the way they handle similar problems.

Scalar-bound solutions

On a vector or array processor, the parallelism inherent in the mathematical model of the diffusing molecules (the vector portion of the problem) facilitates high performance. The scalar portion of the problem, however, which describes the interaction of the molecules with the vessel walls, limits performance, because it cannot be vectorized and must run sequentially. Because the scalar portion also limits an optimal solution of the problem, the very best vector performance will have little effect on increasing the overall performance of the application beyond this point. In effect, the solution of the problem becomes scalar-bound.

A concurrent machine, on the other hand, shares the load equally over a large number of processors. Each processor solves a small portion of the overall task and interacts over high-speed communication channels. This division of labor achieves the concurrency needed for the task.

The task of any processor in the hypercube system is scaled proportionately to the size of the segment being processed. Each processor solves a portion of the outer loop and a portion of the problem's inner loop. Both vector and scalar performance remain proportional because the hypercube architecture mimics the structure of the problem itself.

A concurrent machine such as the hypercube can also handle non-numerical problems, such as event-driven simulation and artificial intelligence. This capability is not found in conventional supercomputers, because their architectures are optimized for handling vector calculations.

Commercially available components were used to conceive this generation of concurrent computers. These computers perform at up to one third the speed typical of a Cray-1. For example, the iPSC family incorporates nodes based on the 80286 microcomputer, the 80287 numeric coprocessor, and the 82586 LAN coprocessor. The 82586 LAN coprocessors handle message passing between processing elements.

Each node contains 512 kbytes of dynamic RAM, and 64 kbytes of ROM. The ROM contains the node monitor, which is responsible for node initialization at power-up or system reset. The monitor verifies that the node is operable, and loads the kernel and application software. The monitor initializes the system by resetting and enabling the node memory, communication controllers, I/O controller, interrupt controller, and CPU. The monitor is also responsible for node confidence testing. The confidence test verifies the node's printed circuit board by running a node confidence test (NCT) on the RAM, peripheral devices and communications controllers.

Each of the three hypercube systems within the iPSC family includes a cube that contains the network of microcomputer nodes, and a System 286/310 microcomputer for overall system management. The cubes for the three systems within the family include one, two, or four 32-node computational units.

In addition to seven I/O channels for internode communication, every node board has an eighth channel—an Ethernet link known as the global channel—for interface with the cube manager. For memory expansion or extended processing capacity, the node processor's local memory bus on each board ties into the iLBX-II bus interface, which is defined in the Multibus II standard.

Selecting components

While the hypercube architecture makes the construction of powerful computers possible using off-the-shelf VLSI devices, certain considerations govern the actual component selection. Although programming concurrent processors is only slightly more difficult than programming a sequential CPU, memory security at each node is critical to guarantee reliable operation. Local memory, for example, must accommodate process instructions as well as high-speed message buffering. In the iPSC, each node carries in local memory a copy of the operating system that directs message passing.

The 80286 microprocessor is a good choice for the iPSC for several reasons. It has built-in memory protection, and supports the isolation of the operating system and tasks, as well as program and data privacy within tasks. In application, local descriptor tables for each process resident at a given node provide a partition between the operating system kernel and user space, and between multiple-user process spaces. The 80286 also has a 16-Mbyte real address space which is more than adequate for first-generation hypercubes. A companion communications controller and numeric coprocessor are available to support the 80286. Finally, it supports a variety of

Wait a MOMENT

An example of Fortran programs that use the hypercube concurrent processing capabilities are the programs (processes) STATS and MOMENT. Running on a one-dimensional hypercube, STATS computes the mean and standard deviation of lists of floating-point numbers on one node.

MOMENT runs concurrently on two nodes, and is used by STATS to do most of its arithmetic concurrently; MOMENT and STATS run concurrently on one node. MOMENT repeatedly receives lists of floating-point numbers, computes a sum of powers of those numbers, and sends the sum back to the host process. The power is equal to the node id plus 1, so node 0 computes the sum of the numbers and node 1 computes the sum of the squares of the same numbers.

Data is passed between the processes using CALL SENDW, CALL RECVW, CALL SENDMSG and CALL RECVMSG. When the process STATS needs to do arithmetic, it uses CALL SENDMSG to call MOMENT, and then passes the information to MOMENT. MOMENT is simultaneously awaiting data from STATs, using CALL RECVW.

Once MOMENT processes the data, it issues a CALL SENDW and waits for STATS to issue a CALL RECVMSG. At this point, the program sends the data back to the host process (STATS) on the originating node.

```
                    PROGRAM STATS
      INTEGER K,M,N,CID,HOST,BYTES,XLEN,TYPE
      INTEGER COPEN
      DOUBLE PRECISION X(100),SUM,SUMSQS,TEMP,MEAN,STDEV
      DATA XLEN /800/, TYPE /1/, HOST /-1/
C
C     Open a channel. Use a process id equal to the node id.
C
      CID = COPEN(HOST)
C     Start the main loop. Read the data from the terminal.

   10 WRITE (*,*) 'Enter n, x(1), ..., x(n)'
      READ(*,*) N, (X(I),I=1,N)
      IF (N .LE. 0) STOP
      WRITE(*,*) 'N = ',N
C
C     Ask node 0 to compute sum of x(i) and
C     ask node 1 to compute sum of x(i)**2.
C
      BYTES = 8*N
      CALL SENDMSG (CID,TYPE,X,BYTES,0,0)
      CALL SENDMSG (CID,TYPE,X,BYTES,1,1)
C
C     Wait for two replies, which can come in either order.
C     M is the id of the node originating the reply.
C
      BYTES = 8
      DO 40 K = 1, 2
        CALL RECVMSG (CID,TYPE,TEMP,BYTES,BYTES,M,M)
        IF (M .EQ. 0) SUM = TEMP
        IF (M .EQ. 1) SUMSQS = TEMP
   40 CONTINUE
C     Compute the statistics and print the results.

      MEAN = SUM/N
      STDEV = DSQRT (SUMSQS/N - MEAN**2)
      WRITE(*,*) 'MEAN = ', MEAN
      WRITE(*,*) 'STDEV = ', STDEV
      WRITE(*,*)
C
C     Do it again
C
      GO TO 10
      END
```

```
                    PROGRAM MOMENT
      INTEGER I,M,N,CID,HOST,BYTES,XLEN,TYPE
      INTEGER MYNODE,COPEN
      DOUBLE PRECISION X(100),SUM
      DATA XLEN /800/, TYPE /1/
C
C     Find the node id
C
      M = MYID ()
C
C     Open a channel. Use a process id equal to the node id.
C
      CID = COPEN(M)
C
C     Start the main loop.
C
   10 CONTINUE
C
C     Wait for n and x.
C
      CALL RECVW (CID,TYPE,X,XLEN,BYTES,HOST,HOST)
      N = BYTES/8
C
C     Compute the desired sum.
C     Note that the node id is involved in the arithmetic.
C
      SUM = 0.0D0
      DO 20 I = 1, N
        SUM = SUM + X(I)**(M+1)
   20 CONTINUE
C
C     Send the sum back to the host.
C
      CALL SENDW (CID,TYPE,SUM,8,HOST,HOST)
C
C     End of the main loop.
C
      GO TO 10
      END
```

standard software environments and the CPU's local bus interface boasts an 8-Mbyte/s bandwidth.

The 80287 numeric processor handles 32-bit, 64-bit, and 80-bit floating-point arithmetic operations in conformance with the draft IEEE-754 floating-point standard. Each 82586 LAN coprocessor provides a 10-Mbit/s pathway for internode transfers. Typically, the devices permit a 100-MIPS aggregate communication rate for the 128-node system. The complex responsibilities of each node dictates selection of devices that integrate the maximum number of functions, yet interact efficiently (a 128-node

hypercube incorporates eight separate communications channels/node, for a total of 1024 active I/O ports).

Without VLSI that integrates most of the control functions for at least one port, the concurrent architecture would become cost-prohibitive. With nodes being formed from such VLSI-intensive devices, compact hypercube systems can be easily expanded. The iPSC system is expanded from one to two or four 32-node computational units by connecting them with cables.

Future of concurrent processing

Concurrent processing allows great increases in computational power. Of all concurrent processing architectures, the hypercube's primary benefit is its scalability. Systems made up of thousands of nodes may become typical.

In the near future, the majority of hypercube system enhancements will focus on greater performance. These performance enhancements can occur in many areas. For example, 32-bit architectures, higher performance processors, and the addition of more floating-point capabilities are all possible. These can more than double individual node board performance. Also, increased memory capabilities will allow larger granularity per node, permitting the solution of larger problems.

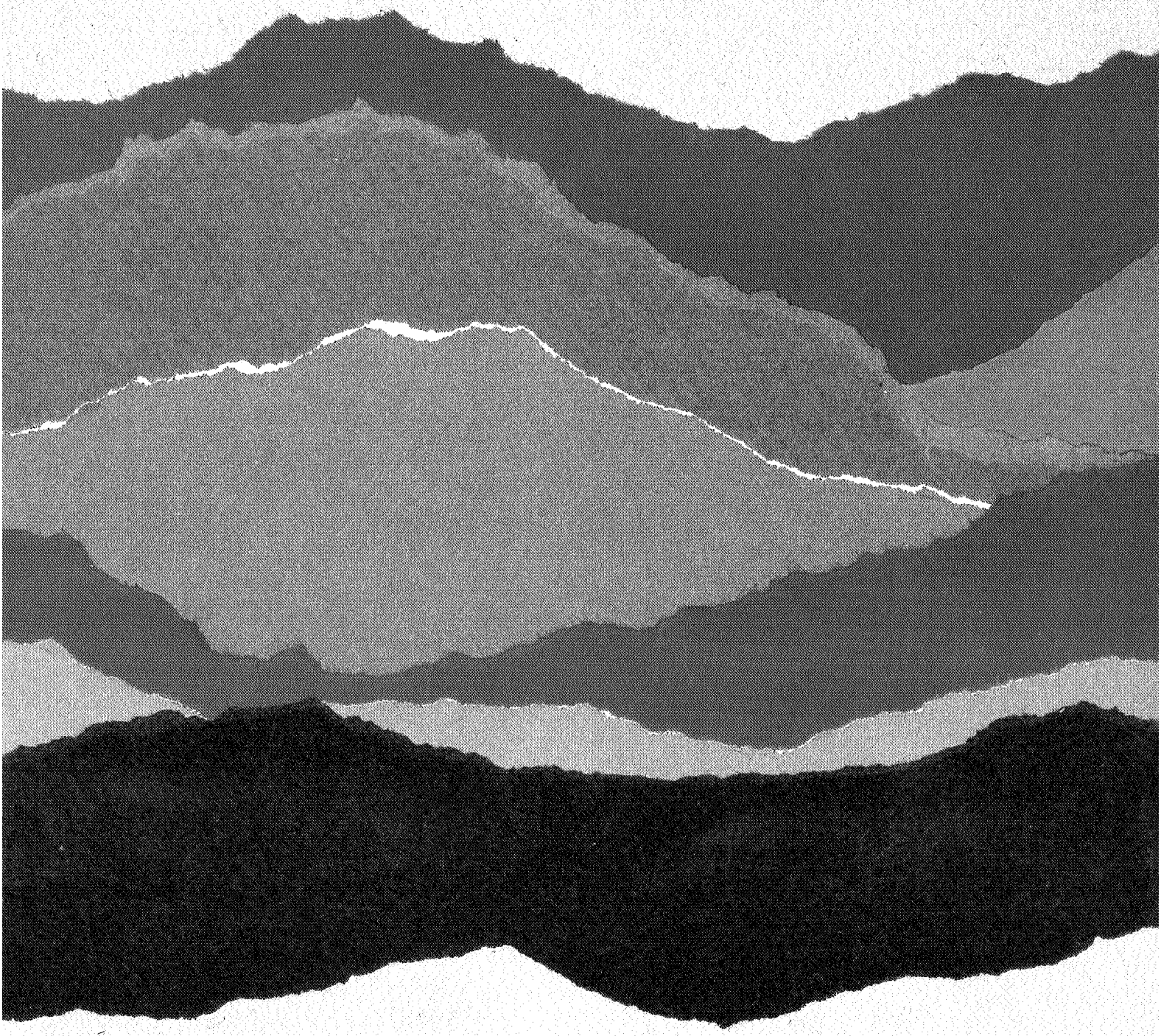
As levels of computing capability increase, more processors in larger systems may become the norm, moving beyond the 128-node level. In the more distant future, implementations will use dedicated VLSI devices, rather than off-the-shelf components. And rather than printed circuit boards, entire nodes—possibly multiple nodes—will be contained on single chips. Future machines may also use wafer-scale integration techniques, leading to a complete machine-on-a-chip.

INTEL CORPORATION, 3065 Bowers Ave., Santa Clara, CA 95051; Tel. (408) 987-8080
INTEL CORPORATION (U.K.) Ltd., Swindon, United Kingdom; Tel. (0793) 696 000
INTEL JAPAN k.k., Ibaraki-ken; Tel. 029747-8511

Printed in U.S.A./SC5-090/3K/CD/MB
ISC

GOLD HILL™

CONCURRENT COMMON LISP™



intel®

ORDER NUMBER 280274-001

Concurrent Common LISP™

Concurrent Common LISP™ (CCLISP™) is an implementation of Common LISP designed to run on the Intel iPSC™ concurrent computer family. As an emerging industry standard, Common LISP provides the foundation for a broad base of AI development tools and applications. Concurrent Common LISP extends this powerful environment to allow multiple Common LISP processes to cooperate asynchronously and spawn new processes via message passing on Intel's concurrent computer architecture.

CCLISP is a complete development toolkit for the iPSC computer consisting of:

CCLISP resident on each iPSC node with 4.5 MBytes of memory
message passing mechanisms to implement concurrency between nodes
network services to link LISP workstation development, and provide remote evaluation of common LISP forms

CCLISP's Common LISP functionality is an extension of GCLISP 286 Developer™ (GCLISP v2.0), which supports the large memory (15 MByte) addressability of Intel's 80286, the 80287 math co-processor, and a compiler.

