MANIPULATOR CONTROL 2

Requirements for Implementing Real-Time Control Functional Modules on a Hierarchical Parallel Pipelined System

Thomas E. Wheatley, John L. Michaloski and Ronald Lumia
National Institute of Standards and Technology
Bldg. 220 Rm. B124
Gaithersburg, MD 20899
February 1989

Abstract

Analysis of a robot control system leads to a broad range of processing requirements. One fundamental requirement of a robot control system is the necessity of a multicomputer system in order to provide sufficient processing capability. The use of multiple processors in a parallel architecture is beneficial for a number of reasons, including better cost performance, modular growth, increased reliability through replication, and flexibility for testing alternate control strategies via different partitioning [GAG86, NAR86]. A parallel architecture mandates three additional requirements not found in a sequential architecture. First, the problem must be partitioned into tasks. Hierarchical structuring will be used to partition a robot controller, with each level in the hierarchy corresponding to a separate task. Each task level is further modeled as a virtual control loop. Second, each task must be scheduled for execution on one or more processors. The problem of scheduling processes to processors will highlight the importance of the programming environment. Third, synchronization of control and data flow (communication) must be performed during execution. A survey of the progression from low-level control synchronizing primitives to higher level communication tools will be presented. The system communication and control mechanisms of existing robot control systems will be compared to the hierarchical control model. The impact of this design methodology on the current robot control systems at NIST will be explored.

1.0 Partitioning Requirement

The partitioning of a job into cooperating processes is called multitasking or multithreading. The effectiveness of a parallel implementation depends on the inherent parallelism of the algorithms and the overhead of interprocessor communication. Algorithms that are purely sequential will not run faster on a parallel machine. To achieve the benefits of parallelism, the cost of communications between processors must not exceed the time savings obtained by parallel execution on the different processors. Job partitioning is usually discussed in the context of a fine-grain versus coarse-grain approach [KRU87, LYO87, LYO86]. An entire process which is allocated one processor for the life of its execution could be considered coarse-grain. Solutions to many problems are too complex to be calculated within an appropriate time frame with a simple coarse-grain sequential execution and may require a fine-grained processing approach.

Parallel machines exhibit different system capabilities. General-purpose coarse-grain machines tend to maximize batch throughput. Supercomputers tend to specialize in fine-grained number crunching in parallel for compute-bound operations. Other fine-grain computers specialize in data-intensive operations. The variety in machine capabilities leads to the distinction between systems that maximize the throughput of many jobs, known as *throughput-oriented multiprocessors*, and systems that maximize the execution of one process, known as *speedup-oriented multiprocessors* [DUB88].

This paper will focus on those parallel architectures that best fit the system requirements of an intelligent robot controller in terms of price versus performance. A robot controller is a large system that is composed of layers of fine-to-coarse-grained processes and can be characterized as a speedup-oriented multiprocessing application. Hierarchical structuring offers an easy and systematic parallel approach to partitioning. With a hierarchical control system, levels in the hierarchy can be developed as sequential processes, and then parallel integration can be modeled as communicating sequential processes [HOA78]. Thus, a control system can be developed functionally independent of the implementation and then hierarchically integrated with the communicating sequential processes model. Two design factors are at the

heart of exploiting the benefits of a hierarchical control system. The notions of "task decomposition" and "response time" are crucial in determining the number of processors in a hierarchical system, the complexity contained within any one level, and how communication should be performed.

Task decomposition can be defined as the process of recursively breaking down a task into smaller, more manageable tasks, until some atomic level of activity is reached. At each level in the hierarchical breakdown, an interface exists where the adjoining levels exchange information. The higher level communicates a command to its neighboring lower level. Likewise, the lower level command communicates a status to its neighboring upper level. This communication protocol is analogous in sequential programming to a subroutine invocation as a command and a return value as a status. Defining a hierarchical decomposition for a robot control system starts with a high level task, and through a series of task decompositions, reduces this task to a set of motion primitives. Concurrently, status of the environment filters up the hierarchy to provide feedback to the task decomposition.

Unlike sequential hierarchical decomposition, each lower level in a parallel pipelined architecture is not completely dependent on its neighboring upper level. Although levels may share some data that models the world, each level can be considered to run independently of any other, responding to a command and supplying a status much like a plant in a normal feedback control system. When composed as a system, hierarchical control is built as layers of *virtual control loops*. When executing, each virtual control layer in the hierarchy can be considered as part of a long chain defining the hierarchical state, yet each level's action is based on its own control flow. The virtual control loop software exhibits cyclic feedback behavior that samples inputs including command and status and guarantees some output within a given time.

