



Fermi National Accelerator Laboratory

FERMILAB-Conf-92/163

CPS and the Fermilab Farms

Matthew R. Fausey

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

June 1992

Presented at the 26th Hawaii International Conference on System Sciences, Jan. 5-8, 1992, Kauai, Hawaii

Disclaimer

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

CPS and the Fermilab Farms

Matthew R. Fausey

Fermi National Accelerator Laboratory
Batavia, IL 60510

ABSTRACT

Cooperative Processes Software (CPS) is a parallel programming toolkit developed at the Fermi National Accelerator Laboratory. It is the most recent product in an evolution of systems aimed at finding a cost-effective solution to the enormous computing requirements in experimental high energy physics. Parallel programs written with CPS are large-grained, which means that the parallelism occurs at the subroutine level, rather than at the traditional single line of code level. This fits the requirements of high energy physics applications, such as event reconstruction, or detector simulations, quite well. It also satisfies the requirements of applications in many other fields. One example is in the pharmaceutical industry. In the field of computational chemistry, the process of drug design may be accelerated with this approach.

CPS programs run as a collection of processes distributed over many computers. CPS currently supports a mixture of heterogeneous UNIX-based workstations which communicate over networks with TCP/IP. CPS is most suited for jobs with relatively low I/O requirements compared to CPU (2000 machine instructions/byte of I/O). The CPS toolkit supports message passing, remote subroutine calls, process synchronization, bulk data transfers, and a mechanism called process queues, by which one process can find another which has reached a particular state.

The CPS software supports both batch processing and computer center operations. The system is currently running in production mode on two farms of processors at Fermilab. One farm consists of approximately 90 IBM RS/6000 model 320 workstations, and the other has 85 Silicon Graphics 4D/35 workstations. The farms are shared by a half dozen experiments each of which run either raw data reconstruction or Monte Carlo programs. An upgrade of approximately 100 additional processors is in procurement.

This paper first briefly describes the history of parallel processing at Fermilab which lead to the development of CPS. Then the CPS software and the CPS Batch queueing system are described. Finally, the experiences of using CPS in production on the Fermilab processor farms are described.

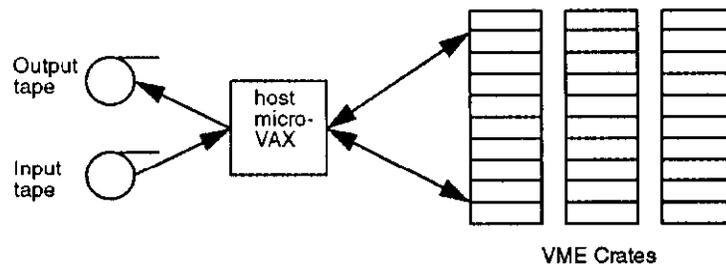
1. History

The field of experimental and theoretical high energy physics is computer technology limited. The amount of new physics that can be done is directly dependent on how many raw computer cycles can be put to useful work. The computing crunch started around 1980, when a new emphasis on heavy quark and strong interaction physics, combined with more elaborate and sophisticated detectors, produced huge amounts of data. This data consisted of hundreds of millions of independent events, each one describing a collision between two or more elementary particles. For each event, particles need to be identified, their tracks reconstructed, and their physics parameters, such as mass and momentum, need to be calculated. Uninteresting events are thrown out and are not considered for the second pass which involves a more detailed analysis. But this first pass, called event reconstruction, would take about a hundred years to compute on a VAX 11/780. Today, an order of magnitude more data is taken and the time required for event reconstruction often exceeds a thousand VAX years. Because the problem maps so obviously onto a parallel architecture, a new group was formed in 1984 at Fermilab to find a cost effective solution to this problem. This group was called the Advanced Computer Program (ACP) and the results of its efforts was the ACP Parallel Processing System [Nash84] [Gaines87a] [Gaines87b].

The idea of the ACP was to integrate commercial components, at as high a level as possible, to produce a parallel computer that could provide the raw processing power of a supercomputer, but at a fraction of the cost. The ACP system consisted of processor boards which plugged into VME crates. Each board was a full-blown computer with a Motorola 68020 microprocessor, a 68881 floating-point coprocessor, and 6MB of DRAM. It ran a simple, single-task operating system capable of handling basic UNIX system calls. Parallel computers were configured from sets of VME crates, each holding 16-20 boards (nodes). A bus developed at Fermilab called the Branch Bus connected each set of VME crates to a host microVAX, where disk and tape resided. The host microVAX drove the parallel computer by sending raw events to nodes through the Branch Bus. Each 68020 node would reconstruct a single event and then send it back to the microVAX to be stored on tape for later processing. The host's job was simple, just keep sending raw events to idle nodes for processing and at the same time collect reconstructed events and write them to some output medium. See Figure 1.

Figure 1

ACP System Topology

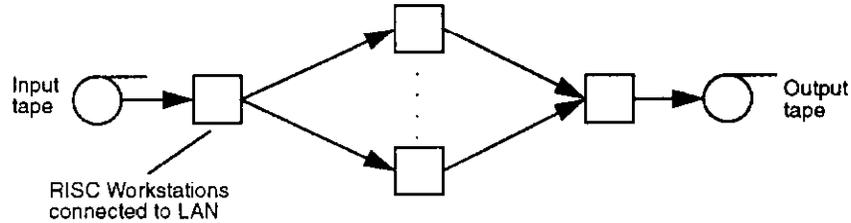


This system was possible because the event reconstruction problem is very compute intensive. Typically 500 to 2000 machine instructions are executed for every byte of I/O. The bandwidth of the Branch Bus made it possible to configure parallel systems containing 70-110 nodes. At its peak, there were 6 ACP parallel computers with over 600 nodes in total. The first systems came on line in 1986 and the last systems were shutdown in May 1992.

The ACP was a success, but it was clearly inadequate for the next round of experiments which began taking data in 1990. The computing requirements for these experiments were an order of magnitude larger than for previous experiments. With the arrival of RISC technology, it was decided to build a new ACP processor board which incorporated the MIPS R3000 processor. These processor boards would each run a complete UNIX operating system, and each would have its own (internet) network address. There would be no need for the host microVAX, since each processor board could write directly to network SCSI devices. It was proposed to completely re-design and rewrite the ACP software, partly because the hardware configuration was no longer asymmetrical, and also because quite a bit had been learned from our experiences with the first system. The project consisted of redesigning the processor board, porting the MIPS RISC/OS operating system to the board, and rewriting the ACP software. This project was completed and some of these systems are in existence today. At about the same time the ACP R3000 project was coming to a close, low-cost RISC workstations became available. The redesigned and rewritten software, which was now called Cooperative Processes Software (CPS) [Kaliher90] [Biel90a], was ported to some of these platforms. A large number of workstations were networked together and run as a single parallel processing system. These loosely coupled workstations were called "RISC processor farms" [Biel90b]. See Figure 2.