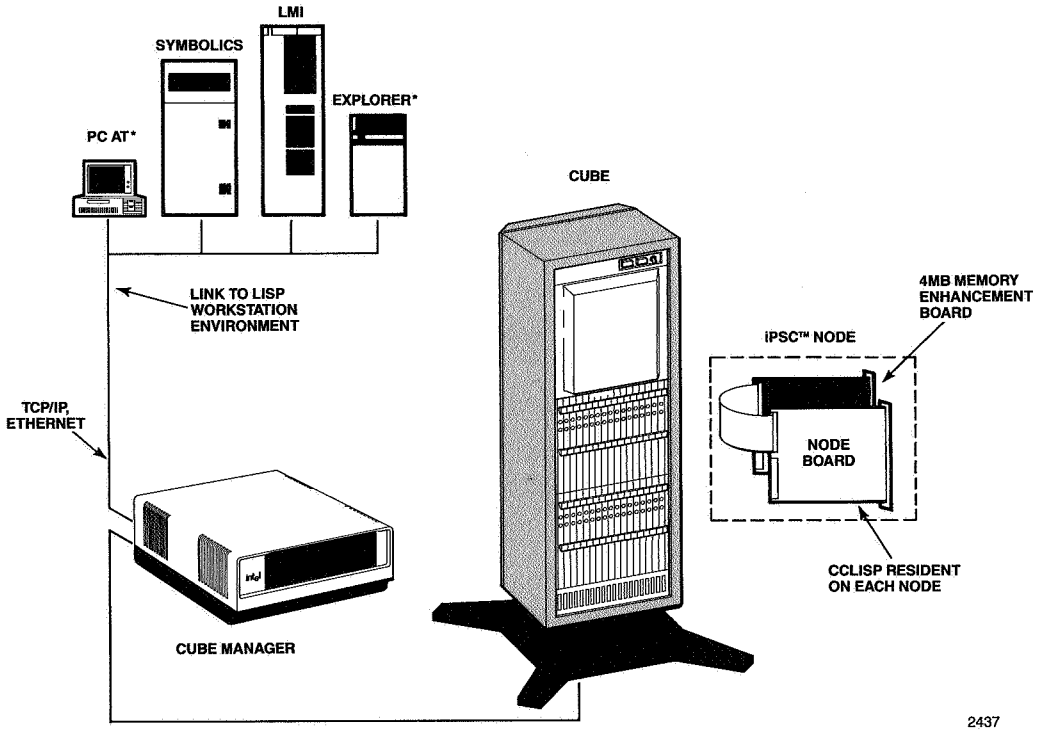


Figure 1—CCLISP on Intel's iPSC concurrent computer

CCLISP v1.0 FEATURES

CCLISP resident on each iPSC node	GCLISP 286 Developer with lexical scoping Steam I/O communication support for message passing Dynamic loading and execution of functions written in C Support for remote evaluation of Common LISP forms by other nodes and LISP workstations Support for 80287 math co-processor Optimized for high performance on 80286 architecture
CCLISP compiler callable by each node	Dynamically callable like a function 8x performance increase over interpreted code Reduction in memory requirements
LISP workstation access to each node	Remote terminal access from Cube Manager, PC AT, or remote LISP machine Ability to load interpreted and compiled programs from Cube Manager, PC AT, or remote LISP machine
LISP workstation debugging capabilities	Access to node CCLISP debugging tools Ability to interrupt, inspect, step, trace, and continue a computation
Network services for nodes	File access on Cube Manager, PC AT, or remote LISP machine Remote evaluation of Common LISP forms on PC AT or remote LISP machines
CCLISP technical support	On-site configuration and start-up assistance Two one-week Gold Hill classes, including CCLISP workshop Current applications consulting available separately from Gold Hill

CCLISP v1.0 — NODE RESIDENT LISP

CCLISP provides a Common LISP environment in the 4.5 MByte memory of each iPSC node. The full complement of Golden Common LISP™ v2.0 data types and primitives are supported.

CCLISP Data Types

Number

Fixnums. These are integers in the range −32768 to 32767, represented using 2's-complement notation (16 bits). Characters are also represented as fixnums.

Single-precision floating-point numbers. Represented using Intel 80287 short real format (32 bits).

Double-precision floating-point numbers. Represented using Intel 80287 long real format (64 bits). The floating-point data types follow the IEEE standard. If the Intel 80287 co-processor is present, it is automatically used for arithmetic operations.

Symbol

A LISP symbol has a *value cell* that stores the symbol's value, a *function cell* that stores the symbol's function, a *property list* that pairs property names to property values, a *package,* and a *print name.*

List

The list data type is the union of the *cons* type and the *null* type. **nil** is the only object of type *null.*

Arrays

CCLISP supports one-dimensional arrays, or vectors, of three types: character vectors (strings), byte-vectors, and general vectors. The displaced arrays and dynamic arrays of Common LISP are not supported, nor are multi-dimensional arrays. The Common LISP feature of *fill pointers* is supported. *Array leaders* (from ZETALISP) are also supported.

Compiled Function

A compiled function is a LISP function implemented in 80286 machine code. Most CCLISP primitives are compiled functions. *Note:* The data type *function* consists of the subtypes *compiled function, lambda-expression, dynamic closure, lexical closure,* and *stack-group.*

Stack Group

A stack group is a functional object that can be used to implement co-routines. It contains two stacks: the control stack and the bindings stack for special variables.

Structures

The **defstruct** facility provides a way to define abstract data types, which may be implemented as named record structures. This facility allows the user to hide the implementation of an object.

CCLISP Facilities

Functions and Binding

An interpreter function is defined by the user with a **defun** form or with a lambda-expression. The lambda-list keywords **&optional, &aux,** and **&rest** are supported. (The lambda-list keyword **&key** is not supported.)

Generalized Variables

CCLISP supports the generalized variables of Common LISP. To change the value of a cell, only an accessor form of that cell need be given to the **setf** macro:

```
(SETF (CAR A) '(A B C) )  ; Equivalent to (RPLACA A'(A B C) ).
```

Users can define their own generalized variables. A new generalized variable is defined by mapping from its access form to an expanded access form composed of access forms already known to **setf**; or else by mapping from its access form and a new-value form to a corresponding update form.

Other supported macros that use generalized variables are **push, pushnew, pop, remf, incf,** and **decf.**

Macros

The Common LISP macro facility enables a user to specify a function which will transform a particular LISP form into another form before evaluation. Then the interpreter evaluates the derived form (the macro-expanded form) rather than the original form. The macro facility enables a user to write LISP code to satisfy both the human reader and the evaluator.

CCLISP supports the macro facility. Destructuring and reader macros are supported.

Packages

CCLISP currently implements the built in packages **list, user, keyword,** and **system**; the global variable ***package***; the package functions **find-package, intern, find-symbol,** etc; and the full symbol qualifier syntax.

Input/Output, Streams, and Pathname

All I/O in CCLISP is done via streams. Seven standard I/O streams are provided as the values of the global variables ***standard-input*, *standard-output*, *error-output*, *query-io*, *debug-io*, *terminal-io*,** and ***trace-output*.**

Users may write their own streams, for example to convert a list to a string, or to direct output to a particular area of the display screen.

Most of the Common LISP sequential I/O functions are provided. CCLISP will read characters, lines, or printed representations of LISP objects from a character input stream, and bytes from a binary input stream.

CCLISP also supports Common LISP pathnames, which are implementation-independent representations of file objects.

Not Supported

Certain implementation-dependent data types are not applicable in the Message-Based Operating System environment. Those data types and associated functions and constants, defined by both Common and LISP and Golden Common LISP, are not supported by CCLISP v1.0.

CCLISP Control-Structure Primitives

CCLISP implements a full complement of control structures. Besides the standard LISP primitives such as **cond, and,** and **or,** the following are provided:

DO	general iteration with parallel binding
DO*	general iteration with sequential binding
DOTIMES	iteration a specified number of times
DOLIST	iteration over a list of items
PROG	"program feature" with parallel binding
PROG*	"program feature" with sequential binding
THROW	for performing non-local exits
CATCH	the target of a THROW
UNWIND-PROTECT	for running clean-up handlers
LET	to establish a binding environment
LET*	the sequential binding form of LET
LABELS	for naming functions locally
BLOCK	for establishing a named code segment with multiple exit points
RETURN-FROM	for exiting from a block
LOOP	for indefinite iteration
PROGV	for multiple binding with sequential evaluation
WHEN	for T/F-based conditional evaluation
UNLESS	the negation of WHEN
IF	for T/F-based evaluation of alternates
CASE	for selector-key-based evaluation

The standard six mapping functions are also provided: **mapcar, maplist, mapc, mapl, mapcan,** and **mapcon.**

C Programming support

C functions can be dynamically loaded and executed from within node CCLISP. C code can originate from the PC AT or the Cube Manager.

The node kernel can support a single LISP environment (CCLISP) and multiple C or FORTRAN processes. Processes written in C or FORTRAN must be dynamically loaded before node CCLISP is loaded.

CCLISP v1.0—MESSAGE PASSING

In addition to the standard Common LISP environment, CCLISP provides extended services that support message passing between iPSC nodes. Two levels of node interface provide the foundation for a concurrent programming environment, all are accessible directly from within node CCLISP by evaluating the appropriate LISP form.

Level 1 Message Passing

Level 1 functions provide a low-level interaction with the resident Message-Based Operating System kernel of the iPSC. These functions are analogous to the functions provided to C and FORTRAN programmers of the iPSC:

SYS:CCLOSE	SYS:MYNODE	SYS:SEND-MSG
SYS:CLOCK	SYS:PROBE	SYS:SENDW
SYS:COPEN	SYS:RECV	SYS:STATUS
SYS:CUBEDIM	SYS:RECVW	SYS:SYSLOG

(See iPSC System Product Summary for more information).

Level 2 Message Passing

Level 2 functions provide a higher-level LISP interface to isolate the programmer from the underlying Level 1 functions.

Level 2 functions support the normal Common LISP stream concepts, and respond to a function call (FUNCALL) interface in a manner analogous to normal file streams within CCLISP.

Any node may create a stream to any other node. Streams may be both written to, and read from. Normally when an input stream is read, only the object that was sent by the sending node is produced. However, streams created by **open-node-stream** will return multiple values when executing input operations. The second value returned is the node identifier of the system which sent the message, with the third value returned as the process identifier of the sender of the message.

Function: **Open-Node-Stream** *nid pid &key :direction :element-type*

Creates a stream object that is capable of communicating with the process pid on the node *nid*. The keyword argument *:element-type* specifies the type of objects transmitted by the stream and may be either string-characters for an ASCII stream or unsigned-byte for a binary stream. The default is string-char. The keyword argument *:direction* specifies the direction of the transfer and may be either *:input, :output,* or *:io.* The default is *:input.* Streams constructed by **open-node-stream** respond to the following keyword arguments:

:READ-CHAR	Returns the next character (byte) from the stream.
:READ-LINE	Returns the next line, as delimited by a #\NEWLINE character, as a string object.
:UNREAD-CHAR	Places the argument character back into the stream.
:READ-CHAR-NOHANG	Returns the next character (byte) available from the stream if one exists, otherwise returns NIL.
:PEEK	Returns the next character (byte) from the stream without removing that element from the stream.
:LISTEN	Returns T if a character (byte) is available on the stream, NIL otherwise.
:FILL-ARRAY	Fills the input argument array with the next n characters (bytes) from the stream where n is the length of array.

:WRITE-CHAR	Outputs the argument char to the stream.
:WRITE-LINE	Outputs the argument string to the stream.
:DUMP-ARRAY	Outputs the contents of the argument array to the stream. Array must be a vector.
:FINISH-OUTPUT	Attempts to insure that all output sent to the stream has reached its destination, and only then returns NIL.
:FLUSH-OUTPUT	Initiates the send of all internally buffered data, but returns NIL without waiting for completion or acknowledgment.
:CLEAR-OUTPUT	Attempts to abort any outstanding output operations in progress.

In addition to the behaviors specified above, streams created by **open-node-stream** are acceptable as optional arguments to the normal CCLISP I/O functions of READ, PRINT, FORMAT...

A second Level 2 function provides a mechanism for rapid transfer of objects in a non-printable format:

Function: **Open-fasl-node-stream** *nid pid &key :direction*

Creates a stream that is used to transfer LISP objects quickly between nodes. The objects are translated into a language that facilitates the reconstruction of the LISP object by the receiving node in an efficient way. Format of objects transferred follows that defined in the CMU Spice Common LISP Document, and the rules of format interpretation are as supported in the FASLOAD function.

Operations supported on streams made by **open-fasl-node-stream** are:

:PEEK	Returns the next character (byte) from the stream without removing the element from the stream.
:LISTEN	Returns T if a character (byte) is available on the stream, NIL otherwise.
:READ-OBJECT	Returns the next CCLISP object available on the stream.
:DUMP OBJECT	Outputs the specified CCLISP object to the stream in a format compatible with the :READ-OBJECT function.
:DUMP-FASL-OPERATOR	Outputs the specified FASL operator to the stream.

CCLISP v1.0—LISP WORKSTATION INTERFACE

As an implementation of Common LISP, CCLISP is upwardly compatible with the Common LISP environments provided on LISP machines. CCLISP provides a variety of networking services that allow the developer to interface the LISP machine environment with the LISP environments on each node of the iPSC computer.

Remote LISP workstation/ Cube Manager access to node CCLISP

Remote terminal access is supported with the ability to load interpreted and compiled programs. Interactive communication with each node CCLISP is supported via window stream capabilities.

LISP workstation node debugging capabilities

Remote terminals have access to the debugging facilities on each individual node via an interactive interface enabling utilization of TRACE, UNTRACE, STEP, BACKTRACE, ERROR, and BREAK.

Also provided is the ability to interrupt, inspect and continue a computation on any node via a keystroke from the terminal from which the iPSC system was initialized.

CCLISP remote support services

File system:

The Cube Manager file system is accessible to each node CCLISP. PC AT and remote LISP workstation random file I/O capabilities are also accessible.

File services supported are normal open, random read/write, and close, as well as, directory services which are appropriate for support of the Common LISP forms:

Directory-List
Probe-File
File-Info
Delete-File
Rename-File
CD

Access to remote file services follow the Common LISP specification of pathname objects.

Access to files via Common LISP file and pathname facilities are also supported. Included is support for the host and device portions of the pathname object.

Evaluation Services:

Evaluation services are supported for the purpose of causing the evaluation of LISP forms in the remote environment. Objects from that evaluation are returned to the node CCLISP that requested the service.

Supported LISP workstations

Symbolics, LMI, and TI LISP workstations and the PC AT running GCLISP 286 Developer are currently supported. TCP/IP Ethernet networking hardware and software are required in the host machines and the Cube Manager.

