NYMPH: A Multiprocessor for Manipulation Applications

J. Bradley Chen, Ronald S. Fearing, Brian S. Armstrong, and Joel W. Burdick
Stanford Artificial Intelligence Laboratory
Department of Computer Science
Stanford University

Abstract

The robotics group of the Stanford Artificial Intelligence Laboratory is currently developing a new computational system for robotics applications. Stanford's NYMPH system uses multiple NSC 32016 processors and one MC68010 based processor, sharing a common Intel Multibus. The 32K processors provide the raw computational power needed for advanced robotics applications, and the 68K provides a pleasant interface with the rest of the world. Software has been developed to provide useful communications and synchronization primitives, without consuming excessive processor resources or bus bandwidth. NYMPH provides both large amounts of computing power and a good programming environment, making it an effective research tool.

Introduction

The real time requirements of modern applications in robotics control require large amounts of computing power. Multiprocessor machines are well suited for these applications because they can economically and flexibly provide the large amounts of computing power required. Single processors of a multiprocessor system can manage single time critical tasks. By coordinating these processors, real time systems can be constructed. NYMPH (Not Your average Multiprocessor Hack) is a system being developed by the robotics group of the Stanford Artificial Intelligence Laboratory to meet the computational requirements of present and future robotics control applications.

The computational requirements of creating a pleasant and functional software environment for a robotics control processor are in direct opposition to the primary goal of keeping the computing power available to the servo loop calculations. In previous control systems, programmers have suffered immeasurable grief searching for clever ways to prevent servo calculations from being interrupted by I/O requests and other operating system tasks. Such interrupt mechanisms are good for systems in which the first concern is to communicate with people, but in real time robotics applications they often prevent the computers from keeping pace with the machines.

Using a multiprocessor architecture further complicates the problem of creating a pleasant user environment. It would be convenient to be able to control arbitrary applications running on all the processors from a single program or a single terminal, but the communications and synchronization problems involved in creating such an environment mandate large and complex structures to coordinate them, making the pleasant programming environment inefficient and difficult to build. Because of the inherent complexity of multiprocessor environments, and the need to preserve available computing resources for the application programs, manipulation multiprocessors tend to have somewhat uncivilized user interfaces.

In NYMPH, multiprocessor control structures have been integrated into an existing system, adding the processing power necessary for robotics applications to an already existing programming environment. While NYMPH's 32K computers provide real time computing power and high speed floating point computation, the 68K manages user interaction. The 68K is part of a Sun 120 computer, with an 800 by 1024 bitmapped display and high speed graphics capabilities. The 68K runs the V-System with the Virtual Graphics Terminal Server (VGTS) window system, developed by the Distributed Systems Group of Stanford University (Cheriton 1982). Using the V-System with the VGTS, the NYMPH programmer can have interaction windows for each processor, plus editors, terminal emulators, graphics capabilities, network access, and other useful facilities provided by the V-System. The researcher can edit files, test software on the multiprocessor, and analyze output with the aid of graphics, quickly and conveniently, all from the same console.

Previous Work

Computers using multiple microprocessors are becoming important as a cost effective solution to the computational demands of real time robotics applications. The 32K one board computers used in NYMPH provide 40% of the floating point speed of a Vax 11/780 (with a floating point accelerator) at 1% of the cost.

Much work has been done in the area of the feasibility and efficiency of manipulation multiprocessors (Nigam and Lee 1985, Zheng and Chen 1985). Research has also been directed towards the development of special system software to run on such machines (Schwan et. al. 1985, Siegel et. al. 1985).

A group of researchers at MIT (Siegel et al 1985) used five MC68010s with a DMA link to a DEC Vax 11/750 to control the Utah/MIT Hand. In this system, the Vax is used for user and file I/O and program development. This system also utilized a message-style communication system and synchronization by means of a servo-loop-scheduler routine.

Ozguner and Kao (1985) have designed a reconfigurable multiprocessor to control the Ohio State University hexapod walking machine. This multiprocessor uses 4 Intel 86/30 single board computers with fault detection and correction hardware, communicating on four busses. Error recovery using redundancy was investigated with this machine.