Figure 2

RISC Processor Farms (Event Reconstruction Topology)

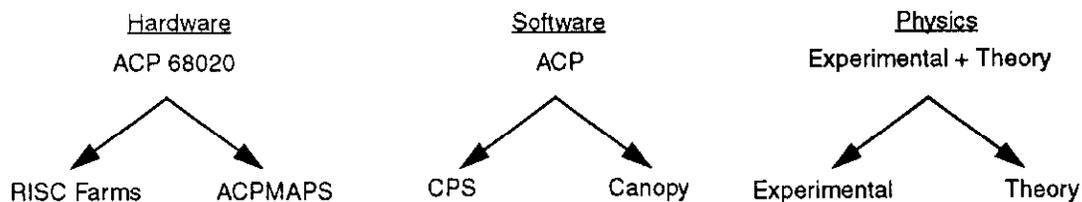


The RISC processor farms take advantage of the explicit parallelism present in the problems of event reconstruction and experiment triggering [Nash89]. The problems of theoretical physics, however, were quite different. Problems such as lattice gauge calculations and quantum chromodynamics map more naturally onto a more tightly coupled grid-oriented architecture. After the original ACP system, development proceeded in two different directions. The first was described above and resulted in RISC processor farms. The second was aimed at the theorists' problems and resulted in a grid-oriented parallel computer called ACPMAPS, which is an acronym for Advanced Computer Program Multi-Array Processor System [Fischler92]. This system is characterized by high floating point performance, extremely dense connectivity, high bandwidth and low latency of communications, and an MIMD architecture. It has run in production at 5 GFlops peak and roughly 1.3 GFlops sustained since early 1991, in its initial configuration. It was based on 256 processors using the Weitek XL chip set. The connectability was 20 MB/channel with an application level message passing latency of 6 μ sec. The system is now running with dual Intel i860 modules as it is being upgraded to 50 GFlops peak.

Just as CPS was developed for the RISC processor farms, a new software system called Canopy was developed for the ACPMAPS. Canopy is designed to support grid-oriented problems. Scientists can use Canopy subroutines to create a grid with known connectivity and to manipulate fields situated on the grid. The parallelism is automatically invoked by Canopy, which calls task routines to do operations on a site. A summary of the evolution from the original ACP to the current systems is shown in Figure 3.

Figure 3

Evolution from ACP to Current Systems. Experimental physicists use CPS software with RISC farms. Theorists use Canopy with ACPMAPS.



2. CPS

CPS is a package of software tools that makes it easy to split a computational task among a set of processes distributed on a RISC processor farm, which is a collection of UNIX-based workstations connected to a local area network. References to other software systems with some features in common with CPS are included at the end of this paper. This includes PVM [Geist91], Linda [Gelernter90], UFMULTI [Avery90], Condor [Litzkow88], and Utopia [Zhou92]. There are many features in CPS that are not present in other packages. These include process queues, asynchronous data transfers, and multiple synchronization mechanisms.

The primary tools include a subroutine library, callable from either Fortran or C, and a Job Manager program. The subroutine library provides mechanisms for interprocess communication, synchronization, remote subroutine calls, process selection, and job termination. The Job Manager starts and stops processes, handles errors, and manages global job information.

Parallel programming with CPS is often described as 'large-grained' and 'loosely coupled'. Large-grained means that the components running in parallel are typically entire subroutines or programs. Loosely coupled describes a set of processors which run relatively independent of each other on a network where communications are slower than in more tightly coupled processors and memory is distributed. Because of this, CPS is best suited for applications which have relatively low I/O requirements compared to CPU requirements. This is in contrast to parallel programming in which a few machine instructions or a few lines of code are executed in parallel, such as in vector processing with parallelizing compilers. Programs of this type typically require shared memory.

CPS supports several models of parallel programming. The client/server model is supported with remote subroutine calls. Processes may also interact with each other through synchronization, message passing, and bulk data transfers.

Each process in a CPS job runs a program written by the user. These processes can be distributed in any way among the computers on the network. For example, consider a ten process job. All processes could run on a single CPU¹ or they could be spread out over ten CPUs. You could also run the job on five CPUs, with two processes sharing each CPU. Apart from considerations of speed, the job will run the same. From a hardware point of view, the processes in a CPS job run on a set of RISC workstations and communicate with each other using TCP/IP. From the user's programming point of view, the job is a

1. In this paper, CPU is a synonym for computer or workstation.

network of fully interconnected processes. Each process can talk to another in the same way regardless of where the two are physically located.

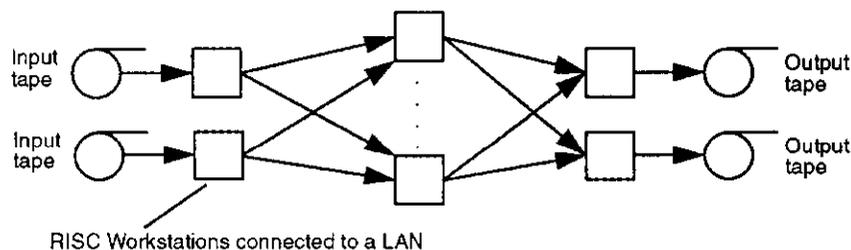
When a program is converted to CPS, the job is split among many processes. To achieve parallelism, many processes can perform the same function. However, some of the processes can perform functions different from others. A group of processes which run the same program on the same type of CPU is called a class of processes. In a CPS program, the job is divided among one or more classes of processes, each of which performs a separate function.

To design a CPS application, one must first split the job into separate functional units (or classes) and then decide how many processes each class should run. CPS programs are typically designed such that the number of processes in one class is adjustable. This way the job can expand to fill the available number of CPUs. The programmer must also decide on what type of machine each class will run, and the flow of data and control among processes.

For example, in a high energy physics event reconstruction program, events are read from tape, reconstructed, and then selected events are written to an output tape. An event is basically a digitized recording of tracks left by particles after a collision between two or more particles. Each event might be 10,000 bytes of data and each event is completely independent of all other events. A CPS program which performs the task of event reconstruction might be divided into three classes. The first class reads events from the input tape and sends them to processes in the second class, which do the event reconstruction. There is only one process in the first (input) class, but there can be many processes in the second (reconstruction) class. When a process in the reconstruction class has reconstructed the event, it may or may not send the data onward to the third class, which writes the event to tape. Like the input class, the third (output) class has only one process. Figure 2 shows this topology.

It is possible to increase the number of input and/or output processes. Figure 4 shows the topology of a CPS job with multiple input and output streams.

Figure 4 Event Reconstruction with Multiple Input/Output Streams

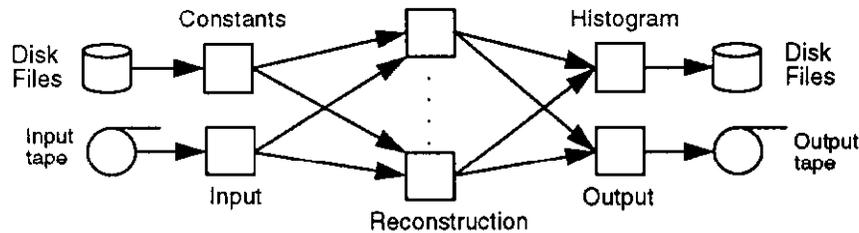


CDF, an experiment at Fermilab, has implemented their event reconstruction program with one input stream and multiple output streams. Most other experiments at Fermilab follow the standard event reconstruction topology in Figure 2.