Performing task decomposition is a function of both the state of the world and the number of operations that must be performed. Determining the amount of work a task must perform is a subjective issue but depends heavily on the amount of time required to make a decision. Responding to an event too late nullifies the control no matter how intelligent the subsequent action. This timing restriction leads to the definition of *response time* as the maximum allowable time duration between an event and an action resulting from that event.

Meeting a response time restriction may limit the amount of intelligent behavior at any one level. By dividing task decomposition into planners and executors, one can maintain real-time control yet concurrently evaluate alternative future actions. The *planner* is responsible for generating a plan consisting of a series of actions. An *executor* is responsible for stepping through a generated plan. The executor matches the current state of the machine against a set of preconditions, which triggers the corresponding action.

The planner and executor together compose the task decomposition module that determines the control behavior. But there are more functions to feedback control than just planning and executing. Supplying and interpreting external sensor information is another important function in feedback control. *Sensor processing* operates concurrently with the control function to interface to the real world. Sensor processing consists of data acquisition, filtering, enhancement, and other functions up to and including object recognition.

The disparity of purpose between control and sensing makes it useful to mediate the information exchanged between the control and sensing functions. The mediating function, defined as the *world model* [ALB81], allows the decision process to become more abstract in its nature of queries about the world, and less dependent on the physical nature of the actual sensors being used. The world model provides the system's best estimate and evaluation of the history, current state, and possible future states of the world. The world model provides accurate and current information for reflexive behavior (i.e. what is), the best estimate of the world in deciding the future plans (i.e. what if), and integrates the information from the sensor processing module. This leads to a partitioning of the system into hierarchical levels of virtual control loops, each of which consists of a control, a sensing, and a world modeling component. Figure 1 illustrates the partitioning of one level within the hierarchy.

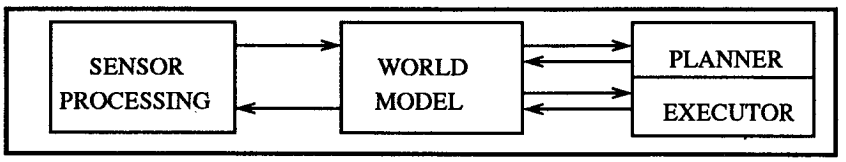

Figure 1. Control Level Partitioning

Further refining and partitioning of a control level is possible but is beyond the scope of this paper. Albus more thoroughly covers the decomposition and partitioning of a hierarchical control system [ALB87].

2.0 Programming Environment and Scheduling Requirements

A traditional real-time software development environment supports the development and testing of algorithms for logical correctness on a "slower" host machine, with later execution of these logically correct algorithms on a "faster" target system, to allow testing under real conditions. The goal of the host system is to offer as many development tools as possible. Target systems are used to meet the requirements of real-time response which implies a premium on efficiency. Assigning processes to processors is a major concern in multiple processor systems. The process of mapping processes to processors and scheduling tasks within these processes is iterative. Should the granularity chosen not realize the desired speed, additional processors may be required, or revision of the algorithm on the host system for subsequent use on the target system may be required. Because of this iterative nature, understanding the requirements of an efficient and robust programming environment will lead to an increase in software productivity.

2.1 Real-Time Target System Software

High data bandwidth, maximized throughput, and fast execution are desirable features of a real-time control system. To achieve this, target systems exhibit a tighter coupling between system support and the control algorithms. Sometimes, the operating systems of target systems are by design as primitive as possible so as to remove any unnecessary overhead [NAR86]. Efficient software performance for real-time systems has been determined to require several features including : 1) fast context switching time between tasks, 2) small interrupt latency, 3) feature priority scheduling, 4) allowing high-priority tasks to instantly pre-empt lower priority tasks, 5) methods for synchronization of tasks, and 6) granting low-priority tasks non-preemptable status.