CCLISP Development Scenario

Operation of the CCLISP development environment is characterized as follows:

Initialization: Local initialization is the same as the standard iPSC computer except that the CCLISP image is loaded.

Remote initialization is done by a remote log-on to XENIX™ on the Cube Manager and then execution of the local initialization procedure.

Editing: Source code editing can be done on the Cube Manager, on a remote LISP workstation, or on a remote PC AT with Gold Hill's GMACS editor.

Compilation: The CCLISP compiler will execute from within any node of the iPSC. In addition, program compilation may be done within any hardware environment currently available from Gold Hill Computers, such as the PC AT.

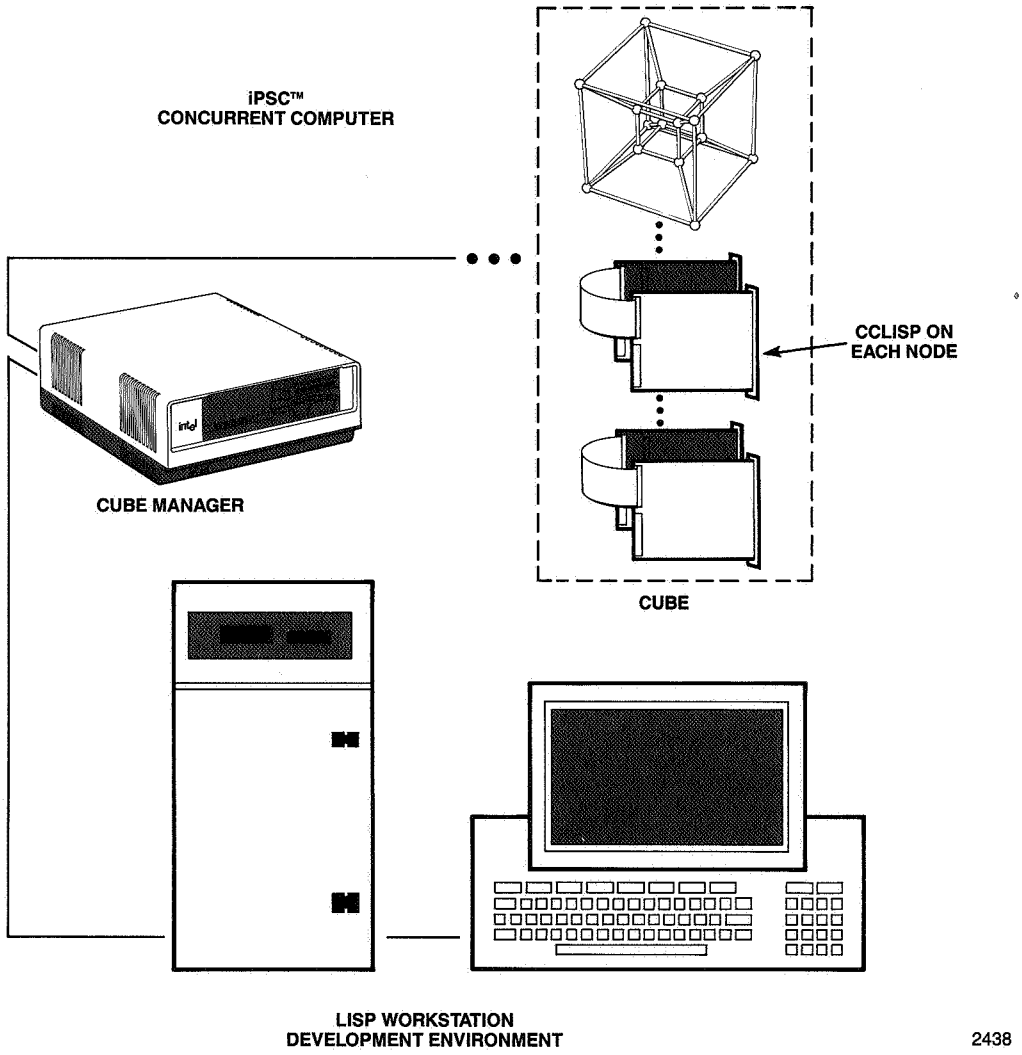


Figure 2—Development Environment

CCLISP v1.0—USER SUPPORT

CCLISP User Training

Refer to CCLISP user training fact sheet for more information.

CCLISP Software Support

Refer to CCLISP software support fact sheet for more information.

CCLISP v1.0—CONFIGURATION REQUIREMENTS

An Intel iPSC-MX/d4, /d5, or /d6 computer is required. This large memory iPSC system uses a 4 MByte memory expansion board to bring total node memory capacity to 4.5 MBytes. An Ethernet TCP/IP network subsystem is also required to interface the iPSC system to a PC AT or LISP machine environment.

(See Intel iPSC System Product Summary for more information).

CCLISP v1.0—REFERENCE DOCUMENTATION

CCLISP V1.0 Technical Summary—available from Gold Hill Computers

iPSC Product Summary: Order number 280101-002—available from Intel Scientific Computers

CCLISP v1.0—ORDERING INFORMATION

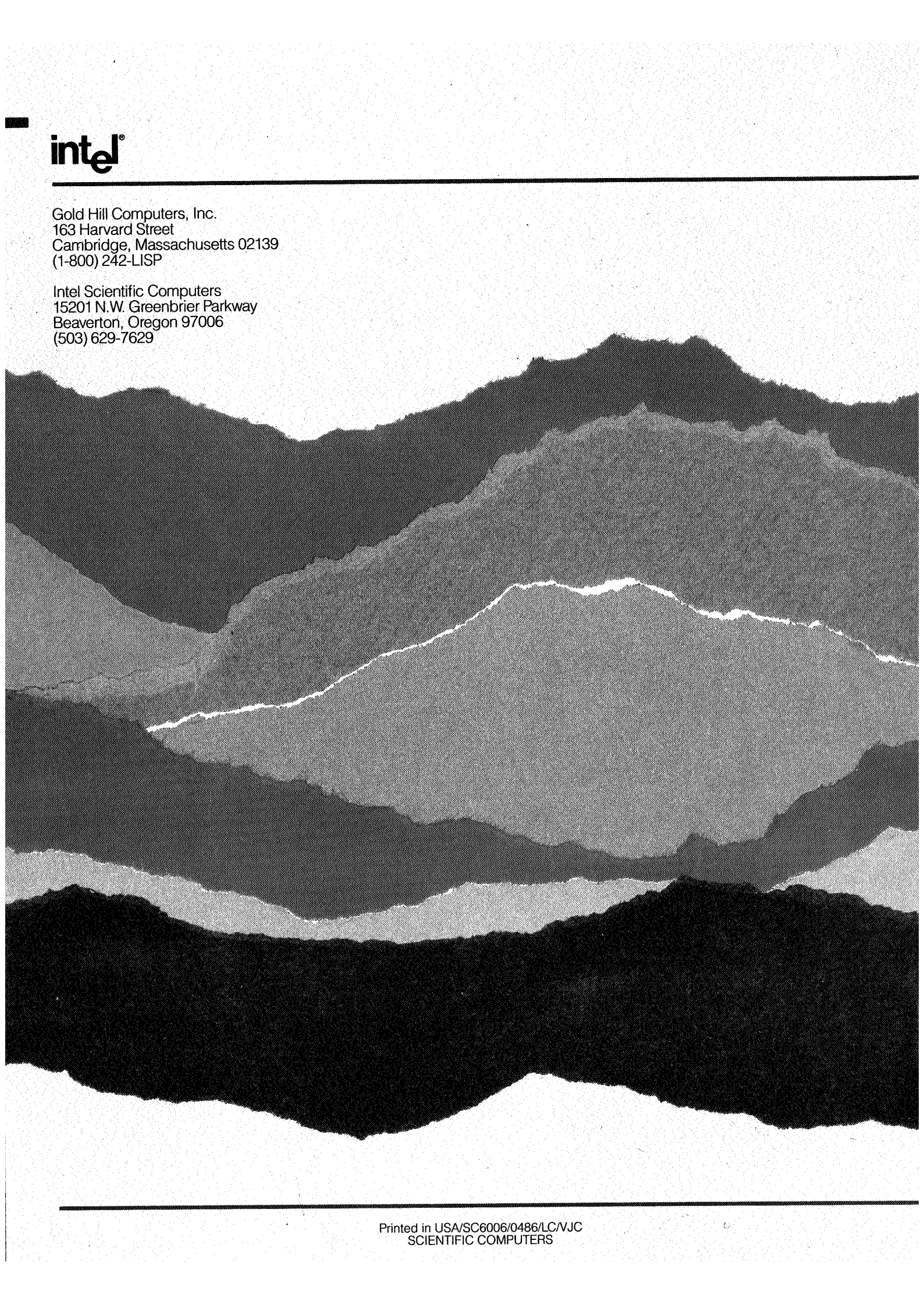
Product Code: CCLISP v1.0

Contact:

Gold Hill Computers, Inc.
163 Harvard Street
Cambridge, Massachusetts 02139
(1-800) 242-LISP

For more information on the Intel iPSC concurrent computer, contact:

Intel Scientific Computers
15201 N.W. Greenbrier Parkway
Beaverton, Oregon 97006
(503) 629-7629



Gold Hill Computers, Inc.
163 Harvard Street
Cambridge, Massachusetts 02139
(1-800) 242-LISP

Intel Scientific Computers
15201 N.W. Greenbrier Parkway
Beaverton, Oregon 97006
(503) 629-7629

Printed in USA/SC6006/0486/LC/VJC
SCIENTIFIC COMPUTERS

intel®

TECHNICAL PAPER

TP-9

Concurrent Processing: A New Direction in Scientific Computing

JUSTIN RATTNER
INTEL CORPORATION

Reprinted by permission from AFIPS Conference Proceedings, Vol. 54.

Order Number: 280265-002

Concurrent processing: A new direction in scientific computing

by JUSTIN RATTNER
Intel Corporation
Beaverton, Oregon

ABSTRACT

Although supercomputer-class performance is a necessity for many research applications, the supercomputer's high price tag places it far beyond the reach of most academic and industrial organizations. In addition, traditional shared-memory architectures, whether single-processor or multiprocessor, are rapidly approaching their performance limits. This paper describes Intel's new iPSC family—multiple-instruction, multiple-data (MIMD) machines implemented by a loosely coupled, distributed-memory, concurrent-processing architecture with a hypercube interconnect topology. The VLSI-based implementation of a concurrent architecture allows these systems to lower supercomputer costs significantly while offering unlimited increases in supercomputer performance. Such a system can provide a focus for parallel-processing research and lead to a greater availability of algorithms and software for a variety of applications.

280265-002

INTRODUCTION: WHY A NEW DIRECTION?

In considering the way scientific computing is done today, two unmet needs become apparent. One is the requirement for a supercomputer that is more affordable by individual academic and industrial research groups. The other is the need for new techniques to take supercomputers beyond the performance limits imposed by today's single-processing architectures.

The Need for Lower-Cost Supercomputing

Supercomputers are an essential tool for research, design, and development. Yet despite their recognized importance, these machines are absent from many university and industry research facilities, where research is hindered by the lack of supercomputing power.

The primary reason for the short supply of supercomputers, outside a few government laboratories, is their cost: Most of the supercomputers available today (from manufacturers such as Cray Research and Control Data Corporation) carry a price tag in the $5 million to $15 million range. Universities and even many commercial users find such prices prohibitively expensive, even though these users recognize that supercomputers are often the most appropriate resource for performing certain complex and important tasks.

In this situation, scientists and others needing supercomputer power have turned to other kinds of computers to help them in their research. One alternative has been the superminicomputer, such as Digital Equipment Corporation's VAX. These systems have proved very popular because they are more affordable than supercomputers. However, their performance is limited—considerably less than 1 million floating-point operations per second (MFLOPS)—and as a result they cannot solve problems quickly. Some problems are actually beyond their capabilities. Thus, researchers frequently find themselves on the horns of a dilemma: They can scale back their problems to the computer's capabilities, in effect solving only part of the problems; or they can solve the entire problems, but at the cost of too much time, frustration, and monopolizing the machine for long periods.

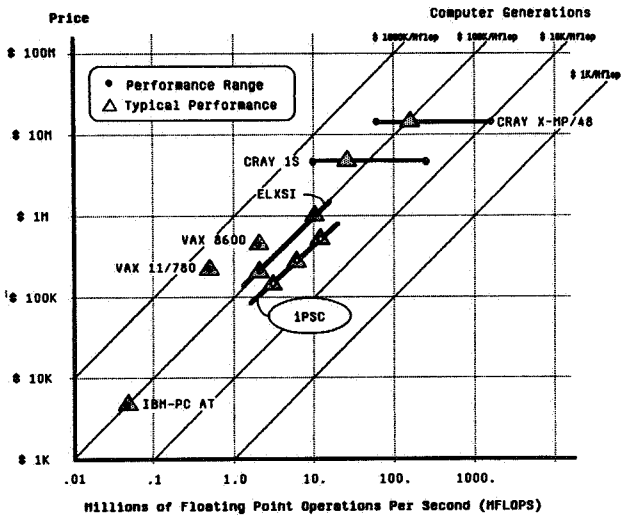
Another alternative for scientific computing has been the array processor, which attaches to a supermini or mainframe and typically delivers performance ranging from 1 to 10 MFLOPS. Array processors have less flexibility than a supermini or mainframe computer, however; and their cost, when added to the amount needed to purchase a host system, puts them out of reach for many researchers.

What is needed is a supercomputer that is affordable and accessible.[1] Currently, researchers need computers with performance of 1 to 100 MFLOPS, but at a cost low enough to justify dedicating the computer to a single project (Figure 1). Such a computer would eliminate the complications and cost of a large shared supercomputer, and it would permit individual researchers or small research groups to own and control the supercomputing resources they need.

The Need for A New Architecture

Soon the high costs of supercomputers will not be the only factor limiting their use. Conventional supercomputer architecture shows clear signs of reaching its theoretical performance limits within the next 5 to 10 years.[2] Already even the large Cray-class supercomputers, with peak performance in the range of 250 to 1,000 MFLOPS, are still inadequate for some types of problems. The demand for performance in some areas today is typically 10 times, and more often 100 times, the projected limit of 3,000 peak MFLOPS for current supercomputer technology.