Architecture

NYMPH uses seven 32K single board computers to provide the bulk of its computing power. The 32K boards run at 10MHz and each board includes 32K bytes of ROM, 512k bytes of RAM, a floating point co-processor, an 8255A Programmable Peripheral Interface, and two serial ports. The 32081 floating point co-processor greatly enhances the performance of the system, enabling the 32K to do a floating point multiply in 6 microseconds.

NYMPH System

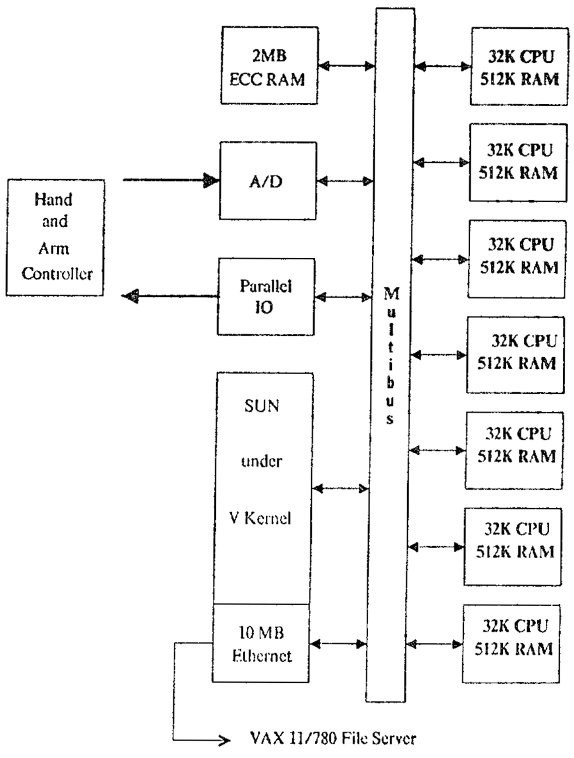

Figure 1: The Nymph System

The 68K provides the interface between NYMPH and the rest of the world. Since the 68K and the 32Ks are on the same Multibus, communication between the machines is as fast as a bus transaction. A 10MB/second Ethernet links the 68K to the Vax computers used for remote file I/O and program development. The dual ported memory of the 32K boards enables the 68K to load programs received over the Ethernet directly into the local memory of the 32Ks.

The onboard parallel port of the 32K boards is used to interrupt the 68K processor. The 32Ks' ability to interrupt the 68K helps the 68K to respond quickly to service requests. Since the 68K does not get any resources from the 32K, and since it is desirable for the 32K to be able to respond quickly to the demands of real time applications, the 68K can not interrupt the 32Ks.

NYMPH also has 2M bytes of EDC RAM, parallel I/O controllers to communicate with robot controllers, A-D converters to process force and tactile sensor data, and D-A converters to drive motors.

Software

NYMPH software is designed to avoid system overhead on the 32K processors. Their role is to control the arms and hands, and to bog them down with interrupts or a sophisticated operating system would deny processing power to the applications. For this reason, there is not an operating system that runs on each 32K, but rather a collection of runtime library routines. Two libraries which greatly improve the ease of programming NYMPH are the Communications library and the Synchronization library.

Communications

The 68K processor in the NYMPH system exists to service the runtime interaction requests of the 32Ks. In the NYMPH system, the 32K processors act as clients, and the 68K is the server, fulfilling the requests of the 32Ks.

Communication in NYMPH is based on a synchronous, typed, dependable message passing system, with which a 32K processor can send a 32 byte message to a V-System process running on the 68K, and receive a reply. The 32K uses the messages to make resource requests from servers running on the 68K. Since the 68K never needs to make a request of the 32Ks, the 68K cannot send a messages to them.

The message passing system used in NYMPH relies on the message passing primitives provided by the V-System. Each message sent by a 32K contains a V-System type message, in addition to other fields such as sender machine id, receiver process id, and message state.