Although speed is important in a real-time control system, reliability and deterministic behavior are the ultimate system measures that cannot be sacrificed at the expense of optimizing performance. Based on the deterministic criteria, a real-time system has a set of processing requirements that are different from a general purpose operating system. For instance, fairness is not an issue in real-time control. Either the processes are permanently dedicated to the hardware or are guaranteed resident during critical sections (i.e. non-preemptable). Another area affecting the deterministic behavior of the real-time model is processor scheduling. Time-sliced and round-robin scheduling algorithms result in nondeterministic performance and imply difficulty in verification. Exact time quantum round-robin scheduling can overcome the non-determinism, but the overhead of continual context switching overrides the use of the round-robin method except in very low speed applications. Instead, a priority-based system where processes execute in known sequences and surrender the processor willingly results in predictable, hence deterministic, behavior.

2.2 Host System Support Software

General purpose computer operating systems provide a robust set of programming tools, such as a wide variety of compilers, editors, source code control tools, and many tools ideally suited for software development. General purpose, multi-user computers are not useful as target systems because of the unpredictability of the underlying operating system. With so many responsibilities to handle concurrently, large general purpose operating systems are not generally designed to include features required for running in real-time. Therefore, these are separated and an efficient communication link (such as a local area network) from the real-time target system to the host development system is not strictly required but greatly improves system flexibility.

2.3 Analysis of System Software Requirements

The programming environment for a hierarchical real-time control system is comprised of both host and target system needs. The necessity to divide the software system into two components is based on the need to meet timing requirements. The distinction between the two components becomes hazy as one moves up the hierarchy until eventually the distinction disappears.

At the very lowest levels, the functional components must be efficient and highly streamlined since parts of the control system may have to run without the luxury of any operating system assistance. A low-level servo controller cannot afford the time to allow multi-tasking or other system overhead. A kernel which handles spurious interrupts, limited background I/O service, and a basic monitor for off-line board

level troubleshooting is sufficient for streamlined system support. However, stand-alone levels still require interprocessor communication to other levels in the hierarchy.

Higher levels are allowed longer processing intervals so that context switching using multi-tasking among different processes can be performed. Further, higher levels may require a high-speed file system for data logging and performance analysis that can be used as an evaluation tool or as a postmortem data recorder box (or "black box") after an unforeseen crash of the system.

This review of system requirements leads to the observation that a computing system with a rich set of software tools is necessary to handle the broad variety of processing needs ranging from the real-time hierarchy to supporting the user interface. A prototype architecture for such a system would exhibit differing degrees of concurrency that then can be implemented as a blend of multiple processors and interleaved execution on a single processor. Because of the disparity of response times required at various levels in the hierarchy, different levels of granularity are required. At the lowest levels, planning may be impossible, and even execution may require multiple processors to achieve a solution. Moving higher in the hierarchy, timing constraints prevent the planner and executor from residing on the same processor. In this case, the two processors would run in parallel, and asynchronously the planner would update plans. At higher levels, completion time for plans is less critical so that multitasking on a single processor can supply enough processing power for both the planner and executor. If the timing constraints are not stringent, multiple levels can be combined onto one processor, or the use of a LAN can be used to connect other computers as a part of the controller. Each of these configurations is based on the response time requirement of the level. Figure 2 provides a guideline for the type of performance required of differing levels in the hierarchy. This is based on the premise of allocating a nominally fixed percentage of response time devoted to communications versus computation for each level of the hierarchy. The actual times chosen are relative, and will differ according to one's choice of hardware and operating system support. This percentage of time concept and the problem of timing analysis for a hierarchical control system is more thoroughly treated in [MIC88].

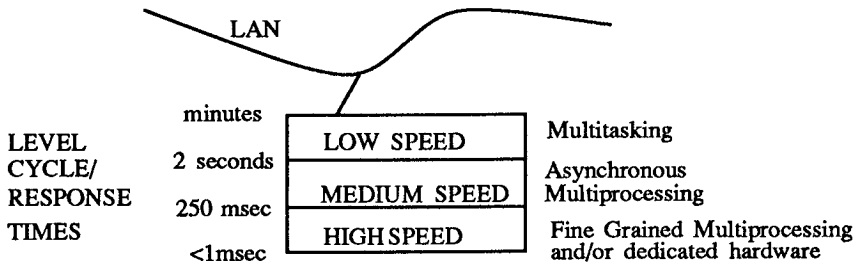

Figure 2. Computing Resource Utilization Based on Cycle Response Time

3.0 Synchronization and Communication

Synchronization is a basic building block that a multiprocessor system must contain. Synchronization serves the dual purpose of enforcing the correct sequencing of processes and ensuring the mutually exclusive access to certain shared, writable data. Synchronization is usually supported by some special-purpose hardware. These basic synchronization primitives are then used to build higher level synchronization and communication tools in software or microcode. Communication and synchronization are difficult to separate because communication primitives can be used to implement synchronization protocols and vice versa. In general, communication can be defined as transferring or exchanging information between processes, whereas synchronization is a special form of communication in which the data is control information.

