Fermi National Accelerator Laboratory

FERMILAB-TM-1780

The ACPMAPS System
A detailed overview

M. Fischler

Fermi National Accelerator Laboratory
P.O. Box 500, Batavia, Illinois 60510

Operated by Universities Research Association Inc. under Contract No. DE-AC02-76CHO3000 with the United States Department of Energy

The ACPMAPS System
(A Detailed Overview)

Mark Fischler
Fermilab Computer R&D Department

Introduction 2
An Overview of ACPMAPS 3
Architecture Requirements 3
System Overview 5
Processors 5
Communication 6
I/O 7
Software 8
Host Computer 8
Hardware 9
Communications Backbone 11
Processor Modules 16
Distributed I/O 17
The Host 19
ACPMAPS Software 20
Creating and Running Applications 20
How the ACPMAPS System Software Runs Applications 21
Canopy 23
Goals of Canopy 23
General Strategy 24
Concepts 25
Geometrical concepts — details 25
Flow control concepts — details 28
Implementation Strategy 33
Organizational Strategy 33
Implementation of Concepts 34
When is Canopy Applicable 35
Costs Associated With Using Canopy 36
Canopy and CHIP Routines and Data Types 37
Acknowledgement 42

Introduction

This paper describes the ACPMAPS computing system — its purpose, its hardware architecture, how the system is used, and relevant programming paradigms and concepts. Features of the hardware and software will be discussed in some detail, both quantitative and qualitative. This should give some perspective as to the suitability of the ACPMAPS system for various classes of applications, and as to where this system stands in the spectrum of today's supercomputers.

The ACPMAPS project at Fermilab was initiated in 1987 as a collaboration between the Advanced Computer Program (now the Computer R&D department) and the lattice gauge physicists in the Theory department. **ACPMAPS** (pronounced A-C-P-maps) is an acronym for Advanced Computer Program Multiple Array Processor System — this acronym is no longer accurate, but the name has stuck. Although research physics computations were done on ACPMAPS as early as 1989, the full-scale system was commissioned as a reliable physics tool in early 1991. The original ACPMAPS was a 5 Gflop (peak) system. An upgrade by a factor of ten in compute power and memory size, by substituting a new CPU board, will occur during early 1992 — this is referred to as the ACPMAPS Upgrade or 50 GF ACPMAPS. The appellation ACPMAPS II has also been applied to the upgrade; this is somewhat of a misnomer, since only one of five major components was changed.

The idea of the project was to create a system suitable both for production running of lattice gauge codes, and for efficient investigation of new algorithms. These goals are less orthogonal than might be supposed, since to seriously study how a group of proposed algorithms will behave requires nearly as much compute power as a production run using one of those algorithms. This is often beyond the reach of ordinary computing systems, so in order to investigate algorithms, a suitable supercomputer is required.

The ACPMAPS system can be divided into five major subsystems, three hardware and two software. The hardware can be described as a distributed memory MIMD (multiple instruction / multiple data) system, with a flat addressing space and high speed / low latency communications network, augmented by a large distributed disk/tape I/O subsystem. The components are:

- A CPU module containing a processor and a fairly large local memory. The first generation, this was based on the 20 Mflop Weitek XL-8032 chip set, attached to 10 Mbytes of memory. The upgraded CPU module is based on the 80 Mflop Intel i860, containing two processing units, each with 32 Mbytes of memory. The original and upgraded systems comprise 256 and 306 CPU modules respectively — total peak speeds of 5 Gflops and 50 Gflops. (The achieved Gflop rate on actual physics problems usually ranges from 15% - 40% of the peak speed.)
- A communication backbone consisting of 36 crossbar switch crates, with active backplanes implementing 16-way crossbar switches. The

crates are interconnected by differential high-bandwidth ribbon cables, to form a flat addressing space network analogous to a telephone switching network. Any processor (node) can establish a connection to any other node independent of other communications going on. The crossbar switches provide sub-microsecond arbitration and 20 Mbyte/sec transfer rates, so communications overhead is quite low.

- A distributed I/O system, capable of storing and retrieving large data sets in a reasonable time frame. This consists of a 20 Gbyte disk system staging data to 32 helical scan large capacity 8mm tape drives. The communications backbone allows processors to access the I/O devices using the same flat addressing space employed in interprocessor communication. This decouples the arrangement of data on processing nodes from the arrangement of data on storage media.
- A software framework allowing the user to code in terms of the concepts which permeate the physics of the problems being attacked. This framework, called **Canopy**, consists of a library of routines which allow the user to express the nature of the algorithm, without having to worry about details of how the processing will be done in parallel. Canopy improves the efficiency of scientists' algorithm investigation, by eliminating the need for vast expertise in parallel programming techniques. It can easily and beneficially be ported to other platforms including MIMD parallel computers and single-thread computers.
- A set of software tools for hosting the users' jobs, controlling allocation of resources, and so forth. ACPMAPS is a multi-user shared system; a spooler facilitates submission of jobs from multiple Unix computers and initiates each job at a time deemed appropriate in light of resource requirements.

This paper will consists of two sections: First there is a complete overview of the ACPMAPS system, for use by those who wish to learn what the system does and how it behaves. This is followed by detailed descriptions of each of the above components of ACPMAPS — the latter section is intended for ACPMAPS users. Further information can be found in documentation including the Canopy Manual and the hardware manuals for the various ACPMAPS components.

An Overview of ACPMAPS

Design Criteria

The requirements for the hardware of ACPMAPS were driven by the sort of software which was intended to be run. This consists of jobs written in the Canopy framework. Although Canopy concepts will be detailed below, some key requirements should be noted when examining the hardware architecture.

It was not considered acceptable to reject out of hand all algorithms requiring MIMD processing. This means that the system has to have

MIMD architecture. For a given algorithm, the MIMD approach is no more complicated to express than the SIMD approach (if one exists), and is indeed easier to implement in a straightforward way when the geometry of the problem is an imperfect match to the system hardware. Certainly the natural approach to programming currently in vogue is to think in terms of objects and actions — inherently MIMD concepts.

Similarly, it was considered unacceptable to restrict algorithms based on the nature of communications requirements. This has several consequences:

- Any processor must be able to access data from any other processor (otherwise particular patterns of communication will become impossible);
- The communication cannot require synchronous operation across many nodes (otherwise MIMD becomes restricted);
- Any processor may initiate a data access without foreknowledge or preparation on the part of any other (since in general algorithms, the pattern of communication may be data dependant).

We require that the processors in the system be capable of running C programs. C allows for clean address manipulation, which is important in managing MIMD concepts, and has the advantage of being ubiquitous.

A further practical requirement on communications is low latency. The Canopy paradigm encourages the user to express the algorithm in terms in which the granularity naturally matches the problem. In many cases, this leads to frequent medium-length data accesses rather than infrequent grouped data transfers. It is desirable that the price paid for the increased communications frequency be acceptably small; otherwise, users will attempt to re-formulate their algorithms in less natural terms.

There are several workable approaches to achieving these communications requirements. True shared memory, of course, will work, but so will a multitude of explicit communications schemes, permitting distributed memory architectures. These can take the form of directly accessing dual-ported memories or message-passing schemes involving interrupts of the slave processor (the owner of the memory being accessed). The connectivity can be any sort of crossbar, mesh, hypercube or other grid; routing can be done in hardware or software, and may involve intermediate nodes (increasing communications latency). On the other hand, systems which require the explicit cooperation (in user code) of the slave node to permit access to its data, or which switch all communications paths in a synchronous manner, or which only permit communication to some set of "neighboring" processors, do not meet the requirements.

The strategy employed by ACPMAPS involves distributed memory accessible across transparent hardware-routed links. This access is done directly (in the earlier 5 Gflop system), or by messages interrupting the slave processor (in the upgraded 50 Gflop i860 system).

Any system meeting the above MIMD and communications requirements, and which can run C programs on its individual processors, may be considered a candidate for porting the Canopy software. We call a system satisfying the above model a **Canopy platform**. Note that

single thread computers generally satisfy these requirements in a trivial way.

To enhance its utility as a tool for algorithm exploration, a Canopy platform should also have several features. The most important of these is adequate memory space. Large memory sizes are important for four reasons:

- Algorithms that might otherwise be rejected out of hand due to memory considerations can be explored;
- The user need not devote substantial consideration to ideas of which data structures can share space, at least until after the behavior of the algorithm itself is understood;
- It is often beneficial to explore the behavior of an algorithm on a problem size which would be too large for a high statistic production run;
- When a system has too little memory for an application, that is a hard limit. If an application is sufficiently important, slightly insufficient speed is a soft limitation — you may be able to accept lower statistics, or longer running time.

Another important feature is a distributed mass storage I/O subsystem. This should have sufficient disk space, and adequate bandwidth to tapes for storing masses of data. It might be imagined that you ought to be able to run the entire program without any external mass storage device, producing a manageably small final result. This is not the case even for mature production jobs, and it is emphatically not true for algorithm exploration.

A third useful feature for any algorithm exploration platform is multi-user capability. If the system cannot be shared, one of two situations will occur: (a) Users with "small" jobs (too large to run on conventional computers) testing new ideas will have to wait for long production and test jobs to complete; or (b) several small development systems will be needed, and the size and power requirements of these systems must be guessed in advance.

A final useful property in any potential Canopy platform is a host computer running a version of Unix which is reasonably close to POSIX compliant. This makes porting the Canopy hosting tools straightforward, and allows the users to make use of shell scripts and other Unix tools they have been using on a variety of platforms.

System Overview

Here we describe the way the ACPMAPS system implements a Canopy platform, from the point of view of the how the system software sees the architecture. The purpose of each component will be put into perspective here; later, further details are provided.

Processors

The system includes numerous processor nodes. Each **processor node** consists of a commercially available CPU chip (or chip set), with local memory. Each processor runs its program with instruction stream

independent of the other processors — thus the first requirement (MIMD processing) is trivially satisfied. The aggregate of the local memories makes up the entire memory in the system, that is, ACPMAPS is a distributed memory system.

In the 5 Gflop system, the processor nodes were each a single board, using the Weitek XL-8032 chip set, and with 10 Mbytes of memory; such a board is called an **FPAP** (Floating Point Array Processor). In the upgraded system, two processor nodes occupy one physical board. Each processor node has an Intel i860 (80 Mflops peak performance) and 32 Mbytes of memory. The board containing two i860 processor nodes is referred to as a **D860** ("Dual 860") module.

Each processor node runs one process at any given time — the system is shared by assigning a set of nodes to each job. Within the system, there are no restrictions on which (or how many) nodes can be assigned to which job. For a given job, one of the processors is designated the **control node**. The non-distributed portion of the user's job runs on the control node; the distributed tasks are of course run by all the nodes assigned to that job.

Communication

The second requirement, of "flat" access to the entire memory, is implemented by means of a communications system, based on interconnected crossbar switch crates. A processor node can establish a channel to access any other node in the system. (We refer to the processor node initiating communication as the **master**, and the node being accessed as the **slave** for a particular communication.) The switching to establish one channel is performed independently of (and transparently to) any other communications which may be proceeding. Thus the communications system is analogous to a telephone switching network: Any phone can access any other phone, asynchronously with various other connections being established. As with a phone network, the establishment of a communications channel can temporarily be blocked (a "busy signal") either because the target is involved in another communication, or because there is no unused path available to get from one crate to another. A strong point of the ACPMAPS hardware is the very low latency required for switching.

Processor nodes are not the only modules which can be reached via this communications backbone. From the viewpoint of all software, the various modules in ACPMAPS can be classed according to whether they have data which can be accessed in this way. Modules which can be accessed by means of an address in this "flat communications space" are referred to as a **node**. Examples of nodes include processor nodes, memory modules used as I/O buffers, and SCSI controllers for the disk and tape drives. (Occasionally, where there is no danger of ambiguity, the ACPMAPS documentation will use "node" when it more properly means "processor node".) Examples of ACPMAPS components which are <u>not</u> nodes are the crossbar switch crates and interconnect boards which make up the communications backbone itself, and the host computer. The processor nodes, for instance, cannot directly control the behavior of the host or switches by accessing data assigned to those devices.