A message transaction involves the following: A process on a National allocates memory for the message from its own local memory, usually compiled in as a variable declaration. After filling in the appropriate fields, including the type of the message and the contents of the 32 byte V message field, it calls `Send(Message, Receiver)`, where `Message` is a pointer to the message and `Receiver` is the V process id of the process to receive the message. The Send routine fills other necessary fields of the message, stores the address of the message in a global Message List, and increments the global variable `NextMsg`, an index into the Message List which shows where the address of the last message was written. To prevent collisions from multiple processors trying to send messages at the same time, the index variable `NextMsg` is protected with a global boolean protection variable using a test-and-set instruction. Access to the protection variable is arbitrated by the Multibus. After `NextMsg` is incremented, the 32K exerts an interrupt on the 68K. The 32K then polls in local memory, waiting for the V system message transaction to complete. Polling in local memory helps minimize bus utilization.

The interrupt from the 32K causes a V-System user process, the message server, to be readied. The message server compares the global variable `NextMsg` to an internal variable `LastMsg`, which is an index into the Message List that shows the last message dealt with. If they are the same, the message server goes back to sleep. If they are not the same, the message server sends the address of the message to the handler process corresponding to the sending processor. Separate processes for each processor promote a high level of concurrency. This is beneficial in a multiprogramming environment such as the V-System, especially since many of the messages are commonly I/O requests. The message handler performs byte and word order translations, then sends the message to the intended receiver and waits for a reply. After the V message transaction is complete, the message handler process notifies the 32K that its reply is ready by setting a 'replyed' bit in the message, and then blocks, awaiting a new message. Back at the 32K, the `Send()` call returns with the reply message where the original message had been, completing the message transaction. Figure 2 illustrates some basic mechanisms of the communication system.

Typed messages are necessary because of the different byte and word order conversions required. Several simple types are available including types for messages composed of bytes, words, and longs. There is also an "untyped" type, in the case that existing types are not suitable for the intended application.

One of the strengths of the system is efficiency. In the present implementation, a concerted effort was made to avoid unnecessary

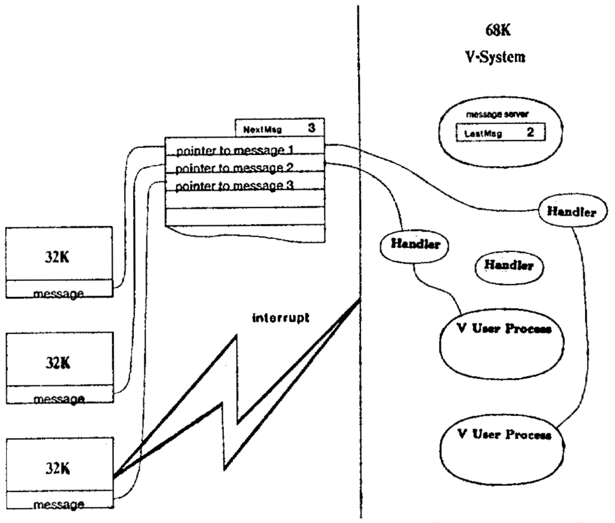

Figure 2: Message Passing Communication. *This figure illustrates two possible message states. Messages 1 and 2 have been sent by the 32Ks, received by V-System processes, and are pending reply. Message 3 has been added to the Message List, and NextMsg has been incremented, and the 32K is now interrupting the 68K to let the message server know that a new message is ready. When the message server wakes up, it will see that LastMsg is not equal to NextMsg, and create a child to manage the new message.*

copying of messages, though some copying is unavoidable because of the necessity of byte and word order translations. Interrupts between the processors also contribute to the speed and efficiency of the system.

One of the weaknesses of the message system is the need to do so much byte and word swapping. However, this is less of a problem in the message passing system, where the amount of data passed is relatively small, than in the applications that will be using it, such as the file I/O system. The heterogeneous processors were chosen because of the processing power of the 32K, with floating point speed which is very useful in manipulation applications, and for the good user interface and I/O support available with the 68K and the V-Kernel. However, if a second machine were to be built, it would likely use processors with the same byte ordering.

Synchronization

To support determinist execution of procedures distributed across several processors, synchronization primitives are included in the NYMPH programming environment. These primitives, `synch_signal(n)` and `synch_wait(n,patience)`, provide dynamic synchronization, data collection for critical path analysis, and a limited channel for message passing.