3.1 Synchronization

Synchronization provides mechanisms for coordinating *control flow* and *data flow* between multiple processes. Synchronization tools evolve from basic hardware primitives to complex software and microcode schemes. Parallel machines include hardware primitives capable of enforcing mutual exclusion on a resource. Semaphores and message passing are software extensions built upon these primitives. For control flow, parallel processes are synchronized to wait until all processes are ready. For data flow, synchronization limits access to a shared resource to one process at a time. This section will cover synchronization of control flow, while the following section on communication will cover synchronization of data flow.

At the user programming level, control synchronization between the master routine and the slave

subroutine in sequential processing is guaranteed because the master waits until the slave is done before resuming execution. In concurrent machines, the master could continue processing or wait for a rendezvous at a later point in time. In parallel programming, the concept of coroutines appear as a more fundamental synchronization structure than subroutines, which can be regarded as a special case [HOA78]. Synchronization in a parallel machine must handle the problem of parallel processes finishing at different times, waiting for all to complete (synchronizing), and then continuing.

Synchronization schemes are based on some indivisible hardware action limited to only one process at a time. For example, a basic semaphore consisting of a test-and-set can be used to mediate requests. These primitive operations are subsequently used as the building blocks for higher level multiprocessing synchronization mechanisms such as *monitors*, *semaphores*, *single-writer/multiple-read blocks*, *master/slave* and *critical sections*. *Priority-level* access is a further enhancement of synchronization. A *barrier synchronization* function is a software or microcoded synchronization tool placed so that processes arriving at the barrier must wait until all pertinent processes arrive. Processes either busy-wait or block until the barrier is lifted. *Global broadcast synchronization* uses some signal to establish the beginning of a time period in which all processes across all boards in the system are allowed to write to their common memory buffers [ALB81]. Global broadcast synchronization is best suited where the standard deviation between cycles is small so that updates can occur at a known intervals with little processing time wasted.

3.2 Communication

Interprocessor communication can be characterized as a read-write sequence with two additional facets, synchronization for the exchange of information and destructive/non-destructive read/write execution. For example, writing to a queue is non-destructive (assuming enough storage), while removing it is destructive. Writing to a variable is destructive,while reading it is non-destructive. Figure 3 summarizes the combinations of non-destructive and destructive execution during communication and the styles of communication that result.

	Destructive Write	Non-Destructive Write
Destructive Read	non-queued message passing	queued message-passing
Non-destructive Read	variables, shared memory	mail (assumes explicit deletemechanism)

Figure 3. Styles of Communication

The execution may also have a time-out feature, so that the process does not wait indefinitely for new information. The information exchange can be synchronous or asynchronous. The following sections more closely explore the methodologies and rationale of different communication techniques.

3.2.1 Evaluating Communication

To evaluate the effectiveness of interprocessor communication, the performance measures of latency and throughput are used. To characterize a robot control system as real-time implies a guaranteed maximum latency for interprocessor communication. *Latency* is defined as the elapsed time before a message is acknowledged. *Throughput* is defined as the number of bytes one process can send to another process in one second, especially important for systems that share large amounts of data. However, 20 bytes of information with a 20 msec update rate cannot be handled with a communication protocol that is efficient for moving large amounts of data but which requires a 100 msec setup time.