Every memory location which can be accessed via the communications backbone is assigned a unique **full address**. The full address specifies a **node number** and a local address within that node. The node number specifies information as to how to reach the node (which slot number and path on the backbone to connect to over the backbone). The node number also contains information as to the type of module, and in the case of multiple nodes accessible through the same slot, a field selecting which node to access. Each node in the system is assigned a unique node number. Any arbitrary processor node can access any node in the system, referring to whatever full address is required. The local address portion of a full address can refer to the local memory of a processor node, or a register address or an address on some other bus when accessing other kinds of nodes. The current implementation of ACPMAPS software supports up to 64K nodes and a 32-bit local address space.

(An exception to the rule that any processor node can access any full address in the system is that the FPAP processors in the 5GF system cannot write to their own local instruction memory. This feature provides the means of protecting one user job from being affected by another job accessing its memory space due to communications specifying erroneous full addresses — a jump table specifying which nodes can be accessed is kept as instruction memory. For the upgraded system, the same protection is implemented by utilizing the i860 memory mapping/supervisor mode capabilities. In neither case do we claim the mechanism to be secure against malicious intentional mischief, but the protection against accidental corruption of results is pretty absolute. Intra-job "security" issues are discussed in more detail later.)

Communications can always be viewed as ordered: If a master performs several data accesses to a slave, the order in which the effects of the communications appear to the slave is the same as the order of communications done by the master. In particular, if, say, ten words are transferred, then the last word will be changed last. However, communications involving the transfer of more than one word of data should not be viewed as being atomic. Mechanisms are provided to establish semaphores which are valid independent of hardware used.

The communications hardware is flexible enough to present a wide choice as to how the system will physically be wired. Details of the physical connectivity of the crates composing the ACPMAPS system impact performance, but conceptually are unimportant. A numbering scheme relates the node field of the full address, to particular crates and slots, in a regular manner. Details of the connectivity chosen and the numbering scheme used in ACPMAPS will be presented later.

I/O

The distributed I/O subsystem consists of multiple disks and tape drives (currently 32 WREN VI disks, providing 20 Gbytes of space, and 32 Exabyte tape drives). Logically, the paradigm for the I/O subsystem is that a file resides on a **volume** consisting of one or more disks or tapes — files on the distributed system are identified by a name in the form *volume_name#file_name* and are accessed via Canopy routines such as

read_field, open_field_file, and so forth. When field data (which is distributed over all the nodes in a job) is written to a distributed I/O volume, the Canopy software will guide the sending of some fraction of the data to each disk or tape drive, by designating some processors to gather and route the appropriate data. This process is communications intensive, but the internode communications bandwidth is much greater than the available bandwidth to disk or tape, so the communications costs are negligible.

The disk and tape controllers for this I/O system are accessed as ordinary nodes. There is also I/O buffer memory, accessible both as ordinary nodes and by the controllers. The disk and tape drives themselves are assigned node numbers for the purposes of bookkeeping and resource allocation, but are unlike the usual ACPMAPS nodes in that they do not have memory — it is meaningless to try to access them directly by specifying some appropriate full address.

Software

The ACPMAPS software will be discussed in detail later. It can be divided into two broad pieces: The Canopy software which is used to create applications to run on ACPMAPS, and various software tools to guide the running of applications. Canopy is a library of routines to be linked with user code; the tools are executables and shell scripts to handle building, scheduling and servicing applications on the shared system.

Host Computer

The ACPMAPS system utilizes a Unix **host** computer to provide access to the outside networked world, start up jobs on the system, service Unix calls made by programs running on the processor nodes, and provide allocation and debugging capability. Various software tools are provided to accomplish these functions — three important ones are the **Canopy hosting tool**, the **spooler**, and the **db** tool.

The spooler is the program controls the system. A single instance of the spooler runs on one host computer for the entire system. Its primary purpose is to schedule jobs, assign processor nodes and I/O resources, and cause each submitted user job to start up at an appropriate time. Secondary functions of the spooler include acting as a central point for control of distributed I/O operations, and keeping logs of system usage and exceptional conditions.

The Canopy hosting tool coordinates with the spooler to run a user job. The user runs one instance of the hosting tool for each job submitted. This tool initiates downloading of the user programs to the processor nodes, and sends commands to the processors to commence running the user code. While the job is running, the hosting tool will service UNIX calls made by the control node, and periodically check all nodes to make sure nothing catastrophic has occurred. Since UNIX calls are serviced by the host, a Canopy job running on ACPMAPS can freely access files on the normal UNIX file system.

Although only one copy of the spooler should be running on the system at any time, multiple copies of the hosting tool may be running. The system

can be have multiple host computers attached at any given time, and hosting tools may run on any of the hosting computers. Currently two varieties of host computers are supported: An SGI 4D25, and an Ultrix Microvax.

The db tool is a low-level processor node debugging aid. It allows the user to allocate sets of processors, to access the memories of processors, and to issue various commands such as reset, run program, suspend and resume. Although in principle there is never a need to allocate a particular set of nodes (as opposed to some number of nodes), the db tool allows one to do so — this can be useful for troubleshooting and for evaluation of the impact of communications topology on performance.

The host computer also can run the various compilers and shell scripts necessary to create an executable for a Canopy job. (Actually, this is just a convenience and not an essential feature of a Canopy platform.) Thus the user logged into the system host can cross-compile his code (using UNIX make tools if desired), submit jobs to run on ACPMAPS via the canopy command (which interacts with the spooler), create UNIX input and result files, and control the execution of jobs via UNIX shell scripts, all on the same host computer.

Hardware

The ACPMAPS system consists of processor boards, communications modules, modules to interface with SCSI I/O devices, the I/O devices themselves, host computers, and modules to interface the hosts to the system. This section will provide an overview of the nature of these components and the interrelations among them.

Physically, the full scale system appears as a collection of a dozen **racks** (arranged as four planes of three racks each). Each rack holds 3 crates of modules and up to 8 disk and tape drives. The racks are about 2 feet wide by 3 feet deep, and are six feet tall. The system comprises roughly 600 modules, each of which is a card occupying one slot in one of the crates. Most of the modules are processor boards or communication interface boards. The layout of the system is as follows:

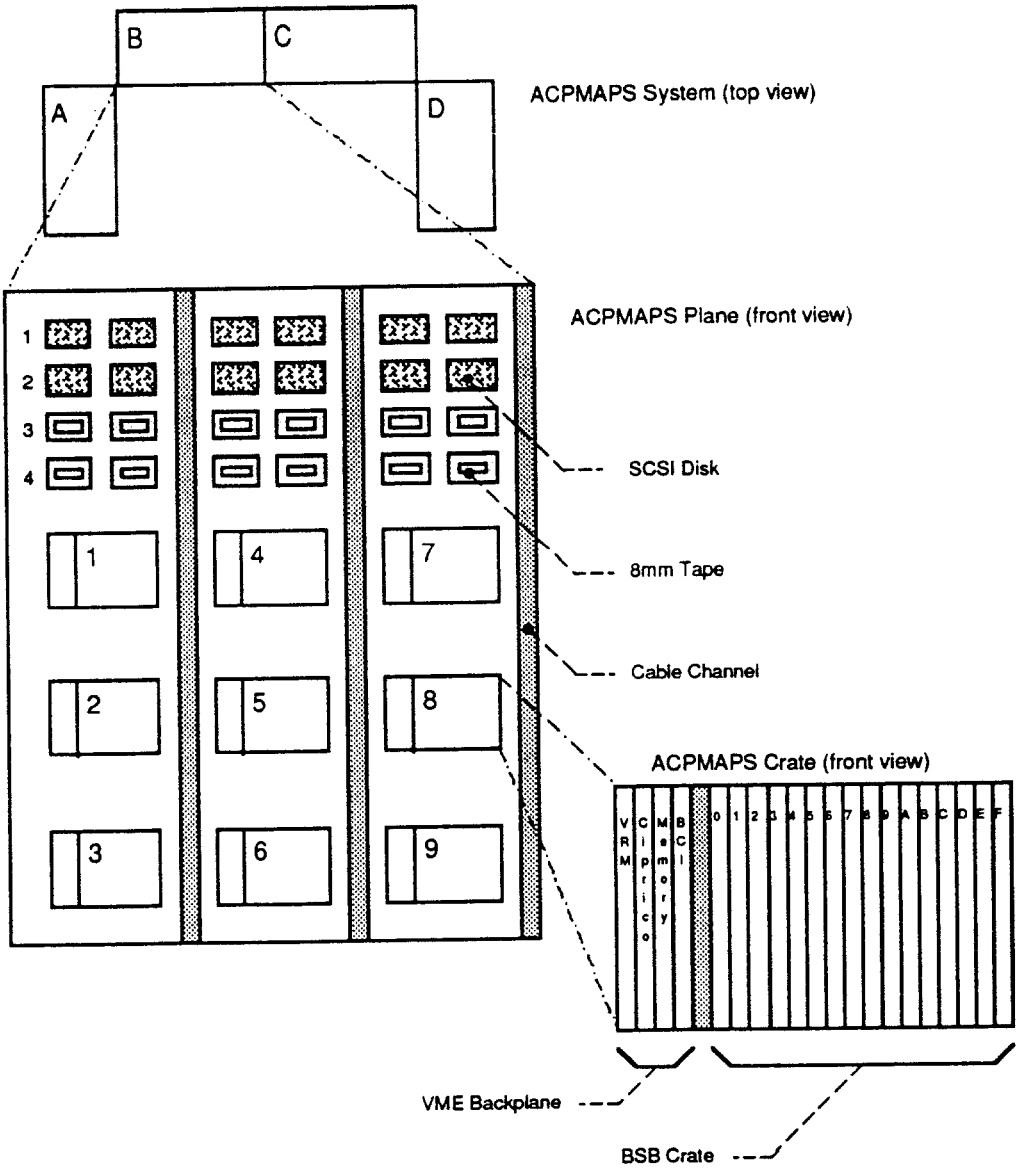

Figure 1: Layout of the ACPMAPS system

Communications Backbone

The ACPMAPS system can logically be viewed as a collection of processor nodes inhabiting a communications backbone. The key element of this backbone is the **BSB** crate (Bus Switch Backplane). This is a crate with 16 slots, each of which can be occupied by a module (card) — analogous to a VME or Multibus II crate. The BSB, however, has an active backplane, consisting of:

- Sufficient 16-way crossbar switch chips to implement full crossbar of a 50-bit data and control paths between the 16 slots.
- Logic to control the switching of the configurations of the crossbars chips, and to arbitrate among requesting masters. The maximum time required to arbitrate and reconfigure the crossbar switches is 700 nsec. If a path from one node to another requires multiple hops (traverses more than one switch crate), then each crate must arbitrate — 700 nsec per hop.
- A PROM to supply routing information — this will allow any node to transparently access the memory of any other node in the system. Each BSB crate configures itself independently of the others, in response to addresses presented by masters in its slots.

The BSB slots can each be occupied by a module containing one or more processor nodes, or by a **BSIB** (Bus Switch Interface Board) card. This card is similar in appearance to the processor cards, except the BSIB has a pair of ribbon cables attached at the front panel. The BSIB allows the establishment of a communications **channel** going across a BSB backplane and out the BSIB onto **BranchBus**. BranchBus is a 50-bit wide bus (control signals plus a 32-bit data path) implemented in differential RS485 as two ribbon cables, each carrying 25 differential pairs. It has multi-master capabilities and a particularly simple bus protocol — the protocol along the BSB backplane matches that of BranchBus.

The bandwidth of data through each channel on the BSB backplane is 20 Mbytes per second. The BranchBus data bandwidth matches this. The BSIB module re-synchronizes data and control signals, acting as both an interface and a repeater; thus a data channel can be established across multiple BSB switch backplanes and multiple BranchBus cables with a reliable bandwidth of 20 Mbytes per second. The addressing information needed to control the reconfiguration of switches to establish the channel is propagated along the same path as the control and data signals. This is illustrated below, for a situation where one CPU has established a channel to another CPU which goes through on intermediate BSB switch crate and across to separate cables. Note that communications across other channels can be proceeding at the same time.