In handling large-scale scientific and engineering problems that require a large number of calculations, existing supercomputers are essentially vector processors that depend on these data being in the form of either vectors or arrays. Though most scientific and engineering computations are dominated by vectors and arrays, and thus benefit from the



The current research need is for an affordable scientific computer with performance in the range of 1 to 100 MFLOPS. Because of cost, most researchers must force tasks onto affordable machines with insufficient capabilities.

Figure 1—The scientific computational environment (price vs. performance range)

vector performance of the supercomputer, a portion of any code will consist of scalar (single-quantity) operations that supercomputers must also be able to perform. For balanced performance, then, supercomputer architectures must accommodate both fast scalar processing and fast vector processing. A supercomputer handles scalar functions about as efficiently as a typical mainframe.[3]

How effectively a supercomputer handles the combination of vector and scalar operations is the key to its performance. Scalar operations are usually the limiting factor. For example, the peak performance rating of a supercomputer represents the maximum rate at which it can handle a completely vectorized problem. Yet the real performance does not meet peak ratings, because few, if any, problems can be fully vectorized. The parts of a problem that cannot be vectorized must run on the scalar portion of the system, resulting in an increase in overall computation time.

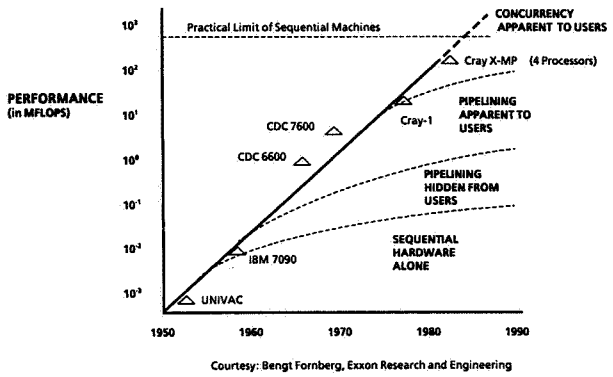
Pipelining is one of the primary methods used in supercomputers to speed performance, particularly vector performance. With the pipelining used in conventional supercomputers, arithmetic operations can be broken down and performed in assembly-line fashion, yielding faster performance. Such pipelining, however, makes programming difficult: Either the programmer must manage the time dependencies in the code, or sophisticated compilers must be used that recognize parts of the code that can be vectorized and handle the pipelining automatically. Ultimately, some hand-tweaking of the code is required to optimize performance.

In spite of the performance increases achieved through pipelining and vectorization, supercomputers are reaching the limits of their capabilities. This occurs because regardless of how sophisticated its design and how fast its components, a single-processor supercomputer eventually reaches limits imposed by fundamental electrical properties—switching speeds and propagation delays. The answer to improving supercomputer performance lies in concurrent architectures. Figure 2 shows the evolution of computer architecture and the performance limits characteristic of each stage in that evolution.

In sum, users with computationally intensive applications need machines with enough potential to meet the projected demand for ever increasing performance. With conventional supercomputer architectures close to their maximum performance levels, architectural breakthroughs are needed to meet the computing needs of these researchers well into the future—at a cost that more research institutions can afford.

CONCURRENT PROCESSING: AN APPROACH THAT MEETS BOTH NEEDS

Concurrent processing represents the most promising long-term approach to achieving affordable, accessible supercomputing.[4] Concurrency is a high-level or global form of parallelism, denoting independent operation of a collection of simultaneous computing activities. A concurrent machine thus uses loosely coupled, multiple, interacting processors to perform many operations at once. Concurrency contrasts with other forms of parallelism, such as pipelining and multiple functional elements. These forms imply some form of lock-



The continued advance of high-performance computing has been fueled by architectural innovation. Large-scale parallelism, the next major step, promises both lower cost and performance beyond the projected limits of today's supercomputers.

Figure 2—Trends in computer performance

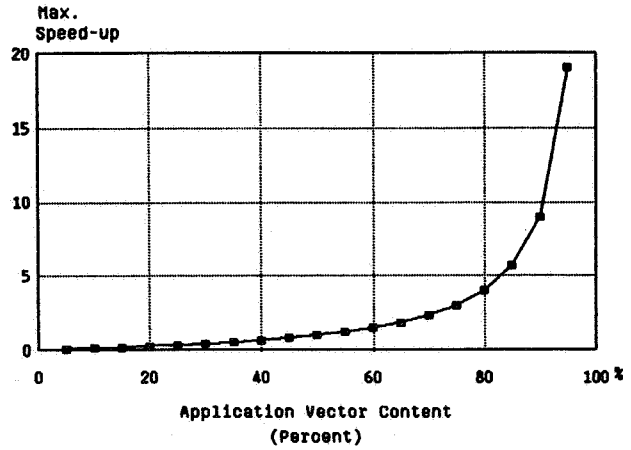
step control, which ultimately limits the expandability and performance of a system. Concurrency allows expansion to a larger number of processors because of the flexibility afforded by distributed memory, distributed control, and loose coupling.

To be solved on a concurrent machine, a problem is broken down into a number of pieces that can run concurrently on more than one processor. Fortunately, this fundamental structure is characteristic of a wide variety of applications, particularly in scientific computing.

Concurrent machines are the next logical step in higher-performance computing because they are able to effectively exploit the parallelism inherent in the physical structure of computational problems. As Charles Seitz suggests, "Concurrency is a fundamental aspect of nature" (personal communication, July 1984). When a computer is used to model a natural phenomenon, as in computational physics or chemistry, concurrency is the underlying operational principle. It is logical, then, that our computational tool should have the same properties that characterize the problem. Simply stated, concurrency makes use of the parallelism inherent in the problem—that is, multiple processes occurring simultaneously. A concurrent machine can, in a sense, become an electrical model of a physical system being studied.

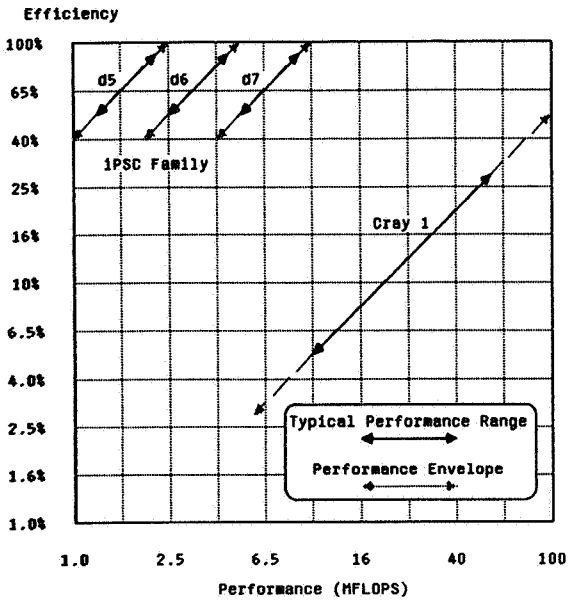
The program to solve a complex mathematical problem is generally separated, conceptually, into a scalar portion (the outer loop or problem space), which mathematically describes the boundary of the problem; and a vector portion or inner loop. On a vector or array processor, the parallelism inherent in the vector portion of a mathematical problem also facilitates high performance.

However, the scalar portion of the problem limits performance, because it cannot be vectorized and must run sequentially. As Figure 3 indicates, infinite vector performance will have little effect on increasing the overall performance of the application beyond this point. It has become "scalar bound." This observation is supported by users who report that the



The vector/scalar ratio of a machine determines the class of problems for which it can be used efficiently. Given a normalized scalar performance of 1.0, the speedup observed by the addition of an infinite-performance vector processor is determined by the vector content of the computational problem. A 75% vector content yields a performance improvement of only 3×, while a 20× increase requires greater than 95% vector content.

Figure 3—Performance speedup vs. application vector content (infinite vector performance)



The low computational efficiencies typical of today's supercomputers rob them of the high performance promised by their peak MFLOPS ratings. Computational efficiency can be expressed as the ratio of actual performance to peak performance. Vector processors typically achieve only a fraction of their peak performance. Concurrent architectures that achieve high computational efficiency offer significant cost/performance benefits.

Figure 4—Computational efficiency vs. performance

efficiency rate for the Cray 1 on common problems is typically between 5 and 20 percent of peak performance.[5]

A concurrent machine, on the other hand, attempts to share the load equally over a large number of processors, each solving a small portion of the overall task and interacting over very high-speed communication channels. Such division of labor results from the natural concurrency of the problem. The task of any given processor in the system is scaled proportionally to the size of the segment being processed; each processor solves a portion of the outer loop and a portion of the inner loop of the problem. Both vector and scalar performance remain proportional, because the architecture mimics the structure of the problem itself.[6] (See Figure 4.)

Furthermore, a concurrent machine can handle nonnumerical problems such as event-driven simulation[7] and artificial intelligence. This capability is not found in conventional supercomputers, whose architectures are optimized for handling vector calculations.

Historical Perspective

Concurrent or parallel architectures are not new. As early as 1945, computer scientist Vannevar Bush proposed parallel architectures.[8] And John von Neumann, whose ideas led to the sequential architecture used in most computers today, preferred the parallel approach.* But because the unreliability and bulkiness of vacuum tubes made a parallel machine impractical, this approach was not implemented.

During the 1950s, programs at IBM, the University of Illinois, and elsewhere were set up to develop parallel architectures for numerical computations. Efforts in the 1960s resulted in the construction of several parallel machines, including the Illiac IV.[9] Projects during the 1970s yielded Burroughs Corporation's Parallel Element Processing Ensemble and Goodyear Aerospace Corporation's Staran.

Parallel processing is exhibited in different ways in today's scientific computers. Array processors use several arithmetic elements (adders, multipliers, arithmetic-logic units) to increase performance. Vector processors combine the multiple arithmetic elements of array processors with pipelining techniques and high-speed electronic components for the hundreds-of-MFLOPS performance claimed for these machines.

Potential Power, Economics

Two approaches are being taken to implementing parallel supercomputers. One connects two or more standard supercomputers together. Cray Research has done this with its Cray X-MP series. But the cost of these supercomputers prohibits connecting more than a few of them. In addition, the use of shared memory in these and similar systems limits the number of processors that can be connected.

* For a history of parallelism, see R. W. Hockney and C. R. Jesshope, *Parallel Computers* (Bristol, U.K.: Hilger, 1981). For a discussion of parallel processing, see Kai Hwang and Fayé A. Briggs, *Computer Architecture and Parallel Processing* (New York: McGraw-Hill, 1984).

A more practical approach uses a large number of today's most economical computational element, the microcomputer. This approach has been made feasible by the performance and cost advances achieved in recent years through very large-scale integration (VLSI). Because VLSI processors are physically compact and widely available, they are 10 to 100 times more cost effective than the semiconductor technology currently used in conventional supercomputers (Figure 5).

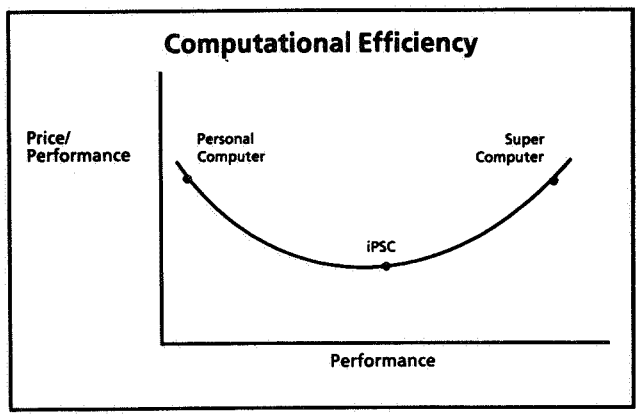
Recently, a number of university research programs have focused on microprocessor-based approaches to large-scale computing. For example, at the California Institute of Technology, Charles Seitz and Geoffrey Fox have developed the hypercube, or "Cosmic Cube," which connects 64 small computers through point-to-point communication channels.[10] Neil Ostlund at the University of Waterloo has built the Waterloop-V2/64, a parallel processor with a 64-node ring architecture.[11] Researchers at Columbia University have produced two concurrent architectures: the DADO machine,[12] which is based on a tree architecture, and the VFPP (Very Fast Parallel Processor), a two-dimensional mesh architecture developed by Norman Christ for computational physics research.[13] At Carnegie-Mellon University, the MMCE (Multi-Micro Computational Engine) is being used for large-scale particle physics computations. The availability of the 8087 floating-point numeric processor and its derivatives has resulted in much of this research activity being focused on Intel microcomputer components and architectures.

Today's supercomputers and array processors are built with high-speed, and sometimes custom-designed, electronic components. Because of limited demand these devices are rarely manufactured in high volumes, and as a result their prices are very high. In contrast, concurrent architectures can be built from standard, off-the-shelf components. Because they are manufactured in large volumes, these components are inexpensive. The result is a dramatic improvement in the ratio of cost to performance.

When the concepts of distributed memory, distributed control, and connected networks are used with concurrent computing architecture, there are few practical limits to the number of processors that can be linked to form a supercomputer system. This is the approach that has been taken with Intel's new iPSC family of high-performance concurrent computers. The iPSC systems employ 32, 64, or 128 processors and have a range of peak performance from 2.5 MFLOPS to 10 MFLOPS, at prices from $150,000 to $520,000.

The effect of concurrent processing on scientific computing will be threefold. First, concurrent processing will increase the computer power available for any problem. Second, because concurrent processors will reduce the cost of large-scale computations, they will broaden the range of applications to which computer-based solutions can be applied. Third, the availability of a "focus" machine will accelerate the development of concurrent applications and facilitate the sharing of concurrent systems technology.

The remainder of this paper describes the iPSC systems in more detail.



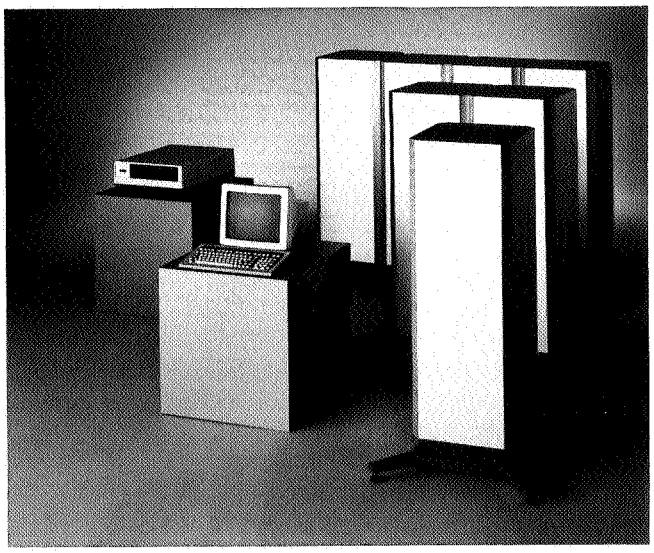
The high commercial VLSI content of the iPSC optimizes the performance for the cost over other computational alternatives.