Additionally, the concepts of flexibility, data integrity, and extensibility are as equally important in evaluating a communication scheme but are more intangible. Data integrity and flexibility can best be explained with an example. On sequential machines, interprocess communication is performed by calling or invoking a subroutine and passing the appropriate parameters on the stack by value or by sharing global variables. How the parameters are exchanged controls the method of communication. Passing parameters "by value" places a copy of the parameter onto the stack and can be considered message passing since each processes' internal variables are independent of the other processes. Passing parameters "by reference" pushes a pointer or address onto the stack so that data is shared between the processes. In a uniprocessing architecture, passing parameters "by reference" saves excessive copying. However, in parallel processing such copying is more

flexible since a user cannot be assured that the two processes both have access to the pointers or addresses, and that the addresses are valid (in a dual ported memory scheme, on-board versus off-board addresses differ). However, complete copying is slower and this highlights the fact that a robot control system must address the issues of flexibility and data integrity in conjunction with performance measures. This coupling of evaluation leads to numerous communication design alternatives aimed at meeting different priorities.

3.2.2 Communication Designs

Different designs are available in order to perform real-time communication. Gauthier, et al provide an in depth survey of the various interprocessor communication designs [GAU87]. Overall, communication depends on the system architecture, either tightly or loosely coupled. Tightly coupled systems have a public memory architecture, where processors communicate through *shared memory*. Loosely coupled systems have private memories where processors generally cannot read each others memory and must have an explicit *message-passing* communication channel.

Message passing systems have memory attached privately to each processor (at least conceptually), so that processors communicate only through explicit transmissions of whole messages [LYO87]. Message passing is more generic in that messages can be passed not only across backplanes but across machines. Message passing is typically slower, as it is generally implemented at the subroutine level and requires coordination at the receiving end.

Shared memory is a special form of message passing, that provides equal access to all processors. Shared-memory offers many advantages, including ease of sharing data, rapid communication, and high performance. The advantages that make shared-memory architectures attractive result from the tight coupling along the critical path between processors and memory. This leads to a notable increase in performance. Shared memory offers such a highly efficient and straightforward method for communicating among parallel processes that it is commonly used for parts of real-time systems [PAU86, KOR86, KAZ87]. Assuming a shared memory communication scheme, the disadvantages that arise are: 1) guaranteeing mutual exclusion of shared resources across processors and 2) maintaining a consistent view of addressing across processors. Another major disadvantage of shared memory is that as the number of processors grow, the arbitration of accesses to the common memory burdens the system and degrades performance. The use of multiple communication links can ease this but increases the problem of internal versus external addressing for common memory mapping.

3.2.3 Communication Synchronization : Asynchronous vs Synchronous

In general, communication mechanisms can be characterized as either synchronous (blocking) or asynchronous (non-blocking). Routinely, blocking implies surrendering the processor and waiting for some condition to occur before continuing execution. Whether to block provides the fundamental distinction between a synchronous "lock-step" communication exchange and an asynchronous "free-wheeling" communication exchange. Synchronous communication uses an explicit handshake for acknowledgment. In an asynchronous communication protocol, the sending process does not have to wait or block for the receiving process to acknowledge receipt of the message. Since asynchronous messages can arrive at random times, queues are attached to save messages. Acknowledgment is optional, but can be installed as part of the mechanism. The message itself must contain instructions whether an acknowledgment is necessary.

One advantage of synchronous communication is simplicity (no queues or queuing monitor) and thus low overhead. Another advantage is the savings in space and time that result because data can be directly copied from the sending processes' buffer to the receivers with no intermediate storage action required. Finally, the one-to-one correspondence of a synchronous transmission provides a stricter notion of accountability and determinism that enhances software reliability. The major disadvantage of synchronous communication is the time spent synchronizing and acknowledging, i.e. waiting, for each communication. Another disadvantage is the lack of flexibility and subsequent extra software penalty for handling dynamic reconfigurations such as many-to-one communication channels. For example, in a dynamic environment, a synchronous communication exchange requires a priori knowledge by both parties of the other's existence in order to synchronize.

The advantages of asynchronous communication are speed and flexibility. The lack of synchronization and acknowledgment steps improves performance. The disadvantage is that error detection may be overlooked. Flexibility results in that the message may embody more of the communication mechanism, i.e. destination of an acknowledgment. With information embedded in the message rather than the a priori synchronous

connection, a new client can be dynamically added to a server by including a destination address within the message. The disadvantages of asynchronous communication is the lack of accountability for errors which requires programmer discipline to foresee and handle errors.

These two models are basically equivalent in that the synchronization (i.e. control information exchange) before the communication can be considered a bi-directional asynchronous communication. Further, asynchronous communication is usually extended to include an acknowledgment step. In addition, synchronous communication can use a table lookup for dynamic reconfiguration. From a software validation standpoint, synchronous communication is preferred, but the system may not tolerate the extra amount of overhead. In general, for connections that are statically predefined one-to-one exchanges, synchronous communication provides the most reliable scheme. For connections mapped as many-to-one and dynamically reconfigurable, asynchronous communication provides the cleaner implementation.