In high energy physics (HEP), event reconstruction programs typically need to read in data files containing constants which are used in the reconstruction of each event. These data files only need to be read once, at the beginning of the job. It is possible to program each reconstruction process to read these files before accepting events. But this means the files would have to be distributed among all CPUs on which reconstruction processes run. If NFS is used, all reconstruction processes would be accessing the same files at once. An easy way to resolve this is to create another process to act as a constants server. It first reads the constants files and then acts as a server accepting requests (from reconstruction processes) to fetch the data. This way the disk is accessed only once by the constants server.

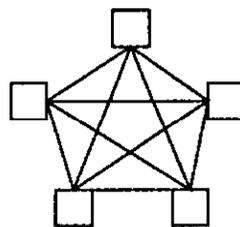
Also, event reconstruction programs typically produce histograms that are written to disk files. One could create another process to serve as a histogram builder. It would accept remote subroutine calls from reconstruction processes to add to the histogram. At the end of the job, one of the processes, either the input or output, would call another remote subroutine in the histogram process to write the histogram to disk. Figure 5 shows the topology of a CPS job with a constants server and a histogram process.

Figure 5 Event Reconstruction with Constants & Histogram Servers



CPS imposes no constraints on the topology of a distributed program. Any process may communicate with any other, as shown in Figure 6.

Figure 6 Fully Connected CPS Processes.



3. CPS Subroutines

This section describes the functionality provided by the CPS subroutine library. The names of CPS subroutines are printed in italics.

Process Initiation and the Job Manager

The Job Manager (JM) is the first process started in a CPS program. The JM is a program that is part of the CPS software, and is not in any class. A Job Description File (JDF) is supplied as input to the JM. The JDF tells how many processes of each class to start, what types of CPUs to use, and possibly which CPUs to use. Based on this, the JM will start user processes (processes which run programs written by the user) on remote CPUs and assign each a logical process number. If the JDF does not explicitly tell where to put each process, the JM will assign processes to CPUs in a way such that the load is reasonably well balanced among the CPUs in the farm. Each user process must first call *acp_init* to complete the start-up protocol with the JM. Once all processes have started, the JM provides central services, such as synchronizing processes and maintaining process queues (to be discussed later). The JM will also receive all output written to standard output from user processes. This output will be received on TCP sockets, and is written to both the JM's standard output and to a log file. Output produced by a user process is prepended with the process number to identify the sender.

Process Termination and Program Crashes

A CPS program which has run successfully to completion ends when one process calls *acp_stop_job*. When this happens, the JM kills each remaining user process and then exits. Processes which act as subroutine servers remain alive until killed by the JM. In other types of processes, it may be necessary to synchronize with the process calling *acp_stop_job* to avoid being killed by the JM prematurely. For example, in the topology shown in Figure 2, the reconstruction processes are servers and will run until killed by the JM. The input process however, should wait until all events have been flushed to the output, and then tell the output process to stop the job.

Processes which have finished their part in a job can call *acp_stop_process* to terminate without ending the job. Processes which have finished, but wish to sleep until the end of the job can call *acp_sleep_process*. The difference is that with *acp_sleep_process*, other processes can still access the sleeping process's data. For example, the constants server in the previous example might call *acp_sleep_process* after it has read the constants. Other processes can retrieve the data by calling *acp_get*, which is described later.

It is also possible to declare in the JDF how many processes in a class need to crash before the job is aborted. Once a class has too few processes remaining alive, the JM will abort the job. This is an optional feature that allows the user to terminate a job which has lost a number of CPUs because of crashes or network problems.

Process Synchronization

Processes may synchronize with each other in a number of different ways. The simplest, most straightforward way is to define a synchronization point (or barrier) which all processes must reach before continuing. CPS allows the user to define the set of processes which must reach the synchronization point before continuing onward. Up to 128 different synchronization points may be active at any one time. Processes can wait at a synchronization point by calling *acp_sync*, passing a bit string identifying a set of processes and a synchronization number as arguments. The JM will notify each process (and *acp_sync* will return) when each process has reached the synchronization point.

Processes may also synchronize by waiting for a process queue to become empty or full. This will be described later under process queues. Another way of achieving synchronization is to exchange user-defined messages. This will also be described later under explicit message passing.

Transferring Data

CPS provides routines for transferring large amounts of data between processes. Each block of data that is to be accessed remotely by other processes must first be declared by calling *acp_declare_block*. With *acp_declare_block*, the data block is associated with an integer identifier. Other processes will reference the block using the identifier.

The subroutine *acp_send* is used to send a block of data to one or more processes. If multiple processes are receiving the data, the data is not sent using internet broadcasts. Instead, since TCP sockets are used, the data is sent individually to each process in the set. Processes on the receiving end of an *acp_send* will receive the data in the background (at the signal handler level) while the process is running. It may be necessary to synchronize the sending and receiving processes to avoid overwriting data which the receiving process may be using.

The subroutine *acp_get* may be used to retrieve a data block from one or more processes. If multiple processes are specified in the subroutine call, the data is summed (as 32-bit integers) in the destination array. In terms of efficiency, *acp_send* and *acp_get* are nearly equal.

Remote Subroutine Calls

One way in which processes can communicate with each other is through remote subroutine calls. A process which is going to act as a remote subroutine server must declare each subroutine it intends to serve by calling *acp_declare_subroutine*. This will associate an integer identifier with each subroutine which other processes can use to call it. The remote subroutine's arguments and the byte count for each argument is also declared in *acp_declare_subroutine*. All arguments are treated as 32-bit integers when passed to the subroutine server.

An optional subroutine, *acp_declare_arguments*, can be used to declare the type of argument passing desired. By default, arguments are read-write. In this mode, the arguments in the remote subroutine server are initialized with the arguments passed, and they are copied back when the subroutine returns. Arguments can also be declared read-only or write-only. In write-only, the arguments at the server are initialized to zero, but the values are copied back when the subroutine returns. In read-only, the argument values are copied to the server, but are not copied back.

Processes can call remote subroutines in three ways. If a remote subroutine is called synchronously, the calling process will wait until the subroutine returns. If the subroutine is called asynchronously, it will return immediately, before the subroutine completes. The arguments will not be copied back regardless of how they were passed. The third method is to call the subroutine asynchronously, but when the subroutine returns, it will put the arguments and the process number onto a process queue. Process queues will be described shortly.

Processes can become dedicated remote subroutine servers by calling *acp_service_calls* after all subroutines have been declared. This will cause the process to sleep forever and wake up only to service remote subroutine requests. Multiple remote subroutine calls to the same process will be queued and serviced in the order in which they arrive.

Process Queues

Process queues provide a simple mechanism for selecting among processes that are in a particular state. A process queue contains a set of processes that are in some user-defined state. The process itself is not stored on a queue, only the process number. A process can put itself onto a queue, or be put onto a queue by another process. Queuing or dequeuing a process involves sending a message to the JM, which keeps a record of the processes on each queue. Processes are dequeued in the order in which they were queued. Processes are typically put on queues when they are ready to service a remote subroutine call,

receive a block of data, or receive an explicit message.

In CPS, there are 128 queues to which the user can assign meaning in any way. For example, queue #12 might hold all idle processes, while queue #20 is for processes holding 'special' events and queue #21 is for processes holding 'normal' events.