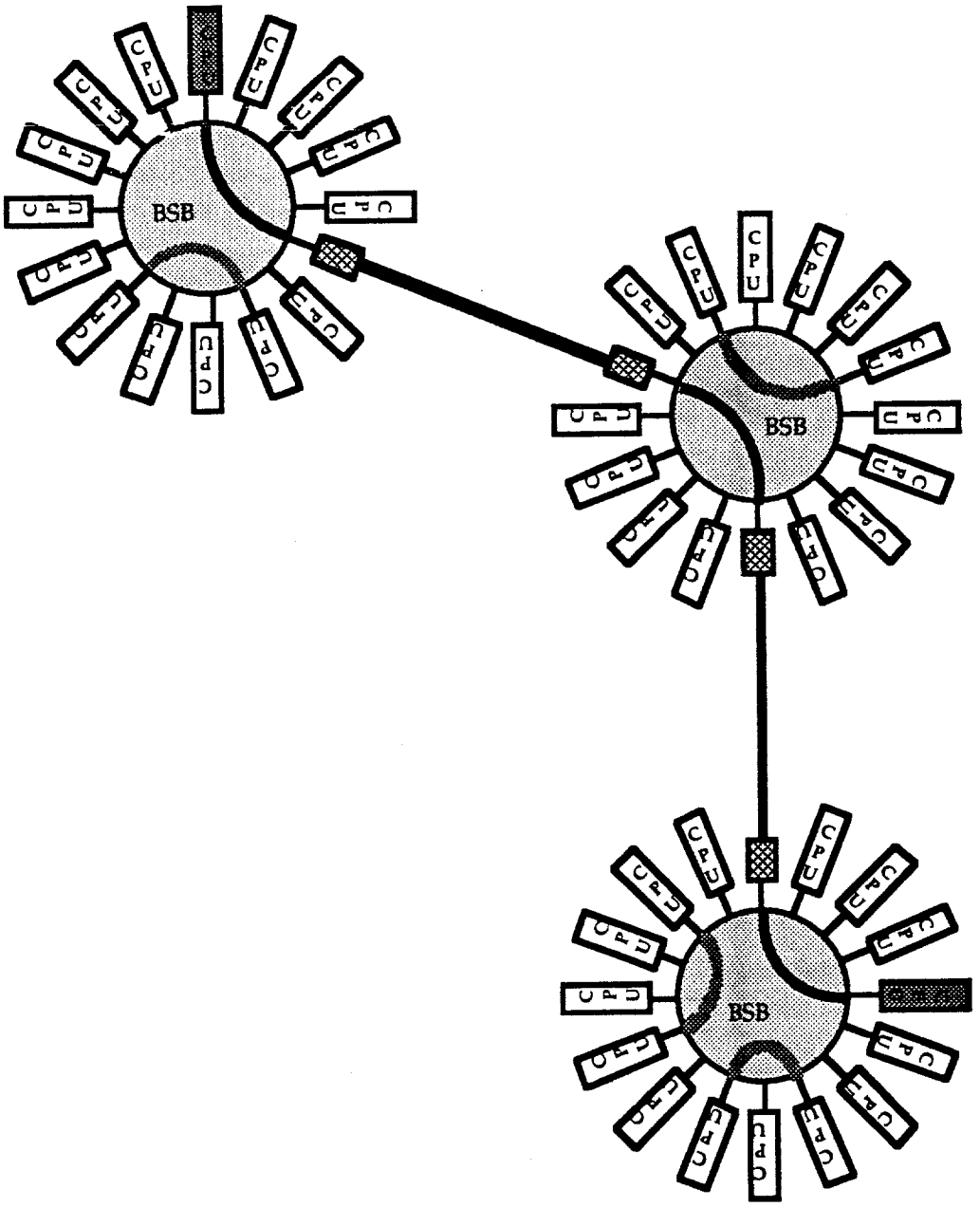

Figure 2: An illustration of inter-CPU communication

Note that the communications hardware does not require every crate to have a direct connection to every other crate. A given pair of crates can have no direct connection, or be connected by one or more BranchBus cables. Typically, a system would be configured with some selected topology, in which certain crates are connected by one bus — the intercrate bandwidth of 20 Mbytes/second is shared by all the nodes in those crates. Thus depending on the nature of the problem being solved, there is potential for a bottleneck in intercrate bandwidth. Changing the topology of the system would be a matter of physically re-cabling the modules and changing the routing PROMs — in general, using more BSB crates and

BSIB modules per processor module would reduce the impact of intercrate bandwidth bottlenecks.

The Communication Topology

The topology selected for ACPMAPS is that of a 3x3x2x2 hypercube of crates, augmented by connections along all the diagonals in the 3x3 and 2x2 sectors. For typical problems, the impact of bandwidth bottlenecks in this configuration is acceptably small (up to 15% for the upgraded system).

As shown in figure 1 above, the ACPMAPS system comprise four "planes" (labeled A, B, C, and D). Each plane consists of 9 crates, connected (using BSIB modules in four slots) to form a 3x3 grid, with all diagonals. Within the plane, each BranchBus cable connects three crates ("three on a bus"). This connectivity is depicted below:

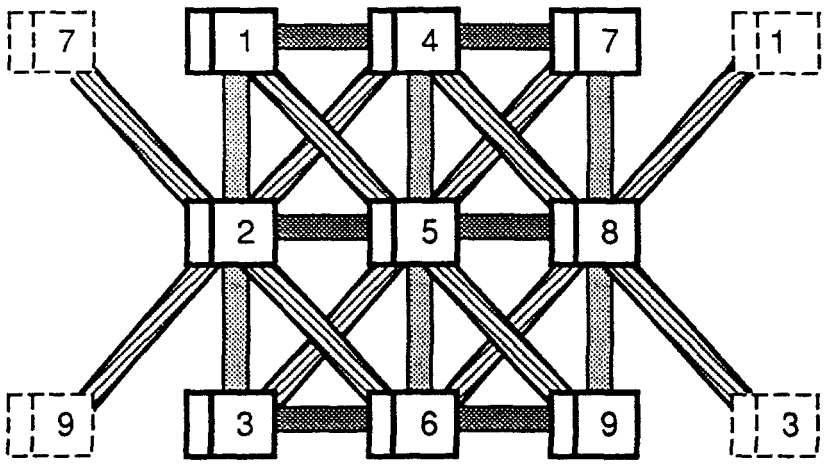

Figure 3: Connections between crates in one plane of ACPMAPS

There are BSIB modules in three more slots of each crate, connecting it with the corresponding crate in each of the other planes to form a tetrahedron. (A tetrahedron is the same as a 2x2 square augmented by both the diagonals — each crate is connected by a Branchbus cable to each of the three others.) The interplane connections can thus be viewed as nine tetrahedrons of crates:

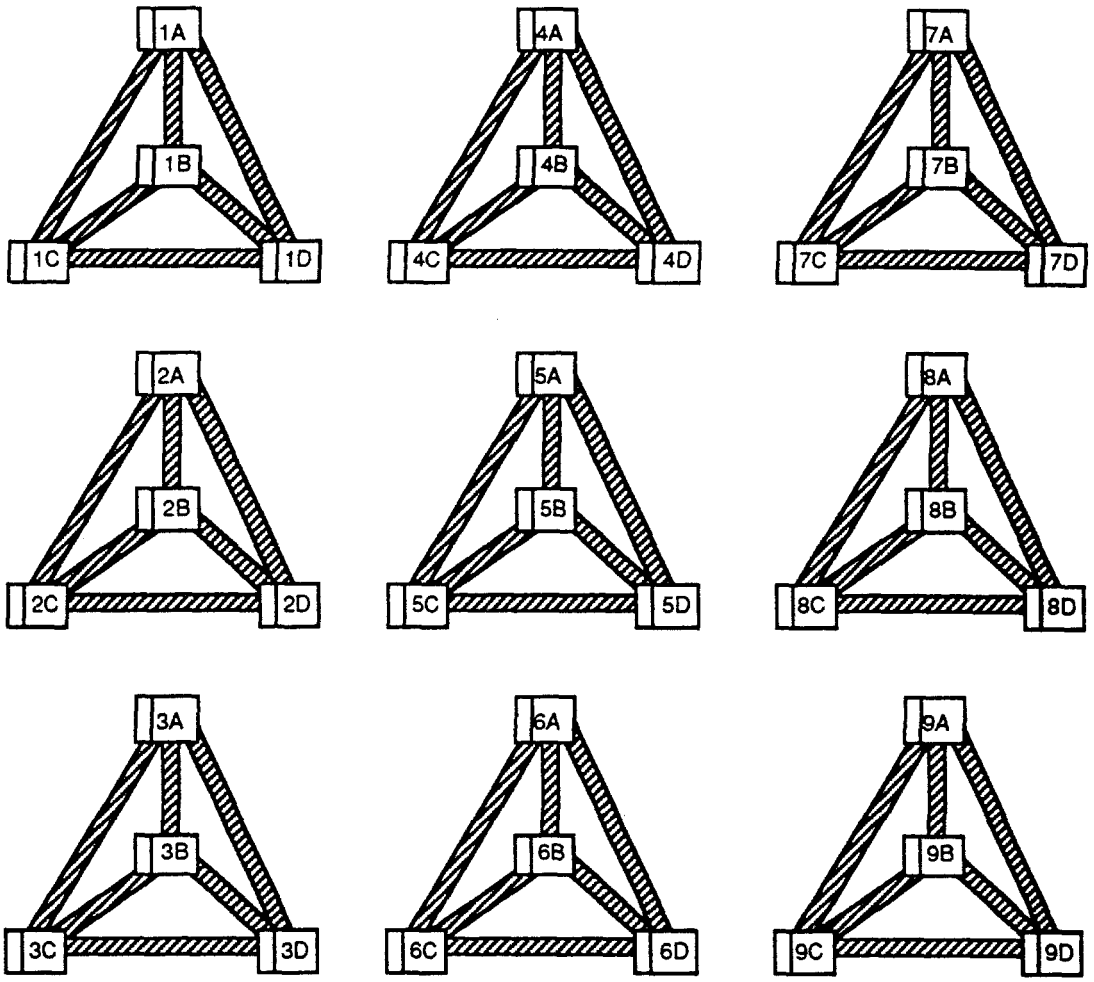

Figure 4: The interplane connections in ACPMAPS

For applications with amorphous communications requirements, this topology has slight advantages compared to a strict hypercube of, say, 32 crates. For example, in such a hypercube, the average path to a randomly chosen crate occupies 2.5 busses out of 80 — exactly the "fair share" of the 80 busses in the system. In the ACPMAPS topology, the average path occupies 1.75 busses out of 102 — only 62% of the fair share. Of course, the hypercube requires only five BSIB slots per crate, rather than seven; even taking that into account, communication is 15% more costly in a hypercube. Another advantage is that in the ACPMAPS configuration, no crate is more than two hops from any other. Of course, this configuration requires "three on a bus" cables, while the hypercube would not.

The Communication Mechanism

The processor boards each contain one or more independent processor nodes (CPU and memory). Here, we will describe the D860 module in the upgraded system, which contains two nodes, each based on an i860 CPU and 32 Mbytes of memory. The two processor nodes occupying a D860 module (a node and its **partner node**) share one interface to the BSB backplane. This interface consists of a local bus to move data from the individual nodes to the BSB interface, a large reprogrammable gate array chip to support the BranchBus protocol, and a set of FIFO chips to buffer up to 8K of incoming or outgoing data. When a node acts as a master to access the memory of another processor (the "slave"), the sequence of events will be:

- The master arbitrates for its local bus (this arbitration logic is done by the gate array chip, referred to as the **BIC** — Bus Interface Chip);
- The BIC performs the protocol to request a channel to be opened on the BSB;
- The BSB makes a connection between the master's slot and the appropriate destination slot (based on the address supplied for the communication; this is looked up in the BSB's PROM);
- The communications might then have arrived at the target node, or might be routed through a BSIB to another switch crate for further routing — ultimately, to the target slave node;
- The slave node responds to the the communication, and *either* provides the data required or accepts the data sent.