The central objective of the synchronization design was to provide a means for dynamically coordinating processes at run time: the programmer should not have to work out and embody in the programs a schedule of synchronization events. A processor indicates its desire to participate in a synchronization event by executing the command `synch_signal(n)`, where n specifies the synchronization event in which process wants to participate. A process may signal at any time. Each process that signals must eventually wait. The synchronization method is dynamic in the sense that neither prior nor global knowledge is required regarding which processes will participate in a synchronization events. When a process executes `synch_wait(n,patience)` it checks to see whether all of the signaling processes have waited. If not, the waiting processes blocks. In a multitasking environment, a context switch to a background job would be possible here. If the waiting process is the last signaling processes to wait, it readies all of the blocked processes. Figure 3 is an illustration of how such a sequence might take place.

```
Program 1         Program 2         Program 3

   .
signal(1)                           signal(2)
signal(2)
   .              signal(1)
   .
wait(1,-1)
{blocked}
   -                                wait(2,0)
   -                                {continues,
{readied}         wait(1,-2)         patience=0}
   .              {readies p-1}
   .              {continues}
   .
wait(2,-1)
{continues}
```

Figure 3: Three programs synchronized at two points

The patience parameter makes it possible for a process to execute `synch_wait()` and yet not block. If `synch_wait()` is called with patience equal to zero, the process will not block. If `synch_wait()` is called with patience equal to a positive value, the process will block until the final wait occurs, or for a period proportional to the value of patience. Thus, with the patience parameter, a process can indicate: 'wait until all synchronizing processes complete', with patience < 0; 'don't wait', with patience $= 0$; and 'wait until all synchronizing processes complete, but no longer than patience', with patience > 0. It is possible that some processes might not need to block. For example the data produced by program 3 above might be required by program 1 after program 1 executes `synch_wait(2,-1)`; so program 1 must block to provide for the possibility that program 3 takes longer. But program 3 might require nothing of program 1; so program 3 should execute `synch_wait(2,0)` and continue execution.

By recording a history of the time each synchronizations completed and the completing process number, the `synch_wait()` primitive can compile the data necessary to perform a critical path analysis. This tool should aid the programmer in spreading tasks across processors and achieving a balanced load.

The `synch_signal()` and `synch_wait()` primitives operate by writing into statically declared data spaces in global memory and the local memory of each processor. The static declaration reduces the runtime overhead of passing pointers. Within this mechanism, at negligible additional cost, the `synch_wait()` primitive is made to return the patience value of the waiting process that completes the synchronization event. Thus, if program 1 had executed:

```
result = synch_wait(1,-1);
```

resuming execution, result would have the value of -2, the patience value passed by program 2 in `synch_wait(1, -2)`. This feature might be used diagnostically, or to communicate simple messages.

Programming Environment

The C standard I/O library has been implemented using the communications system described above. The I/O library is a buffered I/O system, with server processes on the 68K filling and emptying the buffers. In addition to the file I/O capabilities, stdin, stdout, and stderr have been defined to be the input and output streams of windows of the VGTS, allowing user interaction through the V-System.

Applications

Multiprocessor System for the Stanford/JPL Hand

The Stanford/JPL hand has three fingers with three degrees of freedom each, actuated by a coupled pulley system with 4 tendons and motors for each finger, for a total of twelve motors. Each tendon has a tension sensor mounted near the finger to allow control of joint torque, and motor shaft encoders to determine motor position. This hand has the necessary dexterity for fine motion and force control of grasped objects, and for regrasping operations where objects are reoriented within the hand.

Force control strategies have been developed and implemented on a PDP-11/60 minicomputer that enable control of object orientation and allow the object to be regrasped in new orientations (Fearing, 1986). The control program is divided into high and low level servos. At the lowest level, a joint torque servo runs at 100 Hz for all three fingers, servoing motor torques based on desired and sensed tendon tensions and motor velocities. At the high level, desired forces in a spherical reference frame based on the task geometry are specified at 33 Hz. These servo programs saturated the computational power of the PDP-11/60. For improved performance in motion velocity and force accuracy, higher servo rates are required.