The subroutine *acp_queue_process* can be used to put a process onto a queue. This does not cause the process to stop running. The only thing that happens is that the Job Manager enters the process number onto the specified queue within the JM's internal queue data structures. The subroutine *acp_dequeue_process* can be used to get a process number from a specified queue. If there are no processes on the queue, the call will block until a process is queued. *Acp_dequeue_if_possible* can be used to return immediately if there are no processes on the queue.

Before a process can be queued, it must call *acp_declare_queue* to declare that it is eligible to be put on a specified queue. A common misconception is that this routine creates a queue, but this is not true. All 128 queues exist before the job is started. A queue is full when all processes that are eligible to be put onto the queue are on the queue. The subroutine *acp_wait_queue* can be used to block until a queue becomes full. It can also be used to block until a queue becomes empty.

Processes can be queued with a list of arguments. The number of arguments, and the number of bytes for each can be declared in the call to *acp_declare_queue*. This makes it easy to pass information between the queuing process and the dequeuing process. Queue arguments are commonly used in a context called 'call-and-queue'. When a process makes a remote subroutine call, it does not wait for the subroutine to finish, but instead specifies a queue onto which the server is directed to place itself and the remote subroutine arguments when the call returns. A process in another class can then dequeue the process and receive the return arguments of the remote subroutine call. In effect, the remote subroutine call 'returns' in another process.

A process can also put a special end-of-queue marker onto a queue. When this happens, no more processes are allowed onto the queue. The end-of-queue marker is commonly used to pass job termination information. For example, the output process in the standard event reconstruction topology will dequeue processes and write output until an end-of-queue marker is dequeued.

Explicit Messages

An explicit message is a short packet of data sent from one process to another. An explicit message

contains a message type (from 1 to 20) and up to 2048 bytes of information. *Acp_transmit_message* can be called to transmit a message to another process. *Acp_receive_message* can be used to receive a message of one or more specified types. This call will block until a message matching one of the desired types arrives. Calls to *acp_transmit_message* never block. If there is no receiver, the message will be stored internally in a queue (not a process queue) and delivered when a receiver arrives. *Acp_check_message* can be used to check if a message of one or more specified types has arrived.

Asynchronous Data Transfers

When *acp_send* or *acp_get* is used to do a data transfer, the call does not return until the data transfer is complete. In order to increase the throughput of CPS programs, routines have been provided in which a transfer could be initiated, but control would return immediately to the user's program. Also, in most cases the other process in a data transfer is obtained from a call to *acp_dequeue_process*. If the queue is empty, the dequeue operation blocks until a process is queued. CPS supplies asynchronous data transfer routines that combine the dequeue operation with the data transfer. Typically the way this works is that both the queue number and the arguments for a data transfer are specified in one subroutine call, which initiates an asynchronous dequeue operation and returns immediately. When a process is eventually dequeued, the data transfer is initiated. This all occurs in the background. When the data transfer is complete, the process is notified in some way. The notification method depends on the specific subroutine call.

The asynchronous data transfer routines were first implemented specifically for processes which do tape I/O. These processes typically read some data from tape, send it over the network to another process for analysis, and then go back to the tape to get some more data. The same is true for processes writing to tape, which get data from the network, write to tape, and then get more data from the network. In this synchronous way of running, the overall throughput of the CPS job will be some fraction of the data rate of the slowest device. If both network and tape data rates were equal, the overall throughput would be half the tape data rate. If the network throughput dramatically increased, the overall throughput would approach the tape data rate.

At the present (at Fermilab) the tape drive data rate is slower than the network. After each tape read, there is a delay to send the data across the network before the next tape read is started. If the time delay between successive tape reads/writes is too large, the tape drive may leave streaming mode and enter start/stop mode which can drastically reduce overall throughput. The asynchronous data transfer

routines were designed to solve this problem.

To write a CPS program with asynchronous data transfers, some sort of buffering scheme is required. A circular buffer is the solution we have chosen. After each buffer is filled with data from tape, a network send is initiated which will complete in the background. Tape reading can be continuous until a buffer is reached where the network send has not yet completed.

CPS provides two sets of subroutines for asynchronous dequeue and transfer operations. One set is called high-level and the other low-level. High level routines include *acp_buffer_index*, *acp_start_send*, *acp_wait_for_data*, *acp_data_ready*, *acp_start_gets*, *acp_handle_data*, *acp_end_of_data*, and *acp_buffer_set_wait*. Low level routines include *acp_queue_xfer*, *acp_dq_get*, *acp_dq_send*, and *acp_wait_for_get*. The high level routines implement a circular buffering scheme for the user. The low level CPS asynchronous routines leave the buffering to the user. The high level CPS asynchronous routines call the low level routines to do their work.

4. CPS Implementation

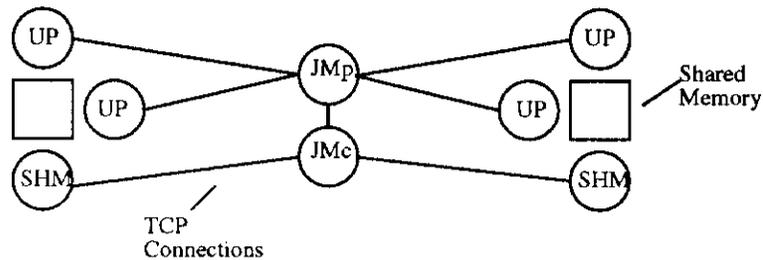
This section describes the underlying architecture used to achieve fast, reliable, and fault tolerant communications between CPS processes. A job is started by first starting the JM, which reads the JDF to find out where to start processes and which program each process is to run. The JM then starts the User Processes (processes which run programs written by the user) on all the nodes. When a User Process starts, its standard input, output, and error are all assigned to a single TCP socket connected to the JM. These connections will be alive during the entire job so that the JM can collect terminal output from User Processes. The JM then forks a process. The child process starts a daemon called the Shared Memory Manager (SHM) running on each system where there are User Processes. Only one SHM per system is started, and TCP connections between the JM child process and the SHM remain active for the duration of the job. The JM child process maintains the process queue data structures and other global job information, such as process status information and which processes have reached synchronization points. The JM parent process only collects terminal output from User Processes.

Each SHM creates an IPC shared memory and an IPC set of semaphores to be used for communications between processes on the same system. The shared memory contains a global process map, message buffers, and message queue structures. Each shared memory will have one queue structure for each User Process on the system, one for the SHM, and a free queue. For example, if two User Processes share a CPU, the shared memory will contain four internal queues. Semaphores are used to synchronize

write access to the queue structures. When the job begins, all message buffers are queued on the free queue. A User Process can send a message to another User Process by dequeuing a buffer from the free queue, depositing the message into the buffer, queueing it on the destination User Process's queue, and sending a UNIX signal to the destination User Process. This involves two semaphore operations, two memory copies, and one signal operation. What we have so far is shown in Figure 7, which shows a job with four User Processes distributed on two CPUs.