The D860 slave node will be interrupted when the communications channel is established. This is necessary because the i860 processor depends heavily on internal cache memory for efficient performance. Since no "cache snooping" mechanism is available to invalidate cache lines for memory that has changed (or to force the processor to write the current value to main memory if it has been updated), the slave CPU must participate in each communication, reading from (or writing to) its own local memory. The cost of this participation by the slave CPU (in terms of additional latency on data access) is partly offset by several advantages in having a powerful processor directly involved in the communication. For example, longitudinal parity checksumming can be implemented to protect against undetected loss of data in a packet, and special-purpose transfers such as semaphores can be provided.

When the slave node is an FPAP (in the 5 GF system), the communications mechanism is somewhat different, in that the slave processor is not interrupted and does not participate in the data access. The FPAP memory is dual ported, and can be read and written directly across the bus, slowing the slave processor only to the extent that memory cycles have been stolen. The i860 memory is single ported, accessible via the processor only — all communications are under program control.

On the D860, communications of up to the length transferred in a single block are truly and inherently atomic — the slave processor is interrupted and does no further user processing until the transfer is complete. Further, the system software will retain the open channel between blocks of

a long communication, so transfers of any length are always atomic. When the slave is an FPAP, transfers are not atomic: The slave can conceivably examine the first word of data sent, and take action based on that, before the last word of the transfer has arrived. The ACPMAPS software is written assuming the worst (non-atomic) case, and thus does not depend on the atomic nature of D860 communication.

Processor Modules

The processor module in the original ACPMAPS, called the FPAP, is based on the Weitek XL-8032 processor, a 3-chip set. The floating point unit has a short (3 stage) pipeline and the integer instruction set is of a RISC flavor. The peak speed of the FPAP is 20 Mflops; 8 Mbytes of data memory (and 2 Mbytes of instruction memory) are provided. The upgraded system uses a D860 board, which contains two independent processor units, each based on an Intel i860 CPU. This chip also has a 3 stage floating point pipeline and a RISC-like integer instruction set. The peak speed of each i860 processor is 80 Mflops; each processor is supplied with 32 Mbytes of memory. Thus to first order, the D860 board is 8 times as powerful as the FPAP.

The D860 memory uses 4 Mbyte DRAM chips; these are mounted on SIPs to minimize the board space needed. (Custom SIPs were required because the BSB crate uses the same board spacing as VME, and most off-the-shelf SIMMs are too tall for this spacing.) The memory system for each node is controlled by fast PALs and can deliver a 64-bit word of data every 50 nsec. The latency for memory access depends on whether there is a "page" change (a change in the row address supplied to memory — the "page size" for this purpose is 4K bytes). Without a page change, the data is delivered 50 nsec after the address is asserted (this is as fast as the i860 can take it). If a page miss occurs, the latency is 125 to 175 nsec.

The cycle times of the FPAP and i860 processors are respectively 100 and 25 nsec. Their floating point architectures are very similar: One 32-bit floating point add and one floating multiply can be performed every cycle, with a pipeline length of 3 steps. The memory size and access bandwidth also scales with the cycle speed, although the page miss latency is only two extra cycles for the FPAP. One difference which can be important for certain problems is that the i860 can perform 64-bit floating point operations (at half its single precision speed), while the FPAP has no double precision capability.

In spite of the similarities in their architectures, the i860 is somewhat less efficient in running actual problems. For example, where the FPAP might achieve an actual performance of 7 Mflops per processor on a particular algorithm, each i860 might only get 18 Mflops (rather than 28). The main causes of this inefficiency are related to memory access. Although the bandwidth to main data memory is one 32-bit word per cycle in both cases, the memory latency for page misses is much greater for the D860. Also, the FPAP uses a separate bus for instruction memory fetches. While these effects should be offset by the fact that the i860 has internal cache, the cache is rather small and uses a write-back virtual cache strategy that leads to surprisingly frequent cache misses. Although the

D860 is less efficient relative to peak power, the factor of four increase in peak speed and memory size (and the presence of two processor nodes per module) means the upgrade to D860's represents a substantial increase in effective computing power.

Distributed I/O

The large-scale distributed I/O subsystem is based on SCSI disk and tape drives. These are "commodity" devices, available at low cost. The tape drives are Exabyte helical scan 2.3 Gbyte 8mm devices; the disks currently used are WREN VI drives with 650 Mbytes of data each. (Thus the 32 disk and 32 tape drives provide 20 Gbytes of disk space and 70 Gbytes of tape on line.) These disk and tape drives occupy 16 separate SCSI buses (two tapes and two disks per bus); although up to seven devices can share a bus, the current arrangement was chosen so as to minimize inefficiencies due to SCSI bus bandwidth limitations (2-4 Mbytes/sec per SCSI bus).

The connection between the BranchBus based communications backbone, and the SCSI buses used for I/O, is based on modules residing in 4-slot VME crates. These are referred to as **VME cratelets** and are located inside each BSB switch crate. (The switch crate backplane is somewhat wider than required for 16 slots, and deeper than a standard 6U VME card; this makes this compact arrangement possible.) The physical arrangement is shown below:

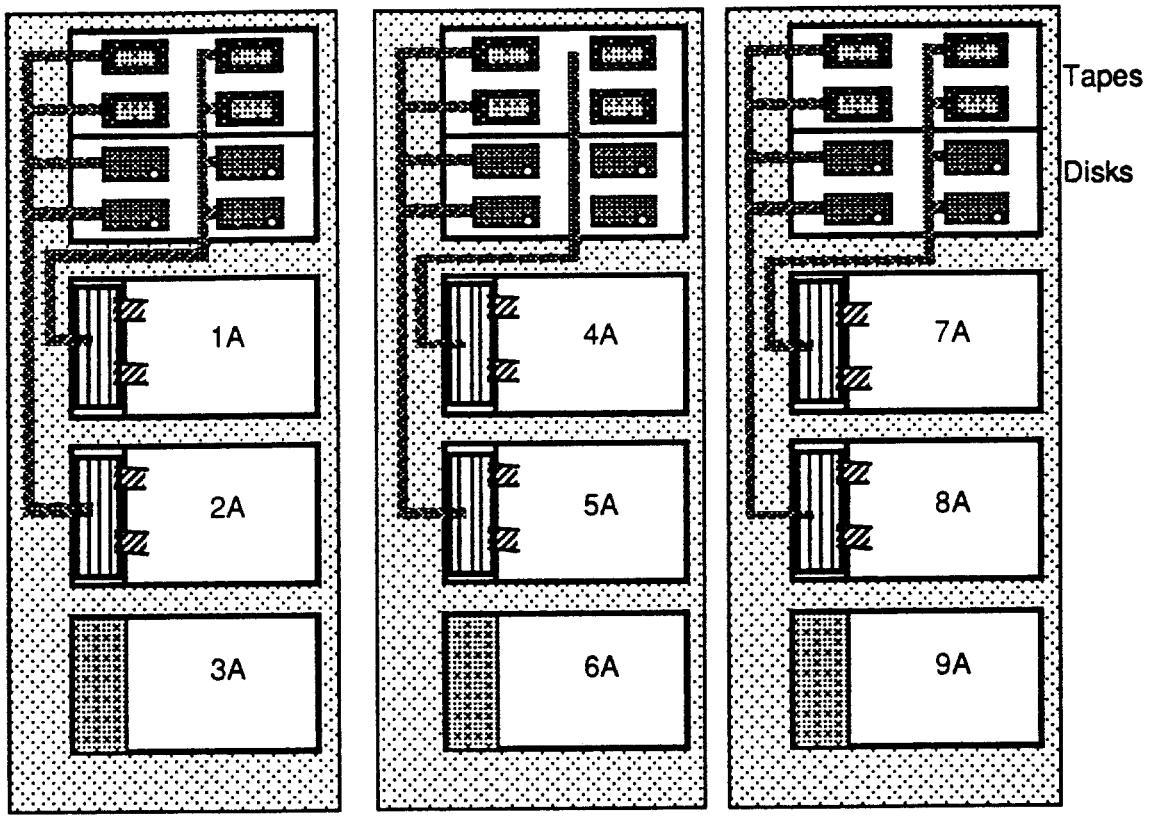

Figure 5: One plane of ACPMAPS, showing SCSI connections for I/O

Each VME cratelet contains:

- a **VRM** VME Resource Module (a crate controller);

- a Ciprico VME-based SCSI bus controller (which controls both Exabyte tape and WREN disk drives);
- a **BCI** BranchBus Ciprico Interface which allows the BranchBus to act as a master on VME in a manner appropriate for using the Ciprico — this is a slight variation on a basic BrachchBus to VME interfaces (BVI);
- and a 4 Mbyte VME memory for use as I/O buffer space.

The data coming from the disks goes over SCSI to the Ciprico, and from there is placed (over VME) into the buffer memory, to eventually be read out across VME and through the BCI into the main communications backbone. This is illustrated below.

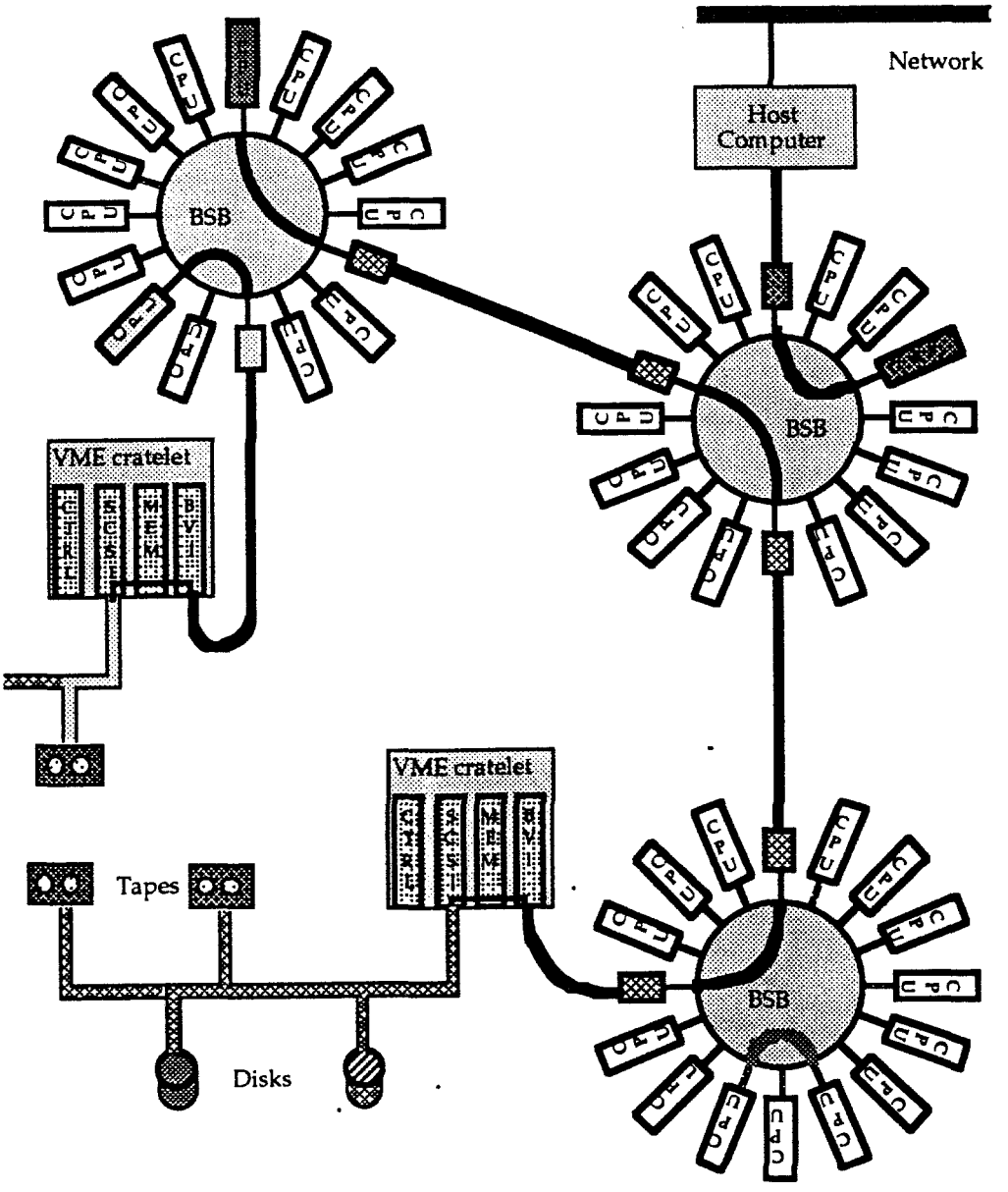

Figure 6: An illustration of a CPU controlling a disk drive

Note that since the host can act as a master on the main communications backbone, it can directly control the I/O devices. Normally, however, the I/O to a given disk or tape drive is controlled by a designated "I/O captain node", which gathers the necessary data and sends it to the VME memory and sends orders to the Ciprico. The disk drives each operate at up to 1 Mbyte/sec; the SCSI bandwidth is 2-4 Mbytes/sec. Each word of data traverses the VME bus twice (once to get into the buffer memory, then again to get out across the BCI); the VME bandwidth is nominally 20 Mbytes per second but realistically half that — 10 Mbytes per second is more than adequate to keep up with the SCSI devices. The 20 Mbytes/sec bandwidth on the BranchBus in the main communications backbone allows the captain nodes to gather and send data without slowing down any of the I/O devices. The aggregate system bandwidth to disk is roughly 32 Mbytes/sec.