Figure 5—Computing cost efficiency

THE iPSC CONCURRENT COMPUTER

Intel's iPSC is a family of expandable, concurrent computers designed to provide the research community with systems upon which to develop parallel programming techniques, tools, and application programs. Figure 6 shows the iPSC system.

In the iPSC, concurrency is achieved by having an ensemble of loosely coupled independent processors executing portions of a larger computational problem simultaneously. The basic



The iPSC consists of two major system components: the Cube Manager and the Cube. The XENIX-based Cube Manager supports the programming environment and acts as the local host for the Cube. The Cube is the computational element of the system. There are 32, 64, or 128 processing nodes, with a message-based operating system resident on each node.

Figure 6—The iPSC concurrent computer

iPSC system—the *Cube*—consists of 32, 64, or 128 high-performance microcomputers connected to each other via multiple high-speed, point-to-point communications channels. Further, each processor is connected directly to a local host processor (the *Cube Manager*) via a global communications channel. The Cube Manager performs two major functions: it supports the programming environment and serves as the systems manager for the Cube. The Cube Manager permits the iPSC system to operate either as a stand-alone system or as a computational server when networked to a host environment.

System Topology

The iPSC is based on the hypercube topology or interconnection scheme developed by Seitz and Fox at the California Institute of Technology.[10] Caltech's research was conducted under the co-sponsorship of the Department of Energy and the Defense Department's Defense Advanced Research Projects Agency (DARPA). Intel also supported the research by donating microcomputer and memory components.

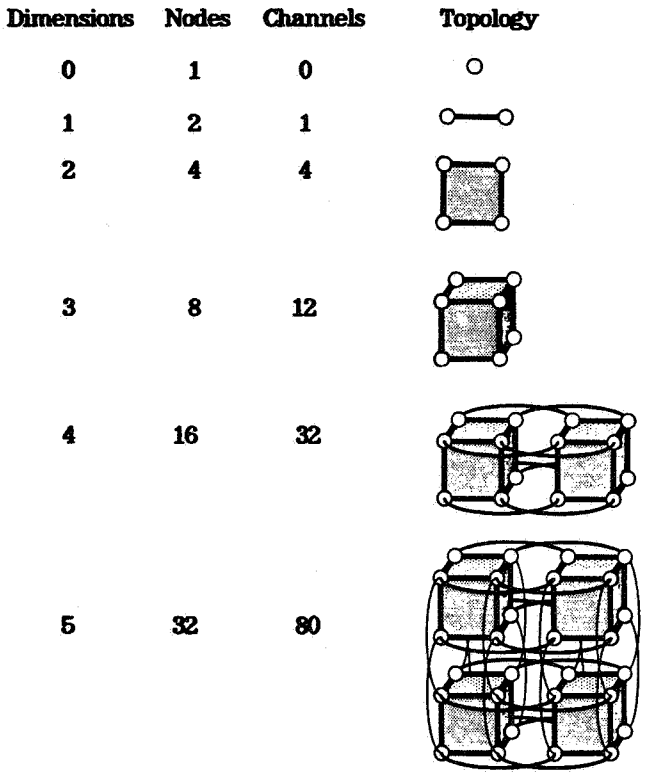
Researchers at Caltech developed the Caltech hypercube from a concept proposed by Sullivan and Brashkow.[14] A 64-node concurrent computer based on the hypercube topology became operational at Caltech in 1983. After licensing the concept from the university, Intel engaged in further development work, leading to performance improvements and the design of the iPSC system.

The hypercube is a binary n-cube, also referred to as a *binary hypercube* or *boolean hypercube.* A three-dimensional hypercube is the familiar cube. Higher-dimensioned cubes are built up from this basic structure, with the "dimension" equal to the power of two corresponding to the number of nodes in the cube. Thus, a 32-node cube is a five-dimensional system (2^5) with each node connected to its five nearest neighbors, and so forth. Figure 7 illustrates the hypercube topology.

The hypercube was chosen for a variety of reasons. First, it offers users an option to expand to larger, more powerful systems as needs increase or VLSI-component technology improves. Other approaches, such as shared memory and buses, are limited in the extent to which they can be expanded. The hypercube implementation of the iPSC can be thought of as an open-ended architecture.

Second, the hypercube offers high communications efficiency and communications capabilities that closely match the needs of real problems. That is, the communications among the system elements are optimized for the kind of interactivity that exists in the problems being solved.

Finally, the hypercube can be adapted to suite the size and performance requirements of the problem. A good deal of the research activity in concurrent architectures has focused on the interconnect structure, looking at options such as ring, tree, 2-D mesh, 3-D mesh, and so on. The interconnect structure of the hypercube topology is robust and provides the flexibility to simulate all of these as research into concurrent processing continues. Being a MIMD organization, the hypercube supports both homogeneous and heterogeneous computations.



The hypercube topology provides a rich communication interconnect structure that can be efficiently swapped to lower-dimension topologies. Nodes are connected by point-to-point, dedicated communication channels, thus avoiding contention. The number of communication channels connecting each node is determined by the dimension of the cube; for example, a six-dimension cube connects to six "nearest" neighbors. Communication latency grows only logarithmically for larger-dimension cubes. Communication bandwidth growth is proportional to the size of the cube.

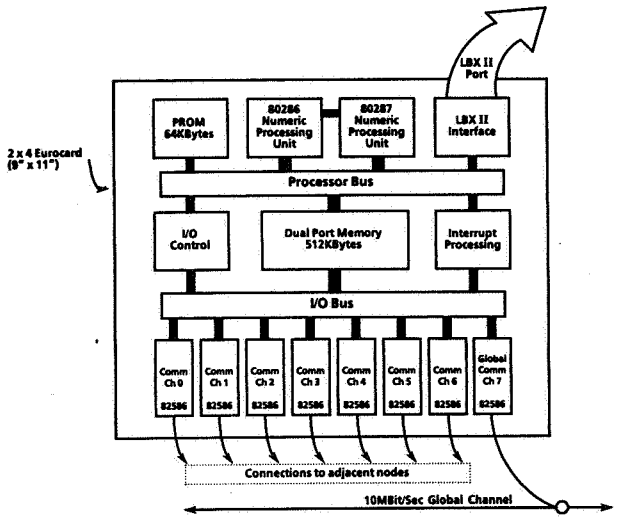
Figure 7—The hypercube topology

System Overview

The initial iPSC family contains three products: the iPSC/d5, a five-dimension machine with 32 computational nodes and 16 megabytes of distributed memory; the iPSC/d6, a six-dimension machine with 64 nodes and 32 megabytes of memory; and the iPSC/d7, a seven-dimension machine with 128 nodes and 64 megabytes of memory. By adding 32 node computational unit users can progressively double the computational power and memory of an iPSC system from 32 to 64 to a maximum of the 128 nodes in the iPSC/d7.

Each microcomputer, or processing node, is made up of an 80286 central processing unit (CPU), an 80287 floating-point numeric processor unit, 512 kilobytes of CHMOS dynamic RAM, and 64 kilobytes of PROM, housed on a single 2 × 4 (approximately 9 × 11 inches) standard Eurocard printed-circuit board. (Figure 8 illustrates the basic node board.)

In addition, each node contains seven point-to-point, bi-directional communications channels and a single global channel. Each channel is controlled by a dedicated communications processor, the 82586 local area network coprocessor.



The importance of VLSI to the design of the iPSC is shown in the more than 11.9 million semiconductor devices contained on each node board. LAN communications coprocessors are used to implement point-to-point communication channels, as well as the Global Channel. The 256K CHMOS DRAMS provide .5 MB of memory per node, and the 80286/80287 CPU/Numeric Processor provides a high-performance computational environment. The iLBX II interface can be used to create two-board nodes.

Figure 8—PSC node board design

These coprocessors move messages between nodes via dedicated bidirectional, point-to-point communications channels with integrated direct memory access to the associated node's RAM.

For enhancement purposes, the local processor-memory bus on each node is accessible via a standard MULTIBUS II iLBX high-speed bus. On the system backplane, the iLBX bus is routed from each even-numbered board slot to the adjacent odd-numbered board slot. The odd-numbered slots may be used for boards that extend the memory or processing capacity of the nodes.

The Cube Manager in the system is an Intel System 310 microcomputer running under the XENIX operating system. It serves as a local host for the Cube and supports program development, applications execution, and diagnostics. It has a 2-megabyte memory, a 40-megabyte Winchester disk drive, and a 320-kilobyte floppy disk drive. In addition to the XENIX operating system, Cube Manager software includes FORTRAN, C, cube control utilities and communications, and complete system diagnostics.

An iSBC 186/51 high-speed communications board provides a global channel to the iPSC system. The iPSC system can also be networked to a mainframe or a supermini-computer by adding a MULTIBUS-based Ethernet TCP/IP interface.

Messaged-Based Interprocess Communications

To coordinate the activities of up to 128 processors, the iPSC uses a message-based operating system resident in each node. This Node Kernel multiplexes the processes that run on that node, provides the system calls that enable processes to send and receive messages, and routes messages as they flow through the hypercube network. The kernel also handles node-to-Cube Manager I/O and process debugging.

Both memory and control are distributed throughout the system. Distributed control means that each node in the system solves its own portion of the larger computational problem semi-independently of other elements in the system. The programmer is responsible for assigning processes to nodes on the basis of the structure of the problem.

The worst-case communication latency for the hypercube is $\log_2 N$; for example, in a 64-node system, the worst case would require a message to pass through six processors. Typically, however, latency is much lower because most applications require only nearest-neighbor interaction. Thus, it is generally necessary only to coordinate the computational results on the edges of each node's data space. For these problems, the larger the computational problem, the better the ratio between computation and communication, and thus the higher the efficiency of the system. The iPSC operating system easily accommodates process-to-process communication, whether the process being communicated with resides in the same node or halfway across the cube.

Low Cost, High Performance

The initial iPSC products will provide a performance range approximately four-tenths that typical of a Cray 1, at no more than one-twentieth the cost. Users report that the Cray 1 typically performs at 10 to 35 MFLOPS on common problems, or at an efficiency rate of about 5 to 20 percent of peak performance. The iPSC typically operates at between 80 and 90 percent efficiency.

The iPSC family is roughly 6 to 24 times as powerful as a DEC VAX-11/780, which costs about $225,000. The price for the iPSC products ranges from $150,000 to $520,000. This reduction in cost—and indeed, the machines themselves—would not have been possible without VLSI technology. As an illustration of the crucial role of VLSI components in the iPSC, consider that each iPSC node board contains 11.9 million silicon devices. Thus, the iPSC/d5, with 32 nodes, contains 380.8 million silicon devices; and the iPSC/d7, with 128 nodes, contains 1.523 billion silicon devices. This level of functionality could not have been achieved in a practical way without the VLSI contribution.

APPLICATIONS

The iPSC is expected to be used in academia, government, and industry. Scientific researchers and computer scientists make up the first two groups. For them, an iPSC system will be the primary vehicle (none is currently available) for research in concurrent computing. The iPSC computer also will be an important research tool for computational physicists and chemists.

The third group of iPSC users will apply the iPSC as a production tool in such applications as circuit simulation and

TABLE I[15]—Examples of problems suitable for concurrent processing

Class of Problems	Examples	Communication Topology
Finite difference equations; finite element equations; partial differential equations	Geophysics, aerodynamics	3D mesh
Statistical	Lattice gauge	4D mesh
	Melting	3D mesh
	Coulomb gas	Ring
Time evolution of 1/r potential	N-body gravity	Ring
Time evolution of general dynamics	Particle motion (sand avalanches)	3D mesh
Fast Fourier transform	Evolution of universe, fluid dynamics	Hypercube
Network simulation	Circuit simulation	Logarithmic graph* such as hypercube
	Neural network	
Isolated	Ray tracing (graphics), data analysis, initial condition study	
Image processing	Analysis of satellite data	Hypercube
Artificial intelligence	Chess	Tree
Event-driven simulation	Industrial, economic, military ("war games")	Logarithmic graph* such as hypercube

* Maximum communication time increases as the log of the number of nodes.

finite-element analysis. The cost effectiveness of the iPSC will stimulate development of applications in the latter group.

Artificial intelligence applications will also benefit from the Intel concurrent processing system. Because concurrent computer architecture is analogous to neurological networks in the human brain (conventional computer architecture is not), computers employing these characteristics may help speed research in this area of artificial intelligence.

Caltech researchers have already put concurrent processing to work in solving a variety of problems in fields such as quantum chromodynamics, structural mechanics, fluid mechanics, high-energy physics, seismology, astrophysics, and computer science. Table I lists several examples of problems that are well suited to concurrent processing.

CONCLUSIONS: A NEW DIRECTION

Research into large-scale computing points to concurrent processing as the most promising technique for increasing supercomputer performance. Concurrent architectures deliver a further benefit in that they can also handle a number of nonnumeric applications that traditional supercomputers cannot accommodate—applications such as artificial intelligence, sorting algorithms, tree searches, and ray tracing.

Nonetheless, significant work remains to be done to develop software for concurrent architectures. Peter Denning, the director of NASA's Institute for Advanced Computer Science, has called attention to this need, pointing out,

> Our research programs are going to have to look at the programming and software problems. We need a lot of work to understand how to make program parts (subroutines) for highly parallel computations and how to plug these parts together to form larger programs. And the programming process will have to include interactive graphic components so programmers can deal with large parallel programs with the aid of pictures. Unless we deal with these issues, we won't be able to take advantage of these machines.[1]

The present state of concurrent computing is similar to the state of real-time control software in the mid-sixties. Before Digital introduced its PDP-8, it was generally felt that such software was very difficult and could be handled only at the largest research centers. Within a few years of the PDP-8's becoming widely available, however, most computer science graduate students knew how to write interrupt handlers for real-time sytems. The widespread availability of a focus machine had dramatically accelerated the accumulation of knowledge by making it easier for researchers to share ideas, exchange programs, and build on each others' work.