Figure 7

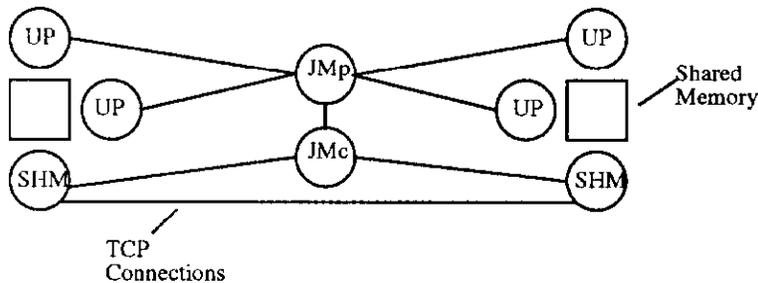
CPS Job with Four User Processes. on Two CPUs. (JMp = Job Manager parent, JMc = Job Manager child, UP = User Process, SHM = Shared Memory Manager)



When a User Process wishes to send a message to the JM or to a User Process on another CPU, a message buffer is first obtained from the free queue, the message is deposited in the buffer, stamped with an address, and queued on the SHM's queue. If the message is intended for the JM, the SHM sends it over the TCP connection to the JM, otherwise the SHM creates a TCP connection with the SHM on the remote CPU where the destination User Process is located. The message is sent to the remote SHM, which delivers it to the destination User Process. Any TCP connections between SHMs created in this way are kept alive for the duration of the job for future possible messages. Figure 8 shows a CPS job with a TCP connection established between two SHM processes.

Figure 8

CPS Job with SHM to SHM TCP connection.



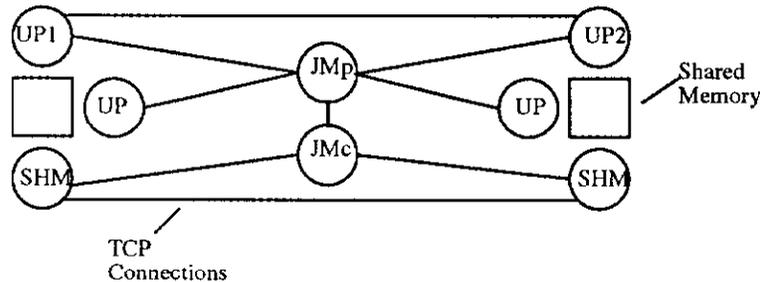
This method of message passing has proven to be very reliable in production. Messages which use this method include remote subroutine calls, synchronization, queueing and dequeuing, and explicit messages. Bulk data transfers (*acp_send*, *acp_get*, and asynchronous data transfer routines) are

accomplished in a different way.

If a User Process transfers data to/from another User Process on the same CPU, the data is broken up into 2K chunks and passed through the shared memory in the normal message passing fashion. If however a User Process (UP1) transfers data to a User Process (UP2) on another CPU, the following occurs. A control message is sent to UP2 telling it to establish a TCP connection with UP1. The control message is sent through the Shared Memory Managers using the method described above. A TCP connection is then established between UP1 and UP2. The User Process with the lower logical process number always accepts a connection request, and the User Process with the higher logical process number always initiates the connection. With the TCP connection established, instructions for the data transfer are sent through the TCP connection from UP1 to UP2. The receiving process will then send a 'go-ahead' message (over the TCP connection) to the sending process. The entire block of data is then sent over the TCP connection to the receiving process. See Figure 9.

Figure 9

CPS Job with TCP connection between two User Processes



The TCP connection established between the two User Processes is not shut down after the transfer completes. Any subsequent transfers will use the connection already established. It is possible to limit the maximum number of TCP connections a User Process can have open at one time, but so far this has not been necessary.

The overhead associated with this method of transferring data is the initial control message. In the case where a TCP connection between User Processes is not yet established, the overhead also includes the time to establish the connection. The performance of this method of data transfer approaches the maximum ethernet throughput (using TCP) as the size of the data block transferred increases.

The question of using UDP rather than TCP is often asked. UDP is faster than TCP and has less overhead, but it is not a reliable byte-stream protocol, like TCP. TCP was chosen because CPS requires reliability, and also because it is common to transfer large amounts of data. UDP is better suited for

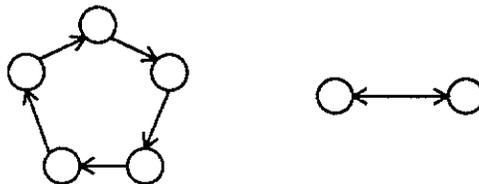
applications in which single messages typically fit in one UDP packet, and reliable message passing is not required.

The Job Manager (JM), which provides centralized services, does not affect the scalability of a CPS program, because data transfers, subroutine calls, and explicit message passing do not involve the JM. The JM is used only for process queueing and synchronization. The benefits of simplicity and increased functionality outweigh the disadvantages of a central control point.

There are two possible deadlock situations, and both are handled in similar ways. The first deadlock situation involves TCP connection establishment. To establish a TCP connection, one process must initiate the connection, and another must accept. There will be a deadlock if two processes simultaneously attempt to initiate a connection with each other. This is solved by forcing the lower numbered process to always initiate the connection request, and the higher numbered process always accepts a connection request, regardless of who is the sender, receiver, or initiator of the data transfer. The other deadlock can occur after the connection has been established. This is the classical circular deadlock problem shown in Figure 10. Simultaneously, processes try to send to each other forming a loop where everybody is sending, but nobody is reading. In the case with two processes, the solution is to give the 'right-of-way' to the process with the lower process number, which may send the data while the other must wait. In the general case, each process handles the data transfer with the lowest process number first, regardless of which process initiated the transfer. These are not fair solutions, but deadlock situations are typically rare occurrences and the method works.

Figure 10

Circular Deadlock. In the general case, processes form a ring where all are sending and none are receiving. In the simple case, two processes simultaneously send to each other.



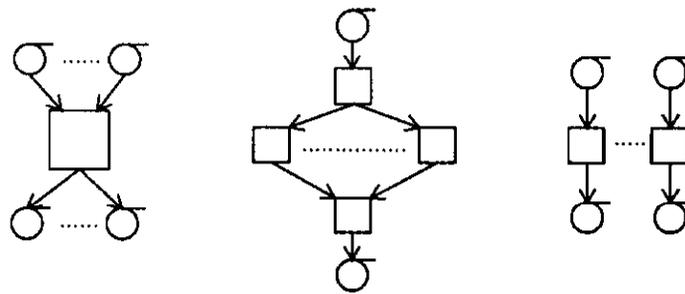
There are certain advantages to having a Shared Memory Manager on each CPU. The SHM periodically checks to see if each User Process is still alive. If one crashed, the JM is notified, and the appropriate action is taken. The SHM is also responsible for stopping processes when a job finishes or aborts, and cleans up shared memories and semaphores, which are persistent system objects.

5. RISC Processor Farms

RISC processor farms are a cost-effective way of satisfying the computing requirements of high-energy physics event reconstruction. HEP jobs are characterized by a large amount of input and output, and an extremely large amount of computing. The farms solution is gaining more popularity and getting more attention because other fields have computing problems with similar requirements. The farms solution is a compromise between two other possible solutions: the supercomputer solution and the classic batch solution. See Figure 11.

Figure 11

Three Computing Solutions. From left to right: supercomputer, farms, classic batch. Farms are characterized by an expandable compute segment.