The Host

The host computer for the ACPMAPS system needs to be able to act as a master on the communications backbone (but need not be accessible as a slave). In addition to downloading the node program to the control node for each job (the remaining downloading is then done by the nodes in a tree-like fashion), the host will monitor the control node to service potential UNIX calls, and will occasionally monitor the remaining nodes to check for exceptional conditions. To act as a master on the communications backbone, the host must become a master on BranchBus; various modules allow different machines to accomplish this.

The computers currently used as hosts are a Silicon Graphics SGI 4D25/S, and a MicroVax 3200 running Ultrix. The SGI has an internal VME bus, in which we place a VBBC (VME - BranchBus Controller) which can be a BranchBus master with transfer rates of up to 20 Mbytes/sec. The Ultrix Vax has an internal Q-bus, in which we place a QBBC (Q-bus - BranchBus Controller). The transfer speed through this is limited by Q-bus considerations, to about .05 Mbytes/sec. This is still adequate for the duties required of a host, but there is considerable impact on the performance of the Ultrix Vax due to bus cycles taken by the QBBC. For this reason, the SGI is superior as a host when the workload includes multiple simultaneous jobs.

ACPMAPS Software

Two distinct aspects of the software are of concern to the ACPMAPS user. There is the issue of how to code up an application. The Canopy library comprises a set of tools for coding a grid-oriented algorithm in a natural manner, such that the multiprocessor aspects are handled automatically. The paradigm and concepts underlying Canopy, and the typical ways of using key routines, are discussed in an overview of Canopy later on.

The other aspect is the issue of how to cause an application to run on the multiprocessor system, and what runtime support can be expected.

Creating and Running Applications

When an application is to be run on the ACPMAPS system (or any system supporting Canopy), the following steps occur:

- The application is written in C, using whatever Canopy routines are appropriate. (The C language was selected because it is ubiquitous, and because pointers are handled in a natural way. Hand-optimized assembly code can, of course, be linked in for major production programs where efficiency is critical.)
- A shell automates compiling and linking the application with the necessary Canopy library routines. There are several such shells: canc to create an executable to run on the current (single processor) computer; acanc for creating an executable for the 5GF system's FPAPs; and dcanc to create an executable for the 50 GF system's D860 modules. These commands are similar in form and arguments to the cc (C compiler) command.
- To run the application on the multiprocessor system, the user issues a canopy command, which takes the form
  canopy <*number_of_nodes*> [<*time_limit*>] <*executable*> [*args*]
  args are passed to the program in the usual (argv, argc). The canopy command is supported on the system host computers.
- If the resources requested are available, the job will start immediately; otherwise, a message will be issued informing the user that the job is being held. If the job is held, one can wait or abort via a ^C.
- The program runs on the nodes. As will be seen in the overview of Canopy, the program is a single-thread application (called the "control program") running on one "control node", with occasional invocations of tasks to be done by many processors. The control node concept is discussed further in the context of the overview of Canopy.
- Standard input is taken from stdin; thus scanf can be used in the control program. The canopy command can be issued interactively, as part of a shell script which may contain further data for stdin, or at the end of a pipe which generates stdin.
- Standard output and error output are directed to stdout and stderr (stderr will also be placed into a system log file). Normally, only the control program will issue any printf commands.

- The canopy command returns the program's exit status when the job completes.

How the ACPMAPS System Software Runs Applications

Although the user may view the system as a "black box" which takes canopy commands and produces results according to the program executed, it may be helpful to see what happens when a job is run. Two key software components running on the host computers are the spooler and the canopy hosting tool. The spooler manages system resources; a single instance of this program is run on one of the host computers. The canopy hosting tool provides host support for a job; one instance of this program runs (on any host computer) per application running on the system.

The ACPAMPS system is coordinated by a daemon process, called the spooler, which runs continuously on one host. The sequence of events described below is initiated when a user logged in to one of the hosts issues a canopy command.

- The canopy command sends a message to the spooler process (which may be running on the same computer or on another host).
- The spooler determines whether the job can be run at this time with the resources requested — if not, a "**job held by spooler; wait or ^C to abort**" message is sent back to the canopy command.
- If the appropriate resources are available, then they are assigned to this job, and the spooler will allow the canopy hosting tool run the job.
- The hosting tool will cause the downloading of the executable to all the assigned nodes. (For efficiency, this is done by downloading the program and the node numbers of all assigned nodes to a single control node, and commanding the control node to complete the downloading. The remaining nodes are loaded in a tree, without further host participation.)
- The user program begins running on the system. While it is running, the hosting tool continues operating, to handle any Unix call issued by the job. Normally, only the control node will issue Unix calls to be handled by the host. Unix calls (other than locally handled heap management calls) issued by other nodes are not formally supported within Canopy — since these nodes are always computing in parallel, the ordering and effects of the calls would be hard to define. Nonetheless, in exceptional circumstances, any node can issue a Unix call. (For example, an fprintf to report an exception will lead to a write() Unix call.) The hosting tool will poll all the nodes on a several second time scale, to verify that the job is proceeding with no obvious problems.

- Several circumstances can indicate that the job has completed:
  - » Normally, the control node will issue an exit() call after the computation has been done.
  - » Any of the nodes can issue a terminate_job() call cause a job to end prematurely.
  - » A fatal exception on one of the nodes will cause the job to terminate.
  - » The user can force a termination by issuing a ^C to the canopy command.
  - » The job can time out by exceeding its time limit.

  In any of these cases, the hosting tool will report to the spooler the fact that the job has terminated, and will exit with the appropriate status.
- The spooler detects that the job has completed (normally, by a message from the hosting tool). It will then de-assign the resources which had been assigned to this job. As long as the spooler program is running, the system and all available resources can be used, even if the other host computer has crashed while running hosting tools which had been assigned resources.

Canopy

Canopy is a software underpinning which allows a user to design a multiprocessor application without having to worry about the details of the parallel system architecture. Canopy has been running for several years on several systems including ACPMAPS.

In this overview, we will:

- Explain the intent and goals of the software underpinning;
- Discuss the general approach to achieving these goals;
- Discuss the key concepts involved in the Canopy paradigm;
- Outline the strategy used by Canopy to implement these concepts and run an algorithm on a multiprocessing system;
- Discuss what sort of problems are and are not suitable to approach via the Canopy paradigm.
- Discuss the costs associated with using Canopy.
- A brief list of Canopy routines and data types.

Further details about Canopy can be found in the Canopy Manual. Any user writing a Canopy application should obtain a copy of that document.

Goals of Canopy

Our overall aim is to provide the scientist with a straightforward way to get from the natural description of an algorithm to an effective implementation of the algorithm. In particular, lattice gauge algorithms typically involve discretized local differential equations, described as some operation done on many points in space simultaneously. Therefore the goal is to use these concepts directly, so that once the user has stated his algorithm in these terms, the intellectual effort involved in structuring the program is done.

Previous approaches required the user to intersperse the natural description of the algorithm with explicit details of how data is to be distributed and accessed, how workloads are to be shared, and how boundary conditions are to be handled. The goal of Canopy is to eliminate the need to think about these things, without introducing other difficulties. We have found a way to do this by using a slightly different paradigm to picture the parallelism of the algorithm.

General Strategy

Canopy provides a set of concepts for thinking about algorithms, a methodology for expressing algorithms in terms of these concepts, and a library of routines implementing this methodology. We have implemented this library in C to take advantage of the availability of C software tools.

Several particularly important considerations are:

- **To provide a natural and unified way for the physicist to express the geometric aspects of an algorithm.** By geometric aspects, we mean concepts like "the site to my left" or "there are 3 grids, of sizes A, B and C". This allows the user to express the local operations independent geometric considerations.
- **To provide a natural way to express the flow control of an algorithm.** Here we mean concepts like "do this step for each site in that set". These concepts are used to automatically take advantage of the parallel system in a straightforward way. The user has provided the necessary insight as to the parallel nature of the algorithm.
- **To shield the user from the details of the parallel system.** These details include most of the effort needed to make the program run in parallel — distribution of data, control of multiprocessor execution, and so forth. In particular, the user need never know anything about the number or connectivity of processor nodes.
- **To automate routine lattice address calculations.** Once the user has expressed the geometric concepts of the algorithm, automating and hiding these details becomes straightforward. The benefit is a factoring out of the constant clutter of "take this index, check against boundaries" and so forth — this brings the code much closer to the way a physicist thinks about an algorithm.
- **To improve overall program robustness.** When an algorithm is expressed in terms of the proper set of standardized, basic concepts, the program tends to be clearer and easier to verify. The fact that the user need not care about which (or how many) nodes are being used makes the system more flexible (in terms of sharing between users). Shielding the user from the details of the parallel hardware makes Canopy applications trivially portable over a broad class of machines meeting certain minimum requirements (as discussed below).

These considerations have several desirable side effects:

- **Better organized, more modular code.** This comes from providing a natural way of expressing the geometric and flow control aspects of an algorithm. It becomes easier to re-use portions of a program in different settings, and to identify and optimize critical code sections.
- **Easy program modification.** When the routine address calculations are automated and "factored out" of the explicit physics code, it becomes easy to modify lattice size, shape, connectivity, boundary conditions, and so forth. Canopy supplies a library of grid definition routines, but the user is free to specify a grid with special properties as well.
- **Optimizations can be done by systems people.** For instance, a side benefit of automating address calculations is that the system

software designers can afford to spend a lot of thought identifying the cleanest and most efficient way to do the calculations. Although in principle for any given algorithm the scientist could come up with a faster method, the automated scheme turns out to be superior to the code actually written when scientists explicitly guide the address computations.

Note that although the details of the parallel nature of the system are hidden, the parallel nature of the algorithm is not; in fact Canopy makes it natural to highlight the tasks that are to be distributed over many processors. Thus Canopy does not attempt to be an automated parallelizer (which would be very difficult to do well). The scientist has better knowledge of how to parallelize an algorithm than any pre-conceived system could possibly hope for. The idea is to make it easy for the user to convey that knowledge, and to use that information to control the program's parallel execution.

Concepts

The set of concepts described here are designed to achieve the above goals for grid-oriented problems. Chapters 2 and 3 of the Canopy Manual Version 4.0 is a detailed tutorial of the Canopy concepts; in the following short summary C structures are presented in bold face.