I believe that the iPSC family will play a similarly vital role in concurrent computing. The hypercube topology allows expansion to larger systems, fits well with projected directions of VLSI technology, and provides flexibility in the kinds of topologies that can be modeled. The combination of VLSI technology with hypercube topology does indeed provide a new direction for scientific computing: of affordable supercomputers and unlimited increases in performance.

REFERENCES

1. Dallaire, Gene. "American Universities Need Greater Access to Supercomputers." *Communications of the ACM*, 27-4 (1984), pp. 229–298.
2. Seitz, Charles L., and Juri Matisoo. "Engineering Limits on Computer Performance." *Physics Today*, 37-5 (1984), pp. 38–45.
3. Buzbee, B. L. "The Efficiency of Parallel Processing." *Los Alamos Science*, 9, Fall 1983, pp. 71–75.
4. Browne, James C. "Parallel Architectures for Computer Systems." *Physics Today*, 37-5 (May, 1984), pp. 28–35.
5. Davidson, Howard. "Some Predictions on the Performance of Future Supercomputers for Simulation and Control." *IEEE Transactions on Nuclear*

Science, NS-31-1 (1984). Also see Riganati, John P., and Paul B. Schneck. "Supercomputing." *IEEE Computer*, October 1984, pp. 97–112.

6. Charlesworth, Alan E. "An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FP-164 Family." *Computer*, 14: 9 September 1981, pp. 18–27.
7. Jefferson, David. "Implementation of Time Warp on the Caltech Hypercube." *Society for Computer Simulation Distributed Simulation Conference, SCS Simulation Series*, (5)2, 1985.
8. Bush, Vannevar. *Science, The Endless Frontier: A Report to the President.* Washington, D.C.: United States Government Printing Office, 1945.
9. Cocke, J. and Slotnick, D. L. "The Use of Parallelism in Numerical Calculations," IBM Research Memorandum, RC-55, 21, July 1958.
10. Seitz, Charles L. "The Cosmic Cube." *Communications of the ACM.* 28-1 (1985), pp. 22–33.
11. Ostlund, Neil S., and Robert A. Whiteside. "A Machine Architecture for Molecular Dynamics: The Systolic Loop." to be published *Annals of the New York Academy of Science*, 1985.
12. Stolfo, S. J. and Shaw, D. E. "DADO: A Tree-Structured Machine for Production Systems." *AAAI 82.* Carnegie-Mellon University, August 1982. Also see Stolfo, D. P. Miranker and D. E. Shaw, "Architecture and Applications of DADO: A Large-Scale Parallel Computer for Artificial Intelligence." *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, Menlo Park, Calif.: IEEE, 1983.
13. Christ, Norman H. and Terrano, A. E. "A Very Fast Parallel Processor." *IEEE Transactions on Computing*, C-33-4 (1984), pp. 344–350.
14. Sullivan, H., and T. R. Brashkow. "A Large-Scale Homogeneous Machine I & II." *Proceedings of the Fourth Annual Symposium on Computer Architecture.* New York: IEEE, 1977, 105–124.
15. Fox, Geoffrey C. and Otto, Steve W. "Algorithms for Concurrent Processors." *Physics Today*, 37-5 (1984), pp. 50–59.

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95051 (408) 987-8080

INTEL CORPORATION (U.K.) Ltd., Swindon, United Kingdom; Tel. (0793) 696 000

INTEL JAPAN k.k., Ibaraki-ken; Tel. 029747-8511

Printed in U.S.A./0386/3K/FLG/MB
iSC

intel®

ARTICLE
REPRINT

July 1985

Algorithms, Concurrent Processors, and Computer Science Education:

or, "Think Concurrent or Capitulate?"

ELLIOTT I. ORGANICK
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF UTAH

Order Number: 280138-001

Algorithms, Concurrent Processors, and Computer Science Education:

or, "Think Concurrent or Capitulate?"

by
Elliott I. Organick
Department of Computer Science
University of Utah
Salt Lake City, UT 84112

Abstract

Concurrent processor supercomputers are yet another emerging reality to be reckoned with in preparing appropriate curricula for the remainder of the 1980s and beyond. Old problems need to be revisited and re-formulated for solution on concurrent computing systems. Such systems will soon be marketed. Fortunately for us, learning to exploit and teach the use of concurrent computer systems is exciting and rewarding work. At least one new book presenting concurrent algorithms for well known problems, aimed at graduate students and faculty, will be appearing shortly; it and similar books are likely to trigger a new wave of text and course development tailored for undergraduates.

The conclusion is inescapable that concurrent computer systems and their use are another essential and attractive facet of the computer phenomenon, and another intellectual challenge set before us all. Fortunately, "thinking concurrently" is fun!

Key words: concurrent algorithms, concurrent processing, concurrent programming, hypercube architecture

Introduction

There can be no rest for us. Hardly have we finished upgrading our courses to reflect data abstraction, message-based communication, and objects for system modelling and problem solving; hardly have we adjusted to the Ada/Modula reality, to the new and more powerful dialects of Lisp, and to Prolog and other very high level languages; hardly have we adjusted to windows, mouses, menus and spreadsheets; hardly have we come to understand how hardware and logic design systems can be driven by advances in specification and description methodologies, languages, and verification techniques; . . .

The pace of change is swift and leaves many of us breathless. So, what's next, assuming we are up to date with all of the above? My answer today is parallel processing. Actually, the more appropriate term is *concurrent* processing [8]. In case you haven't noticed, concurrent processing is about to leave the laboratory and enter the marketplace. It is apt not only to revolutionize scientific computing, but may enter the world of everyday data processing as well[1]. Concurrent processors embedded in personal supercomputers are yet another emerging reality that educators must come to terms with in preparing appropriate curricula for the remainder of the 1980s and beyond.

Old and now uninteresting algorithms need to be revisited; the problems that gave rise to these algorithms need to be re-formulated in the framework of concurrent computing systems. Such systems will soon become new factors contending for attention in the supercomputer race and in the race to provide ever-more powerful workstations — and so, we must understand principles of their use, their scope of applicability (far wider than at first thought), and the art of "thinking concurrently" in approaching algorithm design for execution on concurrent processing systems. Fortunately for us, learning to exploit and teach the use of concurrent computer systems is pleasant and rewarding work. Soon there will be new textbooks (e.g., [2]) on concurrent algorithms aimed at graduate students and faculty. These will trigger, as did Knuth's "Art of Computer Programming", a new wave of text and course development tailored for undergraduates.

Scope of Concurrent Processing

One can easily be caught off guard, believing that concurrent processing is limited to a small and perhaps uninteresting class of problems. I now perceive otherwise.

[1]In some ways it has, e.g., with distributed database systems.

Certainly the field of scientific computing is poised for new advances by exploiting concurrent processors, especially using systems which at the top level provide parallel concurrency and which at the next-to-top level exploit pipelined concurrency (e.g., vector processing). There is much concurrency being "discovered" in problems originating in physical sciences, engineering, and numerical mathematics [1, 9]. Those working on improving performance of scientific applications are beginning to discover that models on which these computations are based are often most easily represented by regular structures of processes which interact through messages. Moreover, the same process structures can be viewed as regular geometric structures (as with systolic arrays [5]), within which the primary flow of messages is between "adjacent" processes. There is also much concurrency in problems that originate in computer science, such as string searching, sorting, and other problems that fall in the general category of "problem heap problems" [7] (which includes divide and conquer problems).

Architectures and Models of Computation

Design efforts leading to concurrent architectures and models of computation are numerous. Since I understand only two of them [4, 9], and then only moderately, these remarks are confined accordingly. Common to each approach is an ensemble of computers (or "nodes"), each with its own local memory and front-end communication processor (also referred to as message switch). Various node interconnection topologies have been considered, but hypercube connectivity has great appeal for use in a wide class of applications and range of cube degree [6]. Lang refers to this ensemble as a *homogeneous computer.*

A hypercube of degree N (or N-cube) has 2**N nodes with each node having N nearest neighbors. The two computational models of interest here, both of which are supported by hypercube topology, exhibit extremes of binding strategies; from most static to most dynamic.

1. The kernel initially implemented for the Caltech hypercube (called the Cosmic Kernel) supports only static program structures and (at least apparently) fixed workloads. That is, at the system level, processes are statically allocated to nodes. However, for at least some applications, the concurrent algorithms that a user constructs may (implicitly or explicitly) exhibit various degrees of load balancing — through dynamic data distribution.

2. The kernel for the applicative multiprocessing system under development at Utah (Rediflow) will lead, when implemented, to dynamic load balancing and free migration of spawned tasks within the supporting concurrent processing structure — all this without overt management by the application programmer.

The Rediflow approach also supports objects with system-wide unique names, thus permitting addressing across node boundaries. By contrast, each process executing in each node in the Caltech model has its own separate, and hence more limited name (and address space).

While the Rediflow approach has the appeal of greater generality and potentially greater ease of use, favorable performance for it has not yet been confirmed on a prototype. By contrast, the Caltech cube with its Cosmic Kernel[2] exists and works; similar systems are being built and will soon be available for experimentation and use, especially within the scientific computing community. Because there have already been a number of successful computations executed on the Caltech Cube [1], the Cosmic Kernel model has more thoroughly demonstrated its appeal. In short, we need look no further to build the case for concurrent processing and the need to understand where it should fit in computer science education.

Using a Cosmic Kernel-like executive, the simplest computation scenario appropriate for concurrent processing applications is as follows:

1. A problem is formulated (in the abstract) as a graph structure of nodes such that each node holds an identical process (or set of processes) and arcs represent internode (and hence interprocess) communication paths.[3] The (abstract) graph is then embedded in the (concrete graph of the) hypercube. For example, ring, 2-d, or 3-d mesh problem graphs all embed nicely in a cube of degree 6 or higher. Thus a ring of 64 nodes embeds in a 6-cube, but note that only two of the six interconnections from each node are put to active use as communication channels. [The graph-embedding process is surprisingly simple or at least surprisingly straightforward for many problems, especially so when using a well-chosen set of utility routines.]

2. Machine code for an abstract node is loaded into the corresponding concrete node — generated for each separately or independently compiled application process. Note that, in general, the (identical) application code is loaded into each participating node of the hypercube; the host computer serving (driving) the hypercube usually supplies this code.

3. The workspaces for each node are then initialized, usually via interaction with the host.

4. The application in each node is then invoked, thereby starting up to 2**N concurrent (and cooperative) computations. Cooperation is achieved via message exchange. Messages transmitted to distant neighbors are

[2]Both have been designed and implemented by Seitz and his students.

[3]One is free to replace "process" with "data abstraction" or "object" in this sentence.

relatively costly in terms of system resources. For this reason, one tries to design algorithms which exhibit strong locality (i.e., messages are sent primarily to nearest neighbors).

5. Results, intermediate and final, as well as various pieces of status information useful for debugging or performance analysis, are sent from each node to the host.

6. A computation is usually terminated when the host determines that all processes in the cube have terminated or when the host chooses to kill the processes executing in the cube.

The choice of communication protocols by which messages are exchanged affects the simplicity, correctness, reliability, and speedup of a concurrent algorithm. By no means are all the interactions among these attributes yet well understood. Basically, there are two kinds of protocol style one can choose. (Both are available to the user of the Cosmic Kernel):

1. *Fully synchronous message passing.* Here (i) a sender is blocked until a corresponding receive request has been issued, and (ii) a process initiating a receive action waits until a message has been received. While this protocol appears simple to use and safe, it is, contrary to one's intuition, surprisingly easy to accidentally introduce a deadlock condition. For some applications, one may also find that the use of synchronous protocols leads to significantly slower performance.

2. *Nearly asynchronous message passing.* Here the sender is normally not blocked upon issue of a send request and can proceed to compute during the overlap with actual transmissions and receipt of the message. Likewise the would-be receiver can issue a receive request, can proceed with computation that does not depend on the content of the to-be-received message, and can defer polling for actual receipt of a previous request until it is convenient to do so. When appropriate, deferred polling is a useful strategy to minimize waiting for messages and can add to the efficiency of the algorithm when executed on a system for which communication channels to and from the nodes are slow relative to computation speeds at the nodes. I have found that use of asynchronous protocols facilitates easily understood algorithms while eliminating deadlocks due to programmer errors in specifying synchronization for message sending/receiving. (There is still no guarantee, however, against possible system-induced deadlocks which may arise due to message buffer storage limitations).

Two concurrent algorithms with deferred polling illustrated.

Here we briefly sketch "textbook style" concurrent algorithms for two familiar problems. In each case we illustrate the use of asynchronous protocols and introduce deferred polling to show how, if desired, message waiting time can be minimized. Our two examples are:

1. Concurrent solution of LaPlace's equation in two dimensions, $\nabla^2(\phi) = 0$, an "old saw", but definitely worth revisiting, and

2. Matrix multiplication, an even "older saw".

Solution of LaPlace's equation. Figure 1 shows a decomposition of a two-dimensional problem requiring solution of LaPlace's equation. We don't show the boundary conditions, but rather focus on the 16 by 16 subgrid to be dealt with within a single node. Following the explanation in [1] we note "that the finite-difference approximation to LaPlace's equation leads to a matrix inversion that can be solved iteratively. To each internal point (i, j), we successively apply the basic (element update) algorithm

$$\phi_{central} = [\phi_{left} + \phi_{right} + \phi_{up} + \phi_{down}]/4".^4$$

Corresponding subgrid points in each node are updated concurrently, with points on problem boundaries dealt with as special cases. Elements that are internal to a subgrid can be computed immediately. Edge element computations require values of elements residing in adjacent nodes. (In particular, edge calculations require data from the edge of *one* adjacent node, whereas corner calculations require data from *two* adjacent edges of two adjacent nodes.) The algorithm followed for any node, and omitting steps required to test for termination, is given in Figure 2. The speedup of this concurrent LaPlace equation solver over its sequential equivalent is close to 2**N.

Multiplication of two conforming matrices. For simplicity here, let A and B both be n by n in shape, where n is large and divisible by 2**N. The computation is decomposed such that all of the 2**N nodes are interconnected as a one way ring. Each node is initialized with matrix data as follows: The ith node is given the ith section of complete rows of A and the ith section of partial columns of B. Thus, node 1 has rows 1..n/(2**N) of A and parts of all columns of B, namely, the elements in rows 1..n/(2**N). The next node is initialized with the next row section of A and column parts of B. Thus initialized, each node can begin computing terms in the product C matrix for elements of C corresponding to each complete row of A supplied to the node. Upon completion of this stage of the computation, the column parts of B residing in

[4]Actually, slightly more complicated update rules, which lead to much faster rates of convergence, are well known [3]. Moreover, research continues for further improvements, especially in attempts to exploit concurrency.