The first solution is to buy a mainframe supercomputer or possibly a few of them. The supercomputer will run many user's programs simultaneously, each of which reads and writes to disk or tape. The problem with this is that it is much too expensive. For our particular problem, the CPU might be saturated, but much of the I/O bandwidth may go unused. Of the three solutions, the cost/MIPS ratio of the supercomputer is by far the highest.

The second solution is classic batch. This involves buying a moderate number of RISC workstations, attaching disk and tape to each, networking them together, and sending jobs to them through a networked queueing system. Each workstation has one to a few batch queues which accept job requests from other workstations on the network. The larger number of peripherals and the increased system management overhead make this a more expensive solution than a farm. System maintenance includes both operating system administration and hardware maintenance. A less obvious problem is that each user's job may take an extremely long time to run. This may reduce the percentage of jobs which execute to successful, normal completion. On a single workstation, a typical event reconstruction program may take a week or longer to process all the events on a single 8mm tape.

The third solution is to build a computer farm. One part of a farm is characterized by a large number

of workstations, all connected to a local-area network and each equipped with only a system disk. These are called compute servers because the task of each is to accept relatively small amounts of data, do a large amount of computing, and return the result, which is also a small amount of data. The other part of the farm typically consists of a few workstations called I/O servers. This is where all the peripherals are located. Each I/O server may serve anywhere from a dozen to four dozen compute servers. The number of compute servers per I/O server depends on many things. Some are listed below.

- Network throughput.
- MIPS of compute server compared to I/O server.
- Job mix - number of instructions executed per byte of I/O.
- Tape and/or disk speed.

Figure 11 shows only one possible job topology. Any topology is possible, but typically one class of processes is made scalable to fit the number of compute servers available. Processes in a job can map to CPUs in any way. The event reconstruction program running on Fermilab farms typically map both the input and the output process to the same I/O server, and one or two reconstruction processes are assigned to each compute server.

The advantages to a farm configuration are many. Cheaper MIPS can be obtained by buying RISC workstations for compute servers. Operating system management for the compute servers can be automated. That is, a change can be made on one and 'broadcast' to all others. It is most cost effective to have as few I/O servers as possible. The goal is to drive as many compute servers as possible from a single I/O server. This may require a substantial amount of computing power, especially if many jobs share an I/O server, but do not share compute servers. The reason for minimizing the number of I/O servers is to reduce the amount of system management required, and to keep the number of peripherals, especially tape drives, to a minimum.

Each job will take much less time to complete on a farm than on a single workstation, as in the classic batch solution. This is because each job on a farm has a much greater amount of CPU available to it. Of course, fewer jobs run simultaneously in a farm system compared to a classic batch system, so the overall throughput may be the same. However, the farm solution is less expensive because of the smaller number of peripherals. Also, because each job in a farm takes less time to complete, a higher percentage of jobs will finish without a glitch, especially if the system is fault tolerant enough to let a few compute servers crash without affecting the job. A glitch refers to events such as network congestion and/or failures, tape drive malfunctions, media corruption, power failures, etc. This is important since an event reconstruction job may take a week or longer to complete on a single workstation, but only a day or less on a farm.

Fermilab Farms

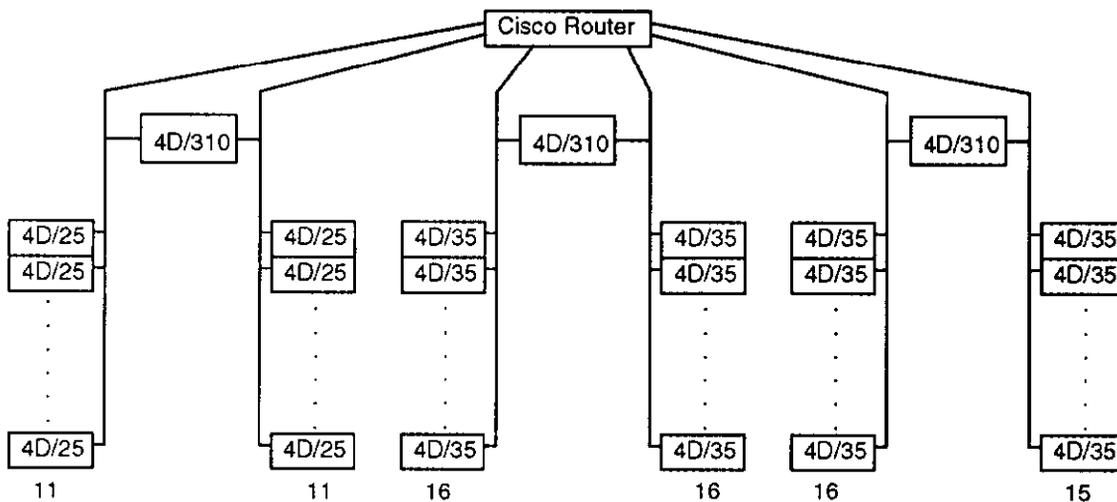
CPS supports farms of UNIX RISC-based workstations connected to any network running the TCP/IP suite of protocols. CPS has been ported to the following UNIX workstation/operating system flavors:

Silicon Graphics	IRIX
IBM RS6000	AIX
DEC	Ultrix
SUN	SUNOS
Hewlett Packard	HP/UX
MIPS	RISC/OS
Fermilab	ACP R3000 Processor Board running RISC/OS

Fermilab currently runs farms of two flavors - IBM AIX and Silicon Graphics IRIX, and provides extensive support for these. The compute servers are rack mounted in tiers, each containing seven or eight workstations. Each compute server has at least 16 MB of memory and a local disk with the operating system and paging space. Systems can be configured to match a target application's requirements. At present, Fermilab farms are configured into 'farmlets', each with one I/O server and approximately 16 compute servers on one ethernet segment. At Fermilab, an arbitrary restriction is imposed - processes in a CPS job are confined to a single farmlet. Each farmlet is separated by a bridge or router to isolate CPS network traffic from the rest of the network. The Silicon Graphics farms are shown in Figure 12 and the IBM farms in Figure 13 (in their present configuration).

Figure 12

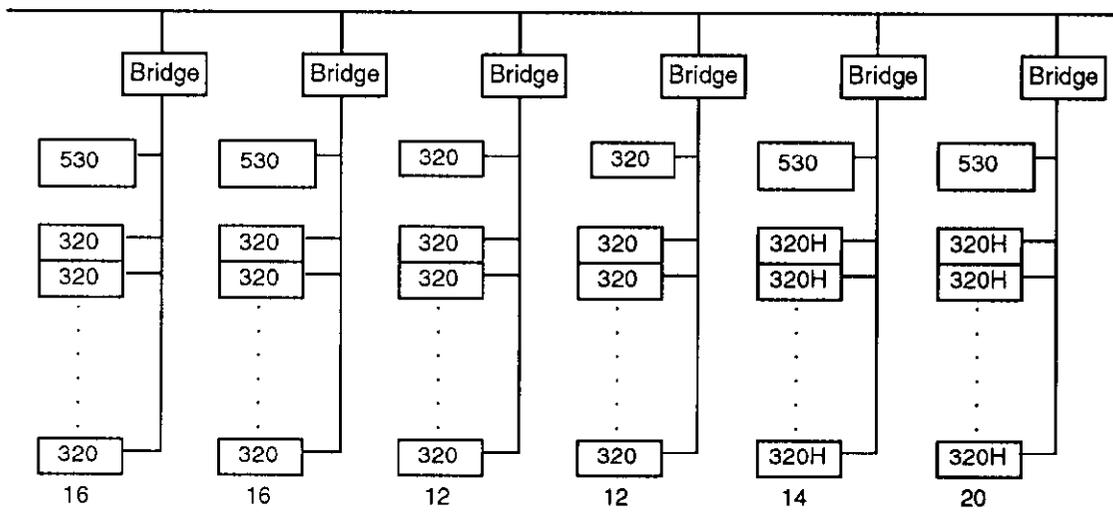
Fermilab Silicon Graphics Farms. The I/O servers are 4D/310 machines, compute servers are 4D/25 and 4D/35 machines. The number of compute servers in each farmlet is shown at the bottom.