Canopy introduces structures which match the natural concepts of discretized differential equations where the operations are done on some set of **sites** which are connected together into a **grid**. Sites in Canopy are exactly the sites of the discretization; a grid in Canopy is the same as the discretized grid. Canopy further introduces the concept of a **direction** from a site leading to a neighboring site on a grid, and the concept of a **path**, which is an ordered set of directions that lead from one site to another. The variables in a discretized problem are **fields** over a grid; Canopy provides field variables that live on the sites of the grid and fields on the links joining the sites. These fields are then the usual targets of operations, and the repository of the mass of data associated with the problem. Canopy allows more than one grid to be defined and provides for **maps** to connect them together.

Canopy flow control is divided into three parts: first, when the program starts, a single-thread control program is running as an ordinary C program. This control program has a declaration section, where grids and fields are defined, followed by an execution section. The execution section invokes tasks, which execute task routines in parallel on **sets of sites**. The task routine is called once for each site in the set, and has special mechanisms for passing arguments and collating return values.

Geometrical concepts — details

Physical problems represented by differential equations on a continuum in some space are often attacked by discretizing the space — treating the continuum as a lattice of points. The differential equations become relations between variables defined on these points. This approach is called the grid approximation. The physical concepts of **sites**, **grid** and **fields** refer to the individual points, the lattice comprised of the entire collection of

points, and the variables in those relations. Canopy defines constructs which correspond to these physical concepts.

site

A **site** corresponds to a point on the lattice. Associated with each is some field data, and other properties such as a site serial number, coordinates, and information about neighboring sites. Every site on a given grid is identical in structure to every other site. Parallel execution in Canopy is logically done simultaneously over a set of sites. Thus a site is similar in concept to a virtual processor — the field data corresponds to the local memory. Routines refer to particular sites by means of a site variable: A variable of typedef `site`.

grid
dimensions
coordinates
directions
links

A **grid** is a collection of identical structures (representing the sites which make it up). Certain natural geometrical properties are associated with a grid. These describe where the sites on the grid are, and how they are connected to one another. The grid is organized as having some number of **dimensions**, — each site is assigned **coordinates**, represented by an array of length matching the number of dimensions of the grid.

Another natural and useful geometrical property of a grid is that of connectivity. The grid has some number of **directions**: Each site may have other sites as neighbors in the positive and negative units of each direction. Although in a Cartesian lattice the number of directions equals the number of dimensions, and each site has neighbors every direction, neither of these properties need hold for general grids.

The concept of boundary conditions on these connections is natural : A site on one "side" of the grid might have as a neighbor in each direction any desired site, or no site at all. The concept of directions and neighbors often provides the most natural and convenient way of specifying another site in local algorithms. It is also natural to picture the lattice as having **links**, lines connecting sites with their neighbors. Although there is no link type defined by Canopy, there is the concept of a **link_field** — data which is logically associated with each link rather than with each site.

Canopy has a collection of pre-defined grid declaration routines, for convenient creation of periodic Cartesian grids in various sizes and dimensions. The user can also use the function **arbitrary_grid** to create a grid of any desired properties. Grid declaration routines return a variable of typdef `grid`. This grid identifier is used in other routines where a grid must be specified — a given application may use several distinct grids.

field
site_field
link_field
field elements
field_pointer
link_field_pointer

A **field** is a collection of data structures — one instance of a structure associated with each site on a grid. Many physical problems involve *fields* of some nature defined on the continuum, such as the electric field in space or a gluon field defined over space-time. When such problems are solved in a grid approximation, the continuum fields are replaced by fields on the sites or links. In Canopy, a field is defined by calling **site_field** or **link_field** — these routines reserve space for **field elements** on each site or link in the grid. The field declaration routines return a variable of typedef `field`, which can later be used as an identifier to specify that particular field. In an application which declares several grids, a different fields might be defined over each grid.

Algorithms often involve accessing field data associated with some given site. Canopy shields the user from the complexities of address computation and off-node data access by providing routines to handle all access to accessing field data.

Read access to site field data is done by **field_pointer** routines, which return a pointer to a read-only copy of the field element. The arguments to these routines specify the desired field, and the site with which the desired data is associated. The user need not know whether the data was kept on the local processor node or on a remote node.

Access to elements of link fields is provided by routines such as **link_field_pointer**. The user specifies the field, the site at which the desired link originates, and a direction to select which link field element is to be accessed.

site variables
site manipulation routines
path
map

One way to specify the site in a field access routine is by specifying a **site variable**. Canopy provides a typdef `site`; such a variable can refer to any site on any of the grids declared for the application. **Site manipulation routines** are provided to set a site variable by specifying coordinates, or relative to another site by one unit in a specified **direction**, by a specified **path**, or by a **map** to another grid.

put_field

When modifying (writing) a field element, a **put_field** routine is used. These routines perform a copy into the appropriate field element. Again, the user need not know whether the field element is stored on the local processor node or elsewhere.

Flow control concepts — details

```
control()
  declaration section
  complete_definitions
  execution section
  tasks
```

The main routine of a Canopy program — always called **control()** — sequences the activities in the job. Before multiprocessor activity can be started, the control program must define the geometric and data structure concepts relevant for the application. This is done in a **declaration section**, which invokes grid definition routines to establish the geometry, and field definition routines to establish the data structures. This section can be interlaced with code to accept input and to perform computations. The declaration section is terminated by a call to **complete_definitions()**, which causes all requisite information about geometry and data structures to be sent to the individual processing nodes.

The **execution section** of the control program expresses the body of the algorithm. Some single-thread global computation and decision making occurs in this section, but the most important activity is the initiation of **tasks** — routines to be executed in parallel on many sites, enlisting the aid of all the processor nodes involved in the job. Each task is invoked by calling the routine **do_task**, and completes on all sites before the control program continues execution. The algorithm can take advantage of the multiprocessor nature of the system to the extent that most of the computational burden is in these tasks.

```
do_task
  task routine
  set_of_sites
  set and grid typedefs
  pass arguments
  integrate arguments
  do_task triplets
  example of do_task and triplets
```

The **do_task** routine is a remote, multiprocessor subroutine call. To initiate processing on the processor nodes, two things must be specified: The **task_routine**, that is, the routine to be invoked for each site; and the **set_of_sites** over which this routine is to be executed. The set of sites is represented by a variable of typedef **set**. This can be an entire grid: a grid identifier — of typedef `grid` — can always be used in place of a set of sites. Or the set of sites can be defined during the declaration section, or dynamically during the execution section. The task routine will be called once for each site in the set, in an arbitrary order. The do_task call returns to the control program when all these task routine invocations have completed.

A mechanism allows for passing arguments to the task routines. These will appear to the task routine as ordinary C arguments, passed by pointer. However, in the argument list to do_task, they must be specified in a special way: In the context of parallel execution, where the task routine will be run in an undefined order (and in principle simultaneously) on

many virtual processors, concept of a subroutine argument is not unambiguous — guidance is required.

First, it is necessary to distinguish between arguments which, from the viewpoint of the task routine, are "read-only" versus "write-only" . In a parallel environment, arguments which are read-only (in the sense that the task routine uses but never alters their values) are handled by communicating their values to each processor node and making them available as ordinary arguments. Canopy refers to these read-only arguments as **pass arguments**.

Other arguments can be considered "write-only": The task routine is passed a pointer, and returns a result by writing it to the indicated address; the calling routine passes the pointer and expects the subroutine to place a single result there. In the context of parallel execution, the results from many virtual processors must be amalgamated into a single result. These "write-only" result arguments are called **integrate arguments**: The user specifies how they are to be amalgamated into a single value. For example, the results from individual task routines can be added as floating point numbers, or to the maximum among the individual results can be taken.

On a single processor it is easy to pass an array of values, simply by supplying a pointer to that array — in a distributed memory environment, this does not work in a naive manner . The mechanism provided for specifying the nature of an argument also provides for passing arrays of values. Each argument to the task routine is presented to do_task as a **triplet** of arguments. The triplet consists of:

- A keyword describing the nature of the argument. The options include **PASS** to supply a read-only argument, and various options for result arguments: **SUM_INTEGER**, **SUM_REAL**, **MAX_REAL**, and so forth. There is also a mechanism for customizing, in case a required amalgamation technique is not among the options provided.
- A pointer to the argument itself (the argument may be a single word, or an array or structure).
- The size of the argument — this is what guides Canopy in determining how much data to communicate to the processor nodes.

Thus do_task takes as arguments the task routine to be done, the set of sites to do it on, as many arguments are desired (one triplet per argument), and an END keyword do delimit the list of arguments. For example:

```
/* in control() */
float f[4];   float sum;   field  *x;
do_task (   my_task , this_set_of_sites
            PASS, &f, 4*sizeof(float),
            INTEGRATE, &sum, sizeof(float),
            PASS, x, sizeof(field *),  END   );
            . . .
/* elsewhere in program, task routine appears: */
void my_task ( float *array4, float *ans, field *x ) {
/* code which uses array4 (f in the calling program) */
/* and the field x to produce a result ans */ }
```

In this example, `my_task` does something to field `x` (it might elsewhere be invoked for some other field) based on array `f`, to produce an answer in

*ans; these answers are summed for all the sites in this_set_of_sites and placed into *sum in the control program.

Since the task routine is invoked for many virtual processors, results of the task might be desired in the form of an array of answers, one associated with each site. There is a natural way to handle this in Canopy: Such an array is a field; the task routine writes its answer by calling put_field.

C allows arguments which are "read/write": A value can be passed by reference, and the pointer to that value can later be used to return a result. This sort of argument can always be split into a read-only argument and a result argument. Canopy does not support read/write arguments to task routines.

field access during tasks

HOME site
field access relative to HOME
Changing fields at the HOME site

A task is logically performed over an entire set of sites simultaneously. The task routine is invoked for each site in the set. At any given time, the processor is doing the computation for a particular site, referred to as the **HOME site**. Canopy defines a variable of typdef site called **HOME**, which refers to this site.

The task routine will virtually always involve accessing (and usually modifying) field elements at this HOME site. Non-trivial algorithms will also involve accessing data belonging to other sites. Field data is always accessed via calls to Canopy field access routines — field_pointer for read-only access, and put_field to modify data. (Task routines should modify data at the HOME site only.) For convenience during task routines, there are routines are provided which access fields at sites relative to the home site — offset by a direction or a path. For example, **field_pointer_at_dir** takes as arguments a field and a direction, and is equivalent to setting a site variable to HOME, moving that site variable one unit a direction, and using that site in field_pointer.

During task routines, there can be one important exception to the rule that field data cannot be changed except by using the put_field routines: When field_pointer has been used to access a field element on the HOME site, that field element is known to be stored on the local node. Thus data belonging to that element can be changed in place, without concern about later copying the new element into the proper location on a remote node.

The control program can use the field access routines to read or write field data, but the concept of the HOME site is valid only within task routines.

global variables

broadcast

Canopy applications differ from single-thread programs in their treatment of global variables. The ability to set values to be known throughout an application is quite useful. Nonetheless, modern programming philosophies disparage the use of global variables, because subtle errors can be introduced through their misuse. Further complexity (hence scope for error) is introduced when multi-processor systems are

considered. Canopy supports global variables, but requires that a set of rules be followed to prevent erroneous usage:

- Values of global variables may be changed only by the control program, not by a task routine. (Were a task routine to alter a global variable, the timing of when the change was to occur — relative to the "simultaneous" execution of the same task routine for other sites — would be ambiguous.)
- Having modified the value of a global variable, the control program must call the Canopy **broadcast()** routine, to inform all processors of the change. This must be done before invoking tasks which use the variable. Task routines may not call broadcast.
- Dynamically allocated global variables are not permitted. That is, a structure which was created by a declaration in the control program can be used as a global variable, but a structure which uses memory allocated by malloc cannot.

The above rules apply to variables which are referenced by task routines. Global variable which are employed exclusively within the control program can be treated as per ordinary C global variables.

```
compound tasks
 levels in sets of sites
 sync_field_pointer
 do_task_n_times
```

Certain algorithms have steps which require performing computations on an ordered set of sites — the task routine for some sites must be completed before the routine for other sites is allowed to begin. To facilitate coding such algorithms, Canopy extends the concept of a task running a routine once per site in no particular order.