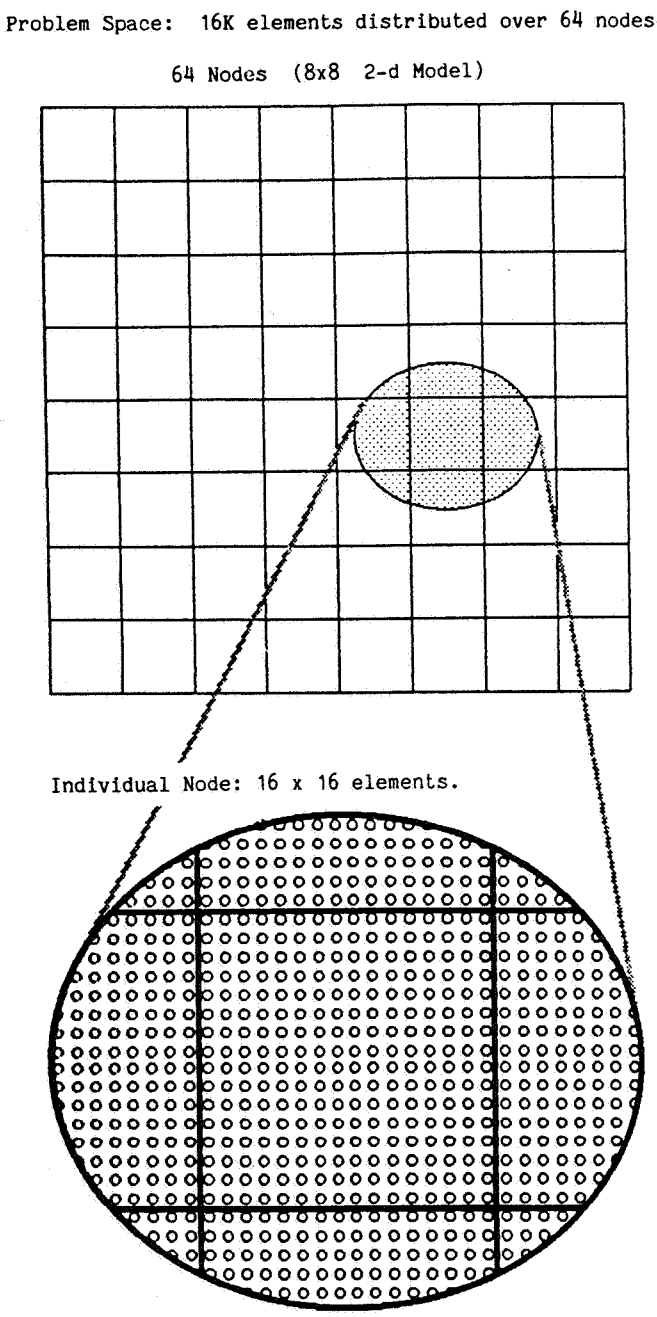


Figure 1: 16K element problem space distributed over 64 nodes.

1. Send row and column edges to adjacent nodes.
2. Receive row and column edges from adjacent nodes.
3. Complete update calculations for each inner element of the subgrid.
4. If an edge message has been received, update element values for that edge.
5. If adjacent edge messages have been received, update the corresponding corner element value.
6. If all elements of the subgrid have not been updated, return to step 4.
7. Start next iteration.

Figure 2: Calculation and communication steps per iteration for zero-wait strategy.

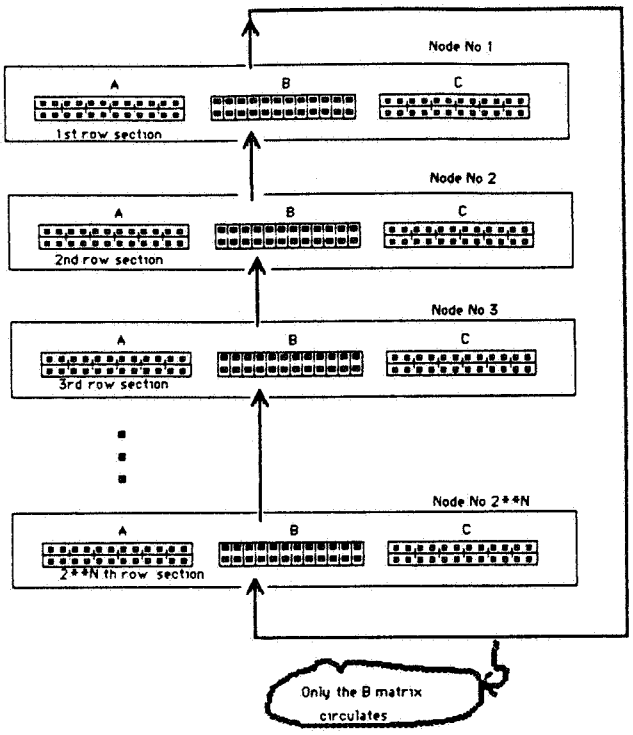


Figure 3: Schematic of the concurrent algorithm for matrix multiplication.

each node are concurrently forwarded around the ring a "distance of" one node. Since the very first stage just completed serves to initialize each sum of terms which eventually become the elements $C_{i,j}$, during each successive stage, new terms are added to existing sums. Figure 3 offers a schematic for this algorithm.

After completing a total of 2**N stages, each of the nodes holds a n/2**N-row section of the product matrix C. Computation is terminated when the row sections of C are reported (sent) to the host, which then assembles C.

In order not to incur delay by waiting for the next subsection of B, the deferred polling strategy can be implemented,

i.e., as soon as a column subsection of B has been used at a node, it can be sent off (i.e., forwarded along the ring) as a message to the successor node and a corresponding column subsection received from the node immediately "upstream" of the node we focus on. Identical asynchronous send requests will be issued concurrently at all nodes (followed by identical receive requests), one send/receive pair per column subsection per node. Waiting time for completion of receives will overlap computation time required by use of column subsections of B to the right of column subsections (of B) that have already been "used up" and are "in flux" along data communication paths from and to each node along the ring. This concurrent matrix multiplication algorithm's speedup should be 2**N over the time required for the sequential matrix multiplication algorithm.

Conclusions

Although parallel processing is a relatively old field, new concurrent architectures exemplified by the Caltech Cube allow us to approach the subject as brand new and fertile. I have sketched, albeit very briefly, the general concepts including those from the viewpoint of an applications programmer. My own experience as a novice concurrent programmer has convinced me that the ideas are all simple or straightforward. Very few if any new programming language constructs are required. The carrot for the computer scientist is that relatively simple but powerful concurrent algorithms will be easy to develop for a growing variety of applications. The reward for the computer science educator is the satisfaction, and indeed fun of discovering how exciting it is to teach concurrent processing.

Acknowledgments

I am indebted to a number of persons who are pioneering the area of concurrent processing architectures, operating systems, and algorithms. The list includes Professors Seitz and Fox at Caltech, Professors Keller and Lindstrom at the University of Utah, and Messrs. J. Rattner, C. B. Moler, P. Wiley, and J. Montague at Intel.

References

[1] G. Fox, S. Otto.
Algorithms for Concurrent Processors.
Physics Today: 50-59, May, 1984.

[2] G. Fox, G. Lyzenga, S. Otto
Solving Problems on Concurrent Processors.
Book in Preparation , 1985.

[3] E. Isaacson, H. B. Keller
Analysis of Numerical Methods.
Wiley, New York, 1966.

[4] R. M. Keller, C. H. Lin.
Simulated Performance of a Reduction-Based Multiprocessor.
Computer 17(7):70-82, July, 1984.

[5] H. T. Kung.
Why Systolic Arrays?
IEEE Computer 15(1):37-46, Jan, 1982.

[6] C. R. Lang, Jr.
The Extension of Object-Oriented Languages to a Homogenous Concurrent Architecture.
Technical Report TR:5014:82, Department of Computer Science, California Institute of Technology, May, 1982.
PhD Dissertation.

[7] P. Moller-Nielsen, J. Staunstrup.
Experiments with a Multiprocessor.
Technical Report DAIMI PB-185, Computer Science Department, Aarhus University, Nov, 1984.

[8] C. Seitz.
Concurrent VLSI Architecture.
IEEE Trans Computers C-33(12):1247-1265, Dec, 1984.

[9] C. Seitz.
The Cosmic Cube.
Communications of the ACM 28(1), Jan, 1985.

INTEL CORPORATION, 3065 Bowers Ave., Santa Clara, CA 95051; Tel. (408) 987-8080

INTEL CORPORATION (U.K.) Ltd., Swindon, United Kingdom; Tel. (0793) 488 388

INTEL JAPAN k.k., Ibaraki-ken; Tel. 029747-8511

Printed in U.S.A./0685/2K/B/MB
iSC

intel®

ARTICLE
REPRINT

AR-420

March 1986

The Science of Computing

Parallel Computation

PETER DENNING
DIRECTOR OF THE RESEARCH INSTITUTE
FOR ADVANCED COMPUTER SCIENCE
NASA AMES RESEARCH CENTER

Reprinted by permission from American Scientist, Vol. 73, August 1985.

Order Number: 280259-001

The Science of Computing

Parallel computation

Peter J. Denning

In February 1985, the Intel Corporation announced the iPSC™, a computer of a new structure, called a hypercube architecture, with the potential to surpass the fastest supercomputers at a fraction of the cost. The machine is based on a project called the Cosmic Cube, headed by Chuck Seitz at Caltech (Seitz 1985). It consists of 2^n processor boards connected by a special high-speed network. (Intel currently offers machines for $n = 5, 6,$ or 7.) The network connects processors as if they were on the corners of a cube in n-space (hence the name, "hypercube"), which means that each processor is directly connected to n others and that the longest path between any two processors spans n links. A 128-processor machine is expected to sustain a processing rate of 10 MFLOPS (million floating-point operations per second) for the fastest algorithms, which is roughly one-fifteenth the sustained rate of the fastest real programs on the Cray X-MP-2X™, at one-twentieth the cost. (The maximal instantaneous speed of the Cray X-MP, in excess of 600 MFLOPS, is not sustained by real programs.)

The hypercube architecture has caught the fancy of many users of large-scale computing. Other manufacturers will soon offer competing products. Yet the concept of a computer containing many processors that simultaneously work on different parts of the same problem is as old as the era of electronic computing.

In the 1920s, Vannevar Bush of MIT demonstrated a general analog computer capable of solving arbitrary differential equations; his computer consisted of many components operating in parallel. The papers of von Neumann in the 1940s considered methods for solving differential equations on a discrete grid—all grid points were updated in parallel, using the differential equation to determine how neighbors affect a particular grid point. Many models of computing substrates for intelligence were based on regular networks of automata. A substantial number of researchers during the 1960s considered a class of models called parallel program schemata; they sought to characterize the behavior of parallel systems and to eliminate certain undesirable behaviors, such as indeterminate computations resulting from unpredictable speeds of components. The ILLIAC IV computer, constructed at the University of Illinois in the late 1960s, consisted of 64 processors operating in lockstep; although its limited memory and expensive hardware prevented its proliferation, it inspired much interesting work in parallel algorithms. The emergence of very large scale integration (VLSI) technology in the 1970s stimulated interest in circuits composed of many parallel computers and in algorithms for using them.

There has thus been a continuing research effort to understand parallel computation. In spite of this, most parallel machines have been laboratory curiosities and most parallel algorithms paper studies. What has held this technology back from general commercialization for so long? Why is an idea that has lain relatively dormant for forty years now getting so much attention?

The answer lies in the strong conceptual simplicity of the sequential stored-program computer, the cost of processor technology, and a plausible argument that "bigger is better."

A sequential computer consists of a processor, a memory, and a communication subsystem. The processor fetches a sequence of instructions from the memory. It decodes each instruction and carries out a specified operation on data in standard registers or in memory locations whose addresses are contained in the instruction. This approach to machine organization appeals to the strong intuitive idea of a step-by-step algorithm. It underlies the most common programming languages and much of the theory of computing. Its simplicity and universality make it a computer model of extremely wide appeal.

Not until the late 1970s was microelectronic technology sophisticated enough that computer architects could seriously consider machines comprising many processors. Prototypes like the ILLIAC IV were based on technology that was unattractive for widespread use. Only recently has it become commercially feasible to build machines capable of comprising hundreds of processors.

Bigger-is-better arguments take a variety of forms. One is Grosch's law, an empirical formulation dating back to the 1940s that says the cost of a computer in given technology is proportional to the square root of its speed. By extrapolation, a computer four times faster than this one would cost only twice as much. So why not seek the fastest possible computer?

Another form of the argument was pointed out to me by Len Kleinrock of UCLA. Suppose we have a supply of jobs requiring an average of X computing operations each, and a processor capable of rate R operations per second. The expected time to complete a job once started is X/R seconds. Next, suppose we replace the single processor with n identical, slower processors each with rate R/n. Let each job be partitioned into a chain of n equal stages, the first stage being completed by the first processor, the second by the second processor, and so on. A total of n jobs can be in various stages of execution in such a pipeline of processors. It is not hard to show that an n-stage pipeline has the same throughput as the single processor but n times longer response time. While a pipeline achieves parallelism, it is less responsive than a single processor of equal capacity.

There are two problems with the bigger-is-better arguments. The more fundamental is that there is a limit to the amount of computing power compressible into one box. The explanation, sometimes called the speed-of-light argument, goes like this. The speed of light is 3×10^8 m/sec in a vacuum, and the signal transmission speed in silicon is at best 3×10^7 m/sec after gate-switching delays are taken into account. A chip slightly over 1 inch in diameter, about 3 cm, can propagate a signal in about 10^{-9} second. Because a nonparallel floating-point chip can perform at most one operation during one signal-propagation, such a chip can support about 10^9 GFLOPS (giga [billion] floating-point operations

Peter Denning is Director of the Research Institute for Advanced Computer Science at the NASA Ames Research Center.

280259-001

per second). For this reason, microelectronics experts do not expect single-processor machines in current technology to significantly exceed 1 GFLOPS. Current supercomputers are within a factor of 10 of this limit.