Each I/O server is equipped with 1-8 GB of external SCSI disk and 3-9 SCSI exabyte 8mm tape drives.

The amount of disk and tape allocated to each I/O server depends on the requirements of the applications assigned to each farmlet. Each I/O server's disks are NFS mounted on all compute servers in its farmlet. This provides access to user and CPS executables as well as scratch space for batch jobs. Network traffic outside a farmlet is limited to activities such as user login and operator assisted tape mount requests and replies.

Figure 13 Fermilab IBM Farms. The I/O servers are the machines closest to the bridges. The number of compute servers in each farmlet is shown at the bottom.



6. CPS Batch and Computer Center Operations

CPS Batch is a batch system specifically tailored to run CPS programs. It provides job queuing, resource allocation, and operator communications services.

The resources that make up a farm, such as CPUs, disks, and tape drives, are kept in a network-wide database. The database is essentially a collection of inter-related data structures that is maintained and manipulated by a server process called the Production Manager, or PM for short. Clients programs can connect to the PM to view the database or to make insertions, deletions, or modifications. All updates are immediately written to disk, and when the PM is started at system boot time, the database is initialized from the disk file. The PM server typically runs on a system that is rarely down.

Resources within a farm can be grouped together to form logical computers, or to use Fermilab jargon, production systems. A production system contains a subset of the total set of resources in a farm. For example, production systems defined on the Fermilab farms typically consist of an I/O server, a farmlet of

11-16 compute servers, and some disk and tape drives located on the I/O server. Note that a farmlet is just a physical part of a farm that has been bridged off to keep network traffic local and to prevent outside traffic from affecting it. CPUs can be shared between production systems. This is not true of tape drives, which can only be allocated to one system at a time.

Each production system is given a name and has a batch queue and a list of users allowed to use it. CPS batch jobs submitted to a production system will run one at a time and in the order submitted. Processes (in the CPS batch job) are assigned to CPUs within the production system. The working directory for the batch job will be located on the disk assigned to the production system. If necessary, the CPS Batch system will copy the user's executables and data files to the working directory before the job starts. (This is not necessary if these files can be remotely accessed with NFS.) The production system definitions, as well as the job queues, are also stored in the database by the PM.

CPS Batch includes utilities for submitting, aborting, and cancelling jobs. Utilities also exist for examining, creating, and updating production system definitions. These utility programs all communicate with the PM server to do their work.

The operator communications subsystem consists of an X-windows operator console and utilities for sending tape mount requests to the console. Tape mount requests can be sent with a shell command or with a subroutine call. On the operator console, pending requests are shown as icons containing the name of the device, the label of the tape, and the write protection required. The operator points and clicks on the icon and chooses an appropriate response from a pop-up menu. If 'success' was the response, the tape request software checks to see if the tape was mounted properly, i.e., there is a tape in the drive and the write protection is correct. If not, the mount request and an error message are sent back to the operator console.

The operator console also monitors the PM server, showing a smiling face if it is running and a tombstone if it isn't. Batch jobs can also be monitored from the operator console. The intent is to make the operator console able to display problems as they arise and supply enough information so that the operations staff can make an intelligent decision to fix the problem. It is also intended to supply the means to fix common problems from within the operator console itself. For example, if a tape drive is found to be broken, it can be removed from the production system and replaced with a spare drive already on the I/O server but not currently assigned to a production system.

The CPS Batch system also supplies a program for receiving tape mount requests on an ASCII terminal

and a program for responding to tape mounts from the command line.

7. Performance

A typical experiment (but by far not the largest) at Fermilab has approximately a thousand 8mm tapes of raw event data that need to be reconstructed. It is therefore very important to obtain the highest possible overall throughput. Consider Figure 2. We define the aggregate throughput as the sum of the rate at which data is read from the input tape and the rate at which data is written to the output tape. There are many possible bottlenecks. The first is obvious. With one input tape and one output tape, the maximum aggregate throughput is the two tape data rates in streaming mode. Another possible bottleneck is the network. How fast can data be transferred from an I/O server to a compute server, and how does this compare to the combined tape data rates. In the Fermilab farms, the fastest an 8mm tape can read or write is approximately 240 KB/sec. The two tapes combined are 480 KB/sec, which is well below the 10 Mbps limit of TCP on ethernet.

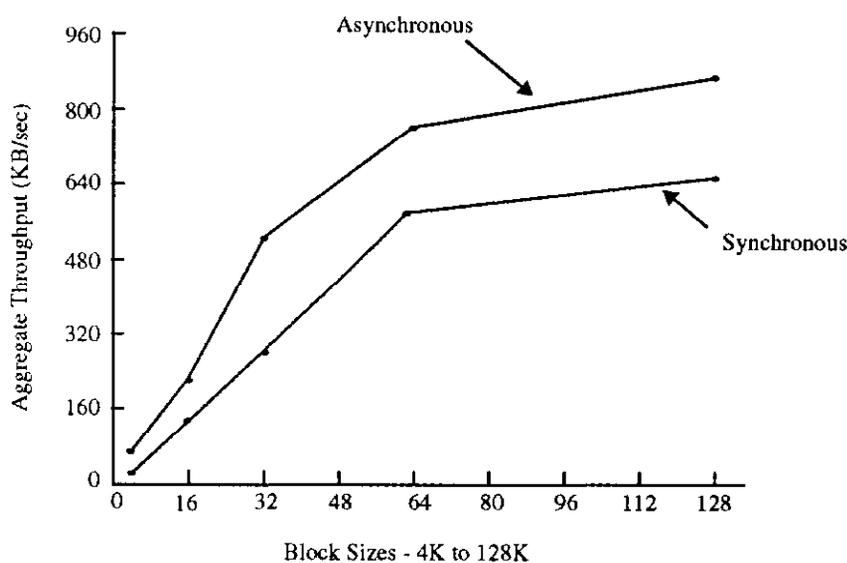
One feature of computer farms is that the bottleneck can often be a combination of several things. For example, in our experiences we have had the following problem. The I/O servers became heavily loaded because each was shared by several experiments. Also, each experiment was running in synchronous mode. This means that each tape and network I/O operation is performed synchronously. An event is read from tape, then sent over the network, then another read from tape, etc. The problem is that if the load on the I/O server is too heavy, and/or the network is too congested, subsequent reads from the tape may be spaced too far apart in time. If this happens the tape drive could enter start/stop mode which drastically reduces overall throughput. The solution to this problem is a combination of two things. First, either the CPU power of the I/O servers can be increased, or the load on them decreased. Second, applications that use tape should take advantage of the asynchronous CPS routines. By doing this, network I/O operations are completed in the background and the next tape read or write can be started immediately. It should also be noted that using asynchronous I/O may increase the load on the I/O server.