Tactile sensor arrays (8x8) in cylindrical finger tips have been fabricated here for incorporation on the 3 finger hand. Each sensor will be scanned at between 10-30 Hz rate, so approximately 5000 samples per second will need to be analyzed to determine essential information for object manipulation such as contact location, object orientation, and contact forces. In addition, a program with global knowledge of the state of all three fingers and the object, and with an overall plan for the manipulation operation must be executing concurrently.

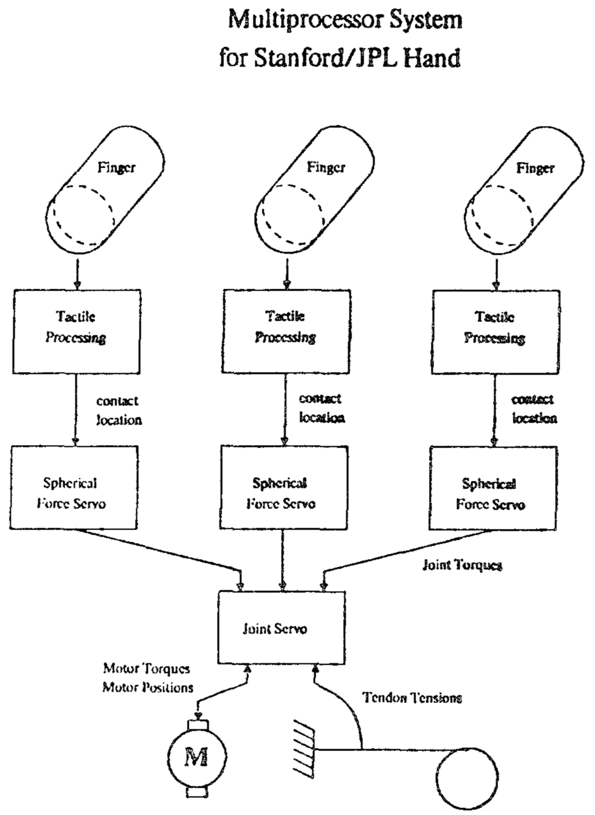

Figure 4: Multiprocessor System for Stanford/JPL Hand

The computational requirements of this hand control system are beyond the capabilities of even large minicomputers like the Vax 11/780. Luckily, each finger's servo control can be run independently from the other fingers, which allows three separate parallel computations to be done. Figure 4 shows the proposed structure of the hand control system, where each box represents a separate CPU. To eliminate IO contention at the hand interface, the joint servo processing is done by one CPU for all three fingers. Each finger will have a dedicated processor for tactile processing, and one for implementing the force control loop. This totals 2 and one third equivalent CPUs per finger. The tactile and force control processors will share responsibility for correcting finger forces to get desired object motion, to prevent unwanted slip, and to recover from errors in object attitude. Benchmark studies suggest that with this architecture, we will be able to achieve joint servo rates of 200 Hz, and spherical force servo rates of about 100 Hz if required.

An important consideration in parallel processing is the communications cost of passing parameters between processors. With the servo rates above, a conservative estimate is a bus bandwidth requirement of less than 200K bytes/second, which is only 10% of the typical 2 MB/s bandwidth of the Multibus.

Cosmos

COSMOS is an experimental programming system designed to facilitate experiments in manipulator position and force control. COSMOS was originally implemented in a PDP 11/45 minicomputer, and subsequently implemented in a PDP 11/45 and PDP 11/60 multiprocessor configuration. However, these computers were found to be unsuitable for manipulator control research. A versatile and useful manipulator control computer system should have the following attributes:

1. *Large computational power.* The natural frequencies involved during force control operation are much higher than encountered during position control, and thus require greater servo bandwidth, and consequently greater computational power.

2. *Large amounts of memory.* In order to evaluate and contrast control algorithms in a quantitative way, it is necessary to store large amounts of data in real time for later analysis.

3. *Graphics Capablilties.* Display facilities for graphical analysis of the run time to enable quick evaluation of experimental results.

The NYMPH multiprocessor system fulfills these requirements, enabling a vastly improved implementation of COSMOS.