This limit is becoming a serious problem because of the nonlinear relationship between a problem's size and the computing power required to solve it. For example, matrix multiplication takes about n^3 operations for $n \times n$ matrices. A problem twice as large thus requires a processor 8 times faster to complete in the same time. By the same token, a processor twice as fast can multiply two matrices of size $n \times 2^{1/3}$ in the same time, i.e., handle a problem only about 25% larger.

In other words, linear growth in our appetites to solve problems results in superlinear increases in the computing power required to solve them in the same amount of time. Sooner or later, we will require power beyond the capability of a single-processor machine.

The second reason the bigger-is-better arguments break down is that they are critically dependent on the assumption about job partitioning. In the case mentioned above, jobs are partitioned into sequences of components (pipelines). Consider, however, a more radical approach: each job is broken into n equal and independent components. Assume that the time to partition the input data among the n components and to aggregate the output results from the n components is negligible compared to the time to complete a component. Now each component takes time $(X/n)/(R/n) = X/R$ to complete on one of the slow processors. Because all components can be completed in the same interval, this system has the same response time and throughput as the single processor it replaces. There is a gain if each slow processor costs no more than $1/n$ of the cost of the single processor. (If Grosch's law applied, the total cost of the n processors would be $\sqrt{n}$ larger than the cost of the single processor. But Grosch's law does not apply, because different technologies are used; for example, the fast processors tend to be handmade, whereas cheap, slow processors tend to be mass-produced.) The conclusion is that the parallel decomposition can lead to a machine of the same throughput and response time at a fraction of the cost of a single processor—provided that the workload can be partitioned into independent, approximately equal pieces.

This proviso reveals that just beyond the speed barrier lies another: the software barrier. We're good at understanding sequential algorithms, but we have little experience with algorithms that direct and coordinate parallel processors. We don't know how to program the new machines. Most of the common programming languages are sequential—programs execute one statement at a time. The old computer languages, such as Fortran, Cobol, and Algol, and new languages, such as Pascal, share this property.

An easy way to think about parallel computation is as a collection of separately running sequential programs, called processes, that exchange information among themselves via specific links. This gives rise to a model of parallel computation called communicating sequential processes (CSP), which dates back to 1965. The programming language must be extended to include statements calling for the explicit creation of processes and for communication among them. The language Ada has such statements, as do Concurrent Pascal and a Pascal derivative called Occam.

The CSP model is more difficult to program than is its conceptual cousin, the sequential machine model. The reason is that the programmer must specify not one but many sequential processes and also the communications among them. The hypercube programmer, for example, is faced with the task of writing 2^n programs, one for each processor, and with the challenge of understanding on the order of $(2^n)^2$ potential communication paths.

Understanding such a maze of possible interleaved executions and interactions is a formidable task, not only because of the large number of possibilities but also because of new types of errors that arise from interactions among processes. One type of potential problem is the nondeterminacy arising when two processes contain instructions to write a value in a shared variable: the value in the shared location will depend on which of the two processes wrote last. Another type of problem is the deadlock, where two or more processes get stuck in a cycle, each awaiting the receipt of information from another. Nondeterminacy can be avoided by forcing processes that share memory to proceed in some fixed order. Deadlock can be avoided by forcing processes to request information from one another in some fixed, global order. These techniques and the errors they prevent are unfamiliar to most programmers.

A major research goal in the years ahead is the development of new programming tools that aid the construction of systems of concurrent processes. These tools should operate at a higher level of abstraction than do today's software tools. The effort to specify a correct concurrent computation should be no greater than the effort today to specify a single sequential process.

An example of such a system is the programming environment known as "Poker" being developed in Larry Snyder's CHiP project at the University of Washington (Snyder 1984). Poker presents a graphics screen containing a block diagram of an array of processors. The programmer specifies desired interconnections by drawing links between some of the processors. The programmer specifies programs by entering them in a particular language (e.g., Pascal) and pointing to the boxes of the processors that are to run them. Poker handles the details of loading the programs into the processors, inserting the correct communication statements into the compiled programs, and automatically establishing all the required links. Poker can also be extended to automatically enforce rules that prevent nondeterminacy or deadlock. The effort required to construct a system of parallel programs using Poker is about the same as that required to construct one Pascal program using conventional programming aids.

We are near the limits of single-processor technology. The only available approach to computing power much beyond 1 GFLOPS is machines consisting of many parallel processors coupled with new programming methods that allow partitioning jobs into many independent pieces.

References

Seitz, C. 1985. The cosmic cube. *Commun. ACM*, January.

Snyder, L. 1984. Parallel programming and the Poker programming environment. *IEEE Computer*, July.

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95051 (408) 987-8080

INTEL CORPORATION (U.K.) Ltd., Swindon, United Kingdom; Tel. (0793) 696 000

INTEL JAPAN k.k., Ibaraki-ken; Tel. 029747-8511

Printed in U.S.A./0386/1K/FLG/MB
iSC

intel®

ARTICLE
REPRINT

AR-421

March 1986

The Science of Computing

The Evolution of Parallel Processing

PETER DENNING
DIRECTOR OF THE RESEARCH INSTITUTE
FOR ADVANCED COMPUTER SCIENCE
NASA AMES RESEARCH CENTER

Reprinted by permission from American Scientist, Vol. 73, August 1985.

Order Number: 280260-001

The Science of Computing

The evolution of parallel processing

Peter J. Denning

The software barrier—the set of limitations in software technology that impedes our effective use of parallel hardware technology—is not a brick wall that can be broken down with a heavy ram. It is a deep, thick jungle through which slow progress can be achieved only by constant chopping and hacking. I would like to describe the four stages of our probable journey through this torturous territory. Stage 1 is nearly finished, and Stage 2 is under way. Tentative explorations are beginning on Stage 3. Stage 4 is more distant.

In Stage 1, parallelism is introduced into the hardware of a single computer, which consists of one or more processors, a main storage system, a secondary storage system, and various peripheral devices. Old examples of this include multiple function units in the processor, multiple-bank memory, pipeline execution of instructions, and vector pipelines that apply one operation to a stream of data. These old ideas are undergoing a new round of improvements in a class of machines called reduced instruction set computers (RISCs). RISCs are fast partly because their instruction sets are small and partly because their compilers carefully analyze programs to create code patterns that keep the instruction pipeline busy most of the time.

The computer hardware is controlled by a program called the operating system. Most operating systems contain a ready queue that lists the names of all processes (programs in execution) awaiting their turns to run on a processor. With a few changes, an operating system can be extended so that multiple processors rather than just one serve the ready queue. This principle is exploited by a genre of machines called symmetric multiprocessors. As early as the late 1960s, a few manufacturers offered such machines with up to four processors. Today there are commercial multiprocessors with as many as two dozen processors; the capacities of these machines can be increased simply by plugging in more processor boards. A more radical multiprocessor is the NYU Ultracomputer, an experimental project undertaken jointly by New York University and IBM; new processors and memory can be added indefinitely without saturating the processor-memory interface.

Stage 1 is relatively easy on users because it requires almost no change in the visible software technology. For example, the Cray-1's vector hardware can be exploited by reorganizing algorithms with minimal changes in the Fortran language once the Fortran compiler has been updated to handle vectors of data. Similarly, key subroutines, such as fast Fourier transform, can be recoded for array or vector machines without changing the calling protocols. A multiprocessor can be exploited without changes in the UNIX™ operating system's command language once the UNIX kernel has been updated to allow multiple processors to execute in it simultaneously. Thus, the software changes needed for Stage 1 parallelism can be confined to the compilers, the libraries, and the operating system and are largely invisible to users of high-level languages.

Stage 1 is now well under way and will be mature within a few years. Most program codes include so many assumptions about sequential execution that they, rather than the hardware, limit our ability to use up the power of parallel machines. Our hunger for more computing power will force us to Stage 2.

In Stage 2, parallel execution of cooperating programs on different machines becomes explicit. Programs exchange data over high-speed communication links rather than by passing addresses of shared data segments. The hypercube architecture, such as in the Intel iPSC™, is of this type.

A few changes in programming language are needed in Stage 2. Because there is no common store, processes operating on different machines cannot exchange data by accessing shared variables. Instead they must invoke network commands. An especially simple notation for this was proposed by C. A. R. Hoare of the University of Oxford (Hoare 1978). A new type of variable, called a port, is added to the programming language. At a place where a value is needed from another process, the program contains the port name followed by a question mark. Where a value is to be sent, the program contains the port name preceded by an exclamation point. The sender and receiver must rendezvous at a port before the data are actually transferred between them. For example, a process that reads values X and Y from input ports and transmits the sum to an output port Z would contain the statement "!Z = X? + Y?" In evaluating the expression, the process waits until both ports X and Y contain values, then sums them, then sends the result on port Z. These notations are contained in a derivative of Pascal called Occam.

More extensive changes are taking place in operating systems. Instead of managing one computer, operating systems are being extended to manage networks of computers. The interface will be kept simple so that the same operations will deal with both local and remote resources. These distributed operating systems will eventually allow the construction of computations that span many machines of different types (e.g., workstations and supercomputers).

To help programmers keep track of a potentially large number of interacting programs and machines and to load programs into their machines properly at run time, we will need new software development tools. An example is the Poker programming system described in this column in the July–August issue.

Although not yet widely used, these changes in software technology are well under way. Progress in Stage 2 will be limited more by algorithm technology than by programming technology. How can kernel codes for fluid dynamics, chemical properties, finite structural analysis, seismic modeling, or petroleum exploration be decomposed into parts that can be run on separate machines? Can the numerical properties and the stability of algorithms that solve linear equations or partial differential equations be preserved under such partitions? How extensively will our basic

Peter Denning is Director of the Research Institute for Advanced Computer Science at the NASA Ames Research Center.

280260-001

mathematical and statistical software libraries need to be reworked for partitioned machines? Ahead may lie a substantial effort to exploit new knowledge about basic algorithms in real codes.

I expect a lot of research on algorithms to be carried out during Stage 2. A great deal of detailed information about algorithms for partitioned machines will be developed. But the complexity of what is learned will create a strong pressure to hide the details behind simple interfaces and to find compilers and operating systems that can partition algorithms automatically. This will force us toward Stage 3.

In Stage 3, new languages will make parallelism implicit, and their compilers will take over the burden of partitioning programs. Conventional programming languages are imperative in the sense that their statements are all treated as commands directing a processor. In contrast, the new languages are functional, which means that their statements are treated as expressions denoting compositions of functions. Examples are LISP, which specializes in the manipulation of strings of symbols that denote expression and values; VAL and LUCID, which specialize in dataflow computations, where operations fire as soon as all their operands are available; REDUCE and MACSYMA, which specialize in symbolic algebraic expressions—differentiation, integration, and reduction to minimal terms; SQL, which specializes in queries of relational databases; APL, which specializes in applying functions to vectors of data; and PROLOG, which specializes in evaluating expressions in a deductive logical system. This list of examples is not exhaustive. The common feature of these languages is that they describe the results of a computation but not the method of obtaining the results. This leaves considerable flexibility for compilers and operating systems to distribute pieces of a computation among many processing elements.

To date there is very little experience with these languages outside computer science. There is good reason to believe that they are incomplete with respect to solving real problems in other disciplines. Many problems decompose naturally into pieces called chunks that can be solved separately and must therefore be accounted for explicitly during the formulation of an algorithm. Different chunks may require solution on different machines and in different languages. An example of this arises in computational fluid dynamics, where a test region may be divided into zones, each zone being treated with a different algorithm. Not only is each zone a chunk in its own right, but each zone may be composed of dissimilar subchunks—for example a chunk to derive symbolic equations from the zone's description, a chunk containing a symbolic manipulator to reduce the equations to minimal terms and generate a corresponding Fortran program, and a chunk to execute the Fortran code. The inability of Stage 3 languages to deal with heterogeneous chunks, or to define chunk boundaries, will motivate progress toward Stage 4.

In Stage 4, there will be very high level user interfaces capable of interacting with scientists at the same level of abstraction as scientists do with each other. These interfaces will help formulate precise descriptions of problems in a given domain, using natural language, pictures, speech, and formal notation. The interface system will convert these descriptions into natural chunks, construct functional descriptions of the chunks, convert each functional description to a specific program, convert each program to executable code, request code execution, and finally collect the results for display at the level of abstraction of the domain. It is likely that Stage 4 systems will use today's expert-system technology as a building tool. This technology facilitates storage and use of rules stating the desired response when given stimuli are presented.

There is a striking resemblance between the probable stages of evolution of computing technology for science and the hierarchy of abstractions in the process of formulating computational solutions to scientific problems. From the most general to the most specific level, the hierarchy is as follows:

4. PRECISE DESCRIPTION. Construct a precise description of the model for the problem and the constraints to be observed during solution. This description may use natural language, mathematical notation, and special terminology and notation of the discipline.

3. ABSTRACT ALGORITHMS. Specify the general strategy to be used to solve the problem according to the given description. How will the solution partition into chunks? What strategies of iteration, data exchange, and convergence will be used? (At this level we do not worry about details like data representation or machine capacity.)

2. CONCRETE ALGORITHMS. Construct or connect program modules for the various pieces of the solution. Each module may be represented in a different language.

1. MACHINE CODES. Construct codes in the instruction sets of machines and distributed operating systems to carry out the detailed computations.

The problem-solving process is a series of transformations from the precise description at the level of abstraction of the discipline down to the precise description of an algorithm at the level of abstraction of the hardware.

The four stages of evolution toward parallel computation correspond one-for-one to the four principal levels of abstraction in the problem-solving process. Each stage of evolution is a step in the development of tools for programming at a particular level of the hierarchy. Computer scientists are likely to develop tools from the easiest (compilers at Level 1) to the most difficult (natural language interpreters at Level 4), the reverse of the order in which the tools would be used in solving problems. There are few tools to support transformations at the higher levels of the hierarchy: this is why few Stage 3 or Stage 4 systems exist. Although parallel computation is less visible at the higher levels, it is no less important: it provides the means to operate systems at Stages 3 and 4.

Reference

Hoare, C. A. R. 1978. Communicating sequential processes. *Commun. ACM*, August.

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95051 (408) 987-8080

INTEL CORPORATION (U.K.) Ltd., Swindon, United Kingdom; Tel. (0793) 696 000

INTEL JAPAN k.k., Ibaraki-ken; Tel. 029747-8511

Printed in U.S.A./0386/1K/FLG/MB
iSC