The CPS software itself also plays a role in performance. Two things need to be considered. The first is the amount of CPU required by the Shared Memory Manager (SHM), which runs on each CPU. Benchmarks have shown SHM CPU usage to be ten percent or below. The SHM uses around ten percent of the CPU in jobs that are I/O bound which send data in small blocks (4K or less). Less CPU is used by applications that send data in large blocks (64K-128K). The second consideration is the efficiency of CPS in message passing and transferring data. Data transfers involve a constant overhead, regardless of block

size, which is the time required for the two processes to complete the 'handshake', where one process agrees to send data and the other agrees to accept data on TCP connected sockets. As the amount of data per transfer becomes larger, the overhead becomes less and less significant, and the CPS data transfer rates approach the TCP 1 Mbps limit on ethernet. Benchmarks have been run on the Silicon Graphics farm at Fermilab to test the aggregate throughput of CPS jobs with varying block sizes. The block size is the amount of data transferred per CPS subroutine call. The benchmark program ran with both the input and output processes on a single 4D/310 and with 16 reconstruction processes, each assigned to a separate 4D/35. The benchmark did not use tape drives. Input was generated by the input process and swallowed by the output process. The results are shown in Figure 14.

Figure 14

CPS Sustainable Aggregate Throughput (KB/sec)



8. Conclusions and Future Work

CPS has been in production-level use at Fermilab since late 1989. At the present, nine experiments at Fermilab use CPS and CPS Batch in a production environment. This means that each of these experiments are running CPS Batch jobs twenty-four hours a day, seven days a week. CPS is also used by the Superconducting Super Collider Laboratory as an integral part of the Physics Detector Simulation Facility. It is also used by an experiment at the National Institute of Nuclear Physics in Bologna, Italy, and by a group at the University of Michigan. Because of the large amount of use, CPS has become very robust. Work is underway to formalize the distribution mechanisms, software releases, and user support channels for CPS, assuming funding for this can be obtained.

The farm solution has caught the attention of other fields with computing requirements similar to HEP's event reconstruction. A collaboration between Fermilab, IBM, and Merck is under way to apply CPS and farms to the problem of rational drug design. This will be a first test of usage in an industrial research environment. The farm solution is not a replacement for a supercomputer. It is an architecture originally designed to solve a specific problem. The main characteristic of this problem is that it is very compute intensive, with each byte of I/O requiring 500-2000 machine instructions. A farm is a very cost effective solution for this problem, and as it is turning out, many other problems with similar characteristics.

It should also be pointed out that a farm is not merely a collection of networked computers. The way they are used and managed sets them apart. All the processors in a given farm work collectively on a single job. Also, processors in a farm are managed 'in parallel'. This means that a modification made to one system is broadcast to all others.

Work on CPS and CPS Batch is by no means finished. Tools to make application debugging easier are being considered. Tools for monitoring CPS Batch operations as well as interactive CPS jobs are currently under development. Both ASCII and X-windows interfaces will be included. Work is also in progress to add dynamic load balancing and load sharing to CPS Batch. If a farm is idle and a job is submitted, the job should be able to use all the CPUs. As jobs are submitted to other batch queues which use the same farm, processes in running jobs can be shutdown to make room for processes in the new job. Also, if a job finishes, other jobs can expand to fill the vacant CPUs. In the current batch system, the farms are divided into static production systems. A system with dynamic load balancing can better utilize the available computing cycles of a farm.

Acknowledgments

I would like to thank Frank Rinaldo, Steve Wolbers, and Tom Nash for their help in preparing this paper. Many people have contributed to make the farm systems at Fermilab a success. Some of the individuals are Chip Kaliher, David Potter, Bob Yeager, Matt Wicks, Marc Mengel, David Oyler, and many others. Special thanks to experimenters such as Joel Butler, Paul Lebrun, Eric Wicklund, Brian Troemel, Mike Diesburg, and many others for thoroughly shaking out the bugs and being patient. The group that developed the original ACP, the R3000 processor board, and began development of CPS included Hari Areti, Bob Atac, Joe Biel, Art Cook, Jim Deppe, Mark Edel, Mark Fischler, Irwin Gaines, Rick Hance, Don Husby, Mike Isely, Mariano Miranda, Tom Nash, Think Pham, and Ted Zmuda.

References

- [Avery90] P. R. Avery, A. P. White, "UFMULTI - Microprocessor Control System for High Energy Physics", American Institute of Physics Conference Proceedings 209, pp. 395-406, 1990.
- [Biel90a] Joseph R. Biel et. al., The ACP Cooperative Processes User's Manual, Fermilab Computing Division, Document #GA0006, November 1990.
- [Biel90b] Joseph R. Biel, "Use of UNIX in Large Online Processor Farms", American Institute of Physics Conference Proceedings 209, pp. 302-308, 1990.
- [Fischler92] Mark Fischler, "The ACPMAPS System: A Detailed Overview", Fermilab Technical Memo #1780, May 1992.
- [Gaines87a] Irwin Gaines et. al., "Software for the ACP Multiprocessor System", Computer Physics Communications, vol. 45, pp. 331-337, 1987.
- [Gaines87b] Irwin Gaines et. al., "The ACP Multiprocessor System at Fermilab", Computer Physics Communications, vol. 45, pp. 323-329, 1987.
- [Geist91] Al Geist et. al., A User's Guide to PVM Parallel Virtual Machine, Oak Ridge National Laboratory/TM-11826, July 1991.
- [Gelernter90] David Gelernter et. al., "Adventures with Network Linda", Supercomputing Review, October 1990.
- [Kaliher90] Chip Kaliher, "Cooperative Processes Software", American Institute of Physics Conference Proceedings 209, pp. 364-371, 1990.
- [Litzkow88] M. Litzkow, M. Livny, and M. Mutka, "Condor - A Hunter of Idle Workstations", Proceedings of the 8th International Conference on Distributed Computing Systems, San Jose, Calif., June 1988.
- [Nash84] Thomas Nash, "The Fermilab ACP Multi-Microprocessor Project", Fermilab Technical Publication Conf-84/63, August 1984, originally presented at Symposium on Recent Developments in Computing, Processor, and Software Research for High Energy Physics, Guanajuato, Mexico, May 8-11, 1984.
- [Nash89] Thomas Nash, "Event Parallelism: Distributed Memory Parallel Computing for High Energy Physics Experiments", Proceedings of the International Conference on Computing in High Energy Physics, Oxford, April 10-14, 1989, also available as, Computer Physics Communications, vol. 57, pp. 47-56, 1989.
- [Zhou92] Songnian Zhou et. al., Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems, Technical Report CSRI-257, Computer Systems Research Institute, University of Toronto, April 1992.