When a set of sites is defined, each site in the set may be assigned a **level** (multiple sites can be at the same level). We call a set of sites with unequal levels a **compound set of sites**. A task routine can be written to run on a compound set of sites in such a way that the routine is logically executed for sites with lower levels first.

To maximize the opportunities for parallel execution, the synchronization is enforced when field data is accessed, rather than when execution is initiated for each site. The task routine accesses field data which is liable to be changed using **sync_field_pointer** instead of field_pointer; this will wait until the site owning that data has been processed, if it is at a lower level than the HOME site. Thus, one processor might be handling level 10 sites while another is only up to level 2; the first does not block until it needs data that has yet to be updated at a lower level.

Logically, compound sets of sites accomplish nothing that could not be done by defining multiple disjoint sets of sites (one for each level) and calling do_task multiple times. There is a convenience advantage in grouping these multiple do_task calls into one **compound task**. A compound task can also be more efficient than the individual simple tasks, since .

A special sort of compound task is invoked by the **do_task_n_times** routine. The user employs explicit synchronization (sync_field_pointer)

within the task routine. Again, the advantage is that synchronization is done only when necessary, not between each invocation of do_task.

distributed I/O routines

```
open_field_file
read_field/write_field
slice_of_field
```

The distributed I/O system is used to store field data. The control program initiates all field file activity. The Canopy routine **open_field_file** is used to specify the name of a file; on ACPMAPS, we use a leading # character to signify a file on the distributed I/O system. Then **read_field** and **write_field** routines can be called. The field is written and read back in a manner independent of the number of nodes being used for the job — a job running on many nodes can save a field to be examined by a job running on fewer nodes, and vice versa.

Canopy provides a **read_slice_of_field** routine to fill a field with only a portion of a previously written field file. For example, one might require only a single time slice of a large lattice. This feature is indispensable for algorithms which require one small part of each of several huge fields — reading in the complete fields might be prohibitive in terms of memory space.

Implementation Strategy

The details of the implementation of Canopy concepts are in principle irrelevant — other than the fact that they work. Canopy has been implemented since 1989, and has achieved its goals as an aid in algorithm development. Canopy applications run applications with good efficiency on ACPMAPS. Nonetheless, a discussion the general principles and strategies is useful in several ways:

- An understanding of how things work often leads to insight about the best way to use them;
- A grasp of how Canopy is structured can help in evaluating the suitability of other platforms for a Canopy port;
- A knowledge of implementation strategies can point out potential areas of inefficiency, which may help in designs of future Canopy platforms.

Organizational Strategy

Canopy is organized into three pieces. The main user-level Canopy routines appear in several files amounting to 7,000 lines of C code. These are compiled and put into a library that the `canc` shells link with the user code. There is a canopy.h file which the user program includes to obtain function prototypes for all the Canopy routines, and definitions for the keywords used.

The second piece is a collection of libraries for frequently needed functions. These include a library of grid definition routines, a collection of functions for complex variables, one for random number routines, one for Fast Fourier Transfer operations, and so forth. New routines can be added to these without altering the basic set of Canopy concepts.

The Canopy library and the supporting libraries are written in C and independent of the target system. Canopy assumes it is running on a system with a particular, well defined interface for doing things like interprocessor communication. This model is defined and implemented by the final piece in the Canopy organization: A set of underlying routines called the Canopy Hardware Interface Package (**CHIP**).

The CHIP routines have well defined arguments and results (described in the Canopy Manual) implementing the model Canopy uses. For example, the basic primitives for interprocessor communication, called by higher-level Canopy routines, are part of CHIP — these operate in terms of Canopy concepts such as full addresses. CHIP includes routines like remote_read, remote_write, and do_on_all_nodes. Although these routines have interfaces defined by the Canopy system model, their implementations will be machine dependant. The system model is rather simple; the implementation of CHIP typically is about 2,000 lines of code.

This strategy of isolating the machine dependant implementation details and providing a well-defined system model has two positive implications:

- To port Canopy to a new system, one only needs to port the CHIP primitives. For example, if the communication on the new system is

based on a message passing protocol, one must write CHIP routines like remote_read in terms of that actual protocol. The bulk of the code that makes up Canopy remains unchanged.

- The sophisticated user has the option of calling CHIP routines directly (bypassing the Canopy concepts), without sacrificing portability. This may be useful in implementing aspects of an algorithm which fit the Canopy paradigm poorly. Some of the benefits of Canopy will be lost when this is done. For example, the user can write a routine which is no longer independent of the number of nodes in use.

Implementation of Concepts

Canopy contains the concept of a site as a virtual processor. The key implementation strategy is: **The responsibility for all the sites in a grid is divided (roughly equally) among all the processor nodes in the job.** This responsibility includes storage of field data associated with the site, and processing of task routines to be run for that site. All processor nodes, including the control node on which the control program runs, are apportioned a share of the sites.

When a task is running, each processor node will execute the task routine for every site in the selected set which it owns, in an arbitrary order. Canopy says nothing about the nature of these task routines; thus the processors must run independent instruction streams (MIMD). When a task is done, the program flow must wait for every processor node to complete the task routine for all the sites owned. If each processor node has responsibility for many sites, the fluctuations in time taken to perform the task routine for a single site tend to average out. Although the entire system must wait for the last node to finish, this wait is usually a small fraction of the overall time taken to perform the task. **Canopy works most efficiently if the number of sites greatly exceeds the number of processor nodes.**

The task routines are the only portion of the computation which is worked by multiple nodes. While the control program performs single-thread computations, the remaining nodes are idle. Thus Canopy applications achieve high Gflop rates only in cases where the bulk of the computational work involves tasks — calculations which can be associated with each site.

Address computations — determining the location of data associated with various sites — are done using pointers. Actually, these pointers are of a type called a `full address`, containing information as to node number as well as a local memory address. For example, a `site` variable contains a full address pointing to a zero-point in the appropriate site data. This scheme is not only completely flexible, but is more efficient than explicit address computation. The concept of grid connectivity, with each site having neighbors in various directions, is implemented by storing (as part of the site data) an array of pointers to neighboring sites.

Most algorithms are largely local in nature, (with sites communicating most frequently with some set of neighboring sites). Thus it is desirable to

minimize internode communications needs by distributing the sites among the processing nodes such that **neighboring sites tend to be clumped on the same node**. Since almost all users can use one of the packaged Canopy grid declaration routines, a moderate amount of systems effort in doing this clumping well has benefits for virtually every Canopy application run.

When field_pointer is used, it returns a "read-only copy" of the data requested. Actually, if is on the local node, no copy is made, and a pointer to this field element is returned. If the data is on a remote node, the field element is copied into the local processor's memory, and a pointer to that copy is returned.

Canopy has no concept of virtual processors associated with the links on a lattice. Link fields are stored at the sites: A given site will own the data for the elements of a link field corresponding to links leaving that site in positive directions. Thus on a grid with d directions, there will be d identical field elements on each site.

Canopy assumes the processors in a job are identical, and are running the identical program in the same address space. Thus the control program node can take the name of a task routine as an argument to do_task, and simply send its address (on the control node) to the other nodes to specify the routine. Similarly, the broadcast routine knows where to send global data to each node: It always goes to the same local address as on the control node. (This is why dynamically allocated globals are dangerous.)

A few techniques are used to enhance efficiency. When tasks are started up or completed, and when broadcasts are done, information is sent in a binary tree fashion, rather than creating a bottleneck at the control node. When distributed I/O is done, "captain nodes" are designated to collect the data and send it to the VME memories, and to control the SCSI devices; the control node does not have to do all the work itself. Finally, frequently used small pieces of Canopy have been hand coded assembler for optimal speed on the processors used in ACPMAPS. For example, field_pointer is called so often that this improves performance by a noticeable amount. Such routines are still defined in C — this makes Canopy easy to port. When hand optimizing code, it helps to compare against a known correct implementation.

When is Canopy Applicable

An algorithm is suitable for attacking with the Canopy paradigm if it has the following properties:

- The arena on which the algorithm is to apply can be viewed as a fixed grid of sites. Canopy is more valuable in cases where the grid has some natural connectivity, but this is not required.
- It must be possible to organize the mass of data in the algorithm into many instances (associated with many sites) of a few types of structures (fields). The data associated with any site cannot be very different in size from that associated with other sites, and cannot dynamically undergo large size variations.

- It must be possible to express the algorithm in terms of tasks — sequential steps done across many sites. If the bulk of the computation does not reside in these tasks, then the Canopy application will not take full advantage of the multi-processor system.

To avoid inefficient running, additional properties are desirable:

- The number of sites worked on should greatly exceed the number of nodes, to minimize between-task synchronization losses.
- The average number of internode communications required per site for the task routines, should be small relative to the amount of computation done by the routine. The extent to which this criterion is met depends on the system's communications overhead.
- The computational loads for various tasks should not differ wildly. If the load does vary greatly, but in a static manner, then explicit load balancing by supplying a carefully constructed site distribution function may be possible.

These requirements are met, at least at some level, by a broad spectrum of grid-oriented algorithms. Some problems which appear to lack one or more of these properties can still be profitably approached using Canopy, either by re-couching the algorithm in a minor way, or by accepting a low efficiency in exchange for the greater ease in coding. An inefficient Canopy implementation may still be close to the best that can actually be accomplished on a particular system for a given algorithm.

There are time consuming problems which involve very little data, or involve data which cannot be organized into instances of a few sorts of structures. These problems tend not to be suitable for attack via the Canopy paradigm. Similarly, if the algorithm cannot be expressed as a sequence of tasks, then it may be unsuitable for running with the Canopy underpinning. Canopy currently does not include concepts appropriate for certain changes while an application is running: dynamic load balancing, changing grid size or connectivity, or changing field allocation for the sites ("time-varying numbers of particles at each site").

Costs Associated With Using Canopy

There are costs associated with the advantages of Canopy. The most basic limitation is that the whole scheme is useful because the proper paradigm and concepts for a class of grid-oriented problems has been identified. Problems falling outside this class are at best awkward to attack using Canopy. It may be possible to broaden the applicability of Canopy by expanding the concepts to include other classes of problems, but that has not been done to date.

A second cost is the requirement that the system constitute a "canopy platform" — MIMD processors and asynchronous, global "memory access" style of communication. The multiple instruction stream limitation is not severe, since the intention from the outset was to be able to explore MIMD algorithms, but the communications requirement can be restrictive.

A third price is paid in program efficiency. In order to implement the natural concepts identified, it was necessary to take an approach in which

the basic unit of work is associated with one site of a grid. That is, the natural granularity of the problem is typically small compared with the portion of the entire lattice residing on a processing node. This has implications which we can identify as being associated with computation granularity and communications granularity.

Because the software underpinning works on a one site basis, vectorized or pipelined operations are restricted to a typical length associated with the work to do for one site. For many nearly-SIMD algorithms, there exist alternative approaches in which the length is associated with a fraction of the entire lattice — Canopy cannot take advantage of these efficiency gains. Except for particularly simple and regular algorithms, this "computation granularity" cost is not severe.

Of more concern is a "communications granularity" cost: The Canopy user gives up, for ease of programming, the option of grouping many data accesses (associated with the same remote processor node) into a single access. That means that Canopy magnifies the cost associated with per-communication overhead. For many systems designed without these requirements in mind, this overhead can be quite high; for systems intended as canopy platforms from the start, the inefficiency is tolerable.

The natural Canopy concepts probably cannot be implemented without paying these granularity costs, unless one is willing to restrict the sorts of algorithms which can run. These inefficiencies are partially offset by efficiency gains due to modular program structures and efficient automated address computations. At any rate, it is more important to do the right algorithm slowly than to achieve a high Megaflop rate on the wrong algorithm — effective use of scientists' thought can be more critical than optimized use of computer time.

Canopy and CHIP Routines and Data Types

We list and briefly explain the various jargon terms associated with a Canopy program — defined data types, routine names, keywords, macros, and global variables set up by Canopy. The file canopy.h should be included in the application to provide these definitions, and function prototypes for the Canopy routines.