The control scheme used in the COSMOS system (Khatib and Burdick 1986) is based on the operation space approach, which employs an operational space dynamic model of the the manipulator being controlled (a PUMA 560 manipulator in this case). The real time control algorithm can be divided into two levels:

1. A "high level" system which computes the configuration dependent kinematic and dynamic models at a relatively lower rate.

2. A "low level" servo system which computes the servo equations at a faster rate using sensor data and the dynamic data from the "high level" .

In essence, the low level system measures the manipulator position and forces, and then computes joint torques using a series of vector and matrix operations. The vector and matrix elements used in these computations are dependent on the configuration of the manipulator, and are updated by the high level. A third level, the "programming level" interacts with the manipulator programmer, and performs run time program execution.

Each of these levels can be further divided to extract more parallelism. Since the control algorithm used in COSMOS is not based on a joint level control, but rather on direct and decoupled control of the task coordinates, each level of the run time system can be further decomposed into sub-systems that control the position degrees of freedom of the manipulator, and the orientation degrees of freedom.

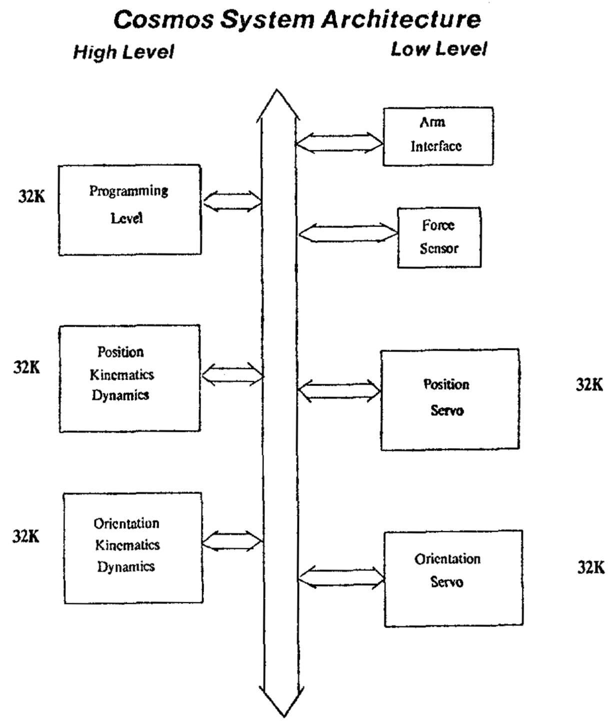

Figure 5: Multiprocessor System for COSMOS.

Figure 5 represents how the computational burden in COSMOS might be spread across five processors. Two processors could perform the low level position and orientation servo (as well as sensor processing); two processors could compute the position and orientation kinematics and dynamics, and a fifth processor could handle programming and run time execution. These tasks could be further broken down into smaller computational units, and allocated to more processors. However, the extra overhead for interprocessor communication and synchronization would minimize the extra computational power gained by further parallelization.

The current COSMOS implementation using the NYMPH system is distributed in a three processor configuration, in which the "high level" position and orientation kinematics/dynamics algorithms and "programming level" are implemented in a single processor. With this configuration, a low level servo rate of 220 Hz and a high level dynamics computation rate of 110 Hz has been achieved. With the five processor implementation (under progress), we expect to have a 300 Hz servo rate, 300 Hz rate for position kinematics and dynamics, and a 150 Hz rate for orientation kinematics and dynamics.

Performance

With the 32Ks and the 68K all on the same bus, communication between the processors is very fast. Data can be passed as fast as 1000k bytes per second between the 32Ks, and as fast as 490k bytes per second between the 68K and a 32K.

NYMPH's message passing system is also very efficient. The average time for a message transaction on NYMPH is 4 milliseconds, compared to 2 milliseconds for the V-System message alone.

The C library performance is reasonable, with file transfers occurring at rates as high as 10k bytes per second, compared to the V-System file I/O rate of 20k bytes per second. At this rate, it takes about one minute for a 32K to fill all 512k bytes of its RAM. This is not surprising if one considers the circuitous path the data takes to get to 32K from Vax, via Ethernet link, buffered in the V-System I/O system, then buffered again in the NYMPH I/O system.