We also list the **public CHIP concepts**. These are terms involved in the "public interface" to the Canopy Hardware Interface Package (CHIP). Canopy uses these to implement its functionality; the user is free to use CHIP concepts directly in applications. The canopy.h file includes chip.h, which provides definitions and prototypes for the public CHIP concepts.

In addition, Canopy supplies various support libraries for convenience in defining grids, working with complex variables, and so forth. We will list concepts provided by the gridlib library, and grid.h.

In these lists, we have highlighted the more basic, or more commonly used, concepts in boldface.

CHIP Objects (Structures and Typedefs)

CHIP object	data type	
node_bits	structure	Specifies a particular node in the system, along with tag fields which are used in various ways.
full_address	structure	Specifies a memory location in the system. Consists of node_bits and an ordinary pointer.
semaphore	structure	Memory locations set aside for implementing a resource-lock semaphore.
CAN_do_dask_keyword		A structure describing how to handle a multi-node function argument.

Canopy Objects

Some of the objects used in Canopy are internally defined as integers; these are often identifiers returned by a declaration routine (for instance, site_field returns a variable of type `field`). Other objects are arrays of integers. Still others are structures containing a full_address — this is defined in chip.h and is the multinode analogue to a pointer, specifying but node number and local memory location.

Canopy object	data type	
grid	int	A collection of sites with specified coordinate and connectivity propeties.
site	full_address	One of the points composing a grid. A variable of type site is used to specify a position o the grid.
set	int	A collection of sites (forming a subset of a grid).
field	int	A collection of identical data structures. One instance of the structure is associated with each site in a grid.
field_address	full_address	Points to a field element: The structure associated with some field at a particular site. The direct use of field_address can improve efficiency in some cases.
coordinates	int *	Array of integers large enough to hold one value per dimension of a grid. The coordinates associated with a site can have the obvious geometric interpretation.
direction	int	Integer selecting to one of several neighbors of a generic site. The obvious geometric interpretation applies.
path	int *	Array of integers containing a sequence of directions, defining a way of traversing from a generic site on a grid.
map	int	A relationship associating points on a domain grid with points on a range grid.
sync_word	full_address	Pointer to a site to be synchronized within a compound task. The direct use of sync_word can improve efficiency in some cases.

Canopy Keywords, Macros, and Global Variables

The following definitions are provided in canopy.h:

Canopy object	data type	
HOME	site *	Pointer to the home site during task routine.
NOWHERE	site	The null site. If a node has no neighbor in some direction, the pointer to its neighbor points to NOWHERE.
NOGRID	grid	The null grid.

READ	Keyword used by open_field_file.
WRITE	Keyword used by open_field_file.
STREAM_PER_SITE	Keyword used to make a random number generator that generates one stream for each site. Useful in writing applications that produce results independent of the number of nodes used.
STREAM_PER_NODE	Keyword used to make a random number generator that generates one stream on each node. Requires less memory and can be more efficient than stream per site.

CHIP Keywords, Macros, and Global Variables

The following global variables are provided in chip.h:

CAN_number_of_nodes	Number of nodes used by this job.
CAN_number_this_nodes	Index of this node. The control node is assigned index 0.
CAN_this_node_bits	A node_bits data type containing bits for this node.
CAN_node_array	An array containing node_bits for all the nodes in the job.

The following are pointers to CAN_do_task_keyword variables, provided in chip.h to support various argument passing and amalgamation options:

PASS	Pass any type of argument except a function.
FUNCTION	Pass a function.
SUM_REAL	Sum up the returned arguments, as floats.
INTEGRATE	Synonym for SUM_REAL.
MAX_REAL	Take the maximum of the returned float arguments.
MIN_REAL	Take the minimum of the returned float arguments.
SUM_INTEGER	Sum up the returned arguments, as integers.
MAX_INTEGER	Take the maximum of the returned integer arguments.
MIN_INTEGER	Take the minimum of the returned integer arguments.
SUM_DOUBLE	Sum up the returned arguments, as double precision.
MAX_DOUBLE	Take the maximum of the returned double arguments.
SUM_DOUBLE	Take the minimum of the returned double arguments.
TAG_MAX_INTEGER	Return the maximum value (integer, real, or double) and
TAG_MAX_REAL	a tag field associated with it. Useful for finding the site
TAG_MAX_DOUBLE	with the largest value of some function.
END	Not a CAN_do_task_keyword pointer, but just an integer value used to signify the end of the do_task triplet list. The same keyword delimits directions forming a path in the make_path routine.

Keywords Defined in gridlib

The following definitions are provided in grid.h:

X, Y, Z, T	1, 2, 3, 4	Useful in specifying various directions, as in:
MINUS_X	-1	q = field_pointer_at_dir (quark, MINUS_Y);
MINUS_Y	-2	(site *)later = move_site(HOME, T);
MINUS_Z	-3	
MINUS_T	-4	

Canopy Routines

Canopy provides the following declaration routines:

arbitrary_grid	Declare a grid using user-supplied functions for the grid connectivity and distribution. More commonly, a pre-packaged routine from gridlib is used.
site_field	Declare a field over the sites in a grid.
link_field	Declare a field on the links of a grid.
overlap_fields	Force fields to share memory space.
cluster_fields	Force fields to be located consecutively in memory for each site.
set_of_sites	Define a set of sites, for later use in a do_task call.
redefine_set_of_sites	Change the definition of a set of sites. This is the only declaration routine called after complete_definitions.
define_map	Define a map from one grid onto another, via a user-supplied mapping function.
compose_map	Forms a map as the product of two previously defined maps.
make_random_generator	Declares a user-definied random number generator. Pre-packaged generators are supplied in ranlib.
declare_lalloc_size	The field access routines set aside an area for copying fields accessed from remote nodes during a task routine. If an unusually large amount of data will be accessed in this way, declare_lalloc_size can increase the size of that local allocation (lalloc) heap.
complete_definitions	Terminates the declaration section of a program.

Task initiation is controlled by do_task routines:

do_task	Call some subroutine on each site in a set.
do_task_n_times	Call some subroutine on each site in a set, multiple times. Synchronization concepts apply, as for compound tasks.
do_task_on_inverse_image	Used within a task routine, calls a sub-task done on the sites which translate (under a given map) into the HOME site.
do_task_on_inverse_image_set	Within a task routine, calls a sub-task done on the sites which translate (under a given map) into the HOME site, and which are in a specified set of sites. This can be used with a compund set, to create a "compound sub-task".

Site and path manipulation routines:

site_at_coordinates	Set a site variable to selected coordinates on a grid.
move_site	Move a site one step in a specified direction, relative to some other site variable.
move_site_at_path	Move a site according to a specified path, relative to some other site variable.
site_at_dir	Set a site variable to one step in a specified direction, relative to the HOME site.
site_at_path	Set a site variable by moving along a specified path, relative to the HOME site.
is_same_site	Test if two site variables refer to the same site.

grid_supporting_site	Return the grid a specified site is part of.
image_of_site	Retutrn the image of a site, under to a specified map.
inverse_image_of_site	Return a pointer to a list of sites which map into the given site under a specified map.
get_coodinates	Coordinates associated with a given site variable.
get_coordinates_at_dir	Coordinates associated with the site one step from HOME in a given direction.
get_coordinates_at_path	Coordinates associated with the site reached by proceeding from HOME via the given path.
sprintf_site_coordinates	Write the site coordinates into a string.
make_path	Place into an integer array data defining a path, formed from steps in a given set of directions (terminated by END).
extend_path	Extend a path by one step in a given direction.
concat_path	Extend a path by adding another path to the end.
copy_path	Copy the data defining a path, into another integer array.
path_length	Determine the length of a path, in number of steps.

Field access and synchronization routines:

field_pointer	Return pointer to read-only copy of the field element at a given site. May point to the actual fieold data, or a copy brought from a remote node. If the site is HOME inside a task (but not inside a sub-task), then the pointer always points to the actual field data, rather than a copy.
field_pointer_at_dir	Return pointer to read-only copy of the field element at the site one step in a given direction, from the HOME site.
field_pointer_at_path	Return pointer to read-only copy of the field element at the site reached by a path, relative to the HOME site.
put_field	Copy an object into the specified field element at a site;
put_field_at_dir	Copy an object into the specified field element at the site one step in a given direction, from the HOME site.
put_field_at_path	Copy an object into the specified field element at the site reached by a path, relative to the HOME site.
link_field_pointer link_field_pointer_at_dir link_field_pointer_at_path put_link_field put_link_field_at_dir put_link_field_at_path	The link field access routines are the same as the site field access routines, but take one more argument —a direction. They provide access to the link field element on the link originating at a site, and traveling in that direction.
synchronize	In compound tasks, wait until the specified site reaches the current synchronization level.
synchronize_at_dir	Synchronizes with a site specified by direction from the HOME site.
synchronize_at_path	Synchronizes with a site specified by a path from the HOME site.
sync_word sync_word_at_dir sync_word_at_path synchronize_with_synch_word	The sync_word routines provide a slightly more efficient way of syncronizing, which requires some preparation in advance.

synch_field_pointer synch_field_pointer_at_dir synch_field_pointer_at_path	Optimized combinations of synchronization and field access routines.
address_of_field address_of_field_at_dir address_of_field_at_path address_of_link_field address_of_fieldlink__at_dir address_of_field_link_at_path field_pointer_from_address sync_field_pointer_from_address put_field_at_field_address	Some programs can run more efficiently by computing and storing field addresses in advance. These routines support that optimization.
length_of_field_address_field	Given a field_address, return the length of the field starting at that address.

broadcast	Make the value of a global variable set in the control node known to all the task routines in every node in the job.
reset_lalloc	Reclaims all the memory in a lalloc heap, pointers obtained by invalidating previous field_pointer calls. Used primarily if the control program accesses field data, since the lalloc heap is automatically cleared between instances of task routines.
intcpy	Copies a word-aligned array of data in the fastest available manner.

CHIP Public Routines

The routines in CHIP are in general machine dependant. However, the interfaces to the routines are public and fixed in a machine independant manner, so they can be used to write applications which go outside the Canopy paradigm, yet which are portable to other Canopy plateforms. These are the public CHIP routines:

remote_read remote_read_and_keep remote_read_more remote_read_more_and_keep	Read some amount of data from a given full_address into a specified address on the local node. The more/keep versions are advisory only, allowing efficiency gains when multiple blocks will be transfered to the same node. These routines check for full_address refering to the local node, and behave properly in that case as well.
remote_write remote_write_and_keep remote_write_more remote_write_more_and_keep	Write some amount of data from a specified address on the local node to a given full_address. The more/keep versions are advisory only, allowing efficiency gains when multiple blocks will be transfered to the same node. These routines check for full_address refering to the local node, and behave properly in that case as well.

init_resource lock_resource free_resource wait_for_resource	These routines provide a standard way of contending for resources via semaphores. Each resource is represented by a "semaphore variable" at a full_address; thus there can be an arbitrary number of resources, the variables for which are distributed arbitrarily among the nodes.
do_on_all_nodes	Invokes a routine on every node. Accepts arguments in the same triplet form as do_task.

Routines in gridlib

periodic_grid	Defines a periodic Cartesian grid in an arbitrary number of dimensions. The distribution of sites to nodes is done in a way which is reasonable for any shape of grid., but not optimal in terms
chunky_periodic_grid	Defines a periodic Cartesian grid in an arbitrary number of dimensions. The distribution of sites to nodes is done in a way which minimizes the surface/volume ratio for the volume handled by each node. This routine is useful when there is a natural way of dividing the grid into rectilinear blocks and distributing them evenly to the nodes.
periodic_linear_grid periodic_square_grid periodic_cubic_grid periodic_hypercubic_grid chunky_periodic_square_grid chunky_periodic_cubic_grid chunky_periodic_hypercubic_grid	Conveniently define Cartesian periodic grids in commonly used numbers of dimensions.

Acknowledgement

This paper summarizes work which was a collaborative effort between the Fermilab Computer R&D and Theory departments. Direct Contributors to this document include George Hockney and Michael Uchima. Fermilab is operated by Universities Research Association, Inc. under contract with the U.S. Department of Energy.