The speed of the 32K/Vax link is much slower than the link between the 32Ks and the 68K. This would be a problem if NYMPH depended on the Vax for processing, but NYMPH uses the Vax only as a file server. Since NYMPH applications do file I/O primarily during initializations at the beginning of programs and before time critical sections begin, and at the end of programs for postmortem information, the relative speed of the 32K/Vax link has little bearing on the overall performance of the system.

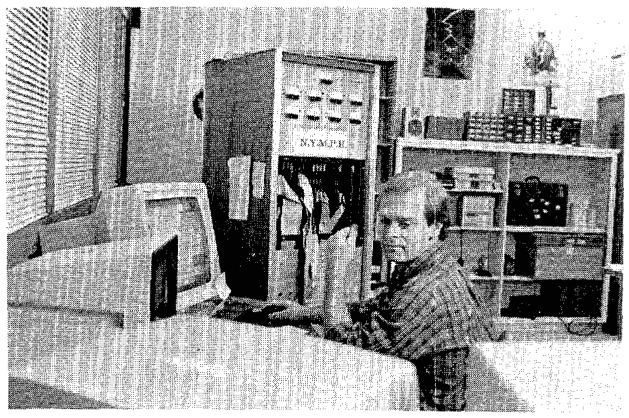

Figure 6: NYMPH running COSMOS.

Conclusion

NYMPH's strengths are in the large amount of computing resources available, and the familiarity and ease of use of the programming environment. In the present configuration, it is not anticipated that any applications will run short of computing power. All of our applications will run much faster than they did in the past. Applications such as Cosmos will not need all 7 processors, and thus will leave some processors idle. NYMPH will be well able to meet our present computational requirements.

NYMPH's user interface will simplify the programming of the applications programs. The multi-window editors available at the NYMPH console allow minor changes to source code conveniently with minimal interruption of work. The graphics capabilities of NYMPH allow researchers to easily analyze date with graphs and plots, without leaving the console. The familiar programming environment of NYMPH, with the C runtime library, reduces the amount of special routines the programmer must learn to use NYMPH. NYMPH's user interface provides ample tools for making research efforts efficient. Overall, NYMPH has combined ample computing power and a good programming environment to make an effective research tool.

Acknowledgments

The authors wish to gratefully acknowledge Professor Thomas O. Binford for his guidance and support. We would also like to thank Lance Berc of Stanford's Distributed Systems Group for making available his expertise on the V-System. This work was funded by DARPA Contract F33615-82k-5108.

References

1. E. J. Berglund et. al.Computer Systems Laboratory, Stanford University, *V-System Reference Manual,* 1985.

2. D. R. Cheriton, "The V Kernel: A software base for distributed systems," *IEEE Software,* April 1984, pp. 19-42.

3. R.S. Fearing, "Implementing a Force Strategy for Object Re-orientation," *1986 IEEE International Conference on Robotics and Automation,* April 1986.

4. Oussama Khatib and Joel W. Burdick, "Manipulator Motion and Force Control," *1986 IEEE International Conference on Robotics and Automation,* April 1986.

5. Ravi Nigam and C.S.G. Lee, "A Multiprocessor-Based Controller for the Control of Mechanical Manipulator," *1985 IEEE International Conference on Robotics and Automation,* March 1985, pp. 815-821.

6. F. Ozguner and M.L. Kao, "A Reconfigurable Multiprocessor Architecture for Reliable Control of Robotic Systems," *1985 IEEE International Conference on Robotics and Automation,* March 1985, pp. 802-806.

7. David M. Siegel, Sundar Narasimhan, John M. Hollerbach, David J. Kriegman, George E. Gerpheide, "Computational Architecture for the Utah/MIT Hand," *1985 IEEE International Conference on Robotics and Automation,* March 1985, pp. 918-924.

8. Karsten Schwan, Tom Bihari, Bruce W. Weide and Gregor Taulpbee, "GEM: Operating System Primitives for Robots and Real-Time Control Systems," *1985 IEEE International Conference on Robotics and Automation,* March 1985, pp. 807-813.

9. Yuan F. Zheng and Ben R. Chen, "A Multiprocessor For Dynamic Control of Multilink Systems," *1985 IEEE International Conference on Robotics and Automation,* March 1985, pp. 295-300.