3.2.4 Communication Duration : Connection Ties

Communication connection ties can be either temporary (datagrams), or permanent (virtual circuits) [POS80]. The communication ties can be established dynamically through a system broadcast [FRI86] or by statically defined connections [NAR86]. A dynamic connection is established by the sending process broadcasting a system-wide request for the receiver location, who responds in turn with its location. Communication is then direct. Dynamic connections offer flexibility, but should be done upon system start-up for permanent communication links. Dynamic connection for temporary links would not be useful for a system with stringent real-time control demands. Static connections use tables to link sending and receiving processes. Static connections are fast, but may require the overhead of some centralized server.

4.0 Analysis of Design Alternatives

Given the rich set of communication design strategies, some general heuristics assist in focusing the communication design. A survey of existing multiple processor robot control systems reveals a wide range of implementations. The requirement to satisfy real-time constraints has lead to numerous implementations using tightly coupled architectures with several processor boards on a common bus [PAU86, NAR86, KOR86, KAZ87, GAU87, SCH87]. The target operating system of these implementations vary in degree of system complexity from the basic use of interrupts and handshaking operations to handle interprocessor communication[PAU86], to a limited operating system with special communication features [NAR86], to an extended real-time operating system [SCH85].

These implementations share common system architectural characteristics with a hierarchical control system. Although few of these control systems are directly labeled as hierarchical, most systems have at least a two-level hierarchy with a low-level, real-time subsystem handling real-time motion control and a slower high-level subsystem responsible for planning. This basic hierarchical decomposition in each of these applications can be generalized into two virtual control loops exchanging commands and status. Implicitly, these applications have additional levels within the hierarchy, but these levels do not directly correspond to a virtual control loop implementation because of the concept that returning status is implicit in a serial execution of the processes. These implicit levels could be partitioned into separate concurrent processes that communicate through established interfaces rather than serial routines that communicate via subroutine calls. Further, with concurrent operation, the concept of sampling the return status rather than just receiving a final status embodies the feedback nature of a virtual control loop.

The concurrent hierarchical model of a robot control system containing virtual control loops has been implemented with a purely executor style of task decomposition for a robot control system at the National Institute of Standards and Technology [BAR82]. The flow of control was based on state-table transitions. Where planning was appropriate, static plan definitions were used. The system offered several benefits. First, the system was sensory-interactive and adapted to perturbations in the environment in real-time even though the world model was limited to basic feature recognition. Second, hierarchical decomposition created well-defined interfaces that allowed substitution of different implementations of a level without major impact on the higher or lower levels that lead to proposed interface standards [FITZ85]. Finally, the system was able to execute in real-time and supply robot updates within a fixed interval on the order of milliseconds.

Table 1 summarizes some of the advantages and disadvantages of the previously discussed communication features. Double lines separate independent communication features, from which one of the alternative designs must be chosen. Sections between the single lines demarcate the selections for each feature.

Although one of the communication features must be chosen for a given level, it generally does not constrict one to that choice for the entire system. Indeed, a very flexible system would incorporate some of each of these using the response time criteria as an initial guide to the selection.

Issue	Feature	ADVANTAGES	DISADVANTAGES
Coupling	Tightly Coupled Shared Memory	high-speed, simple	Rarely portable, interconnect limits, contention problems
	Loosely Coupled/ Message Passing	Flexible, generic better security, cheaper for larger number of processors	Slower, limited data bandwidth
Synchronization	Synchronous Comm.	Simple, guaranteed comm.	Slow, requires time-out
	Asynchronous Comm.	Flexible, fast	Overhead - queuing
Duration	Temporary Channel	multiplex resources	repeated overhead
	Permanent Channel	low overhead	1:1 communication best
Connectivity	Dynamic connectivity	flexible, reconfigurable	slower, larger overhead, special hardware
	Static connectivity	simple, fast	centralized server necessary
	Centralized connectivity	better accountability	extra decode overhead each message
	Decentralized connectivity	unbundled, demon model	slower, resource intensive

Table 1. Communication Feature Summary

The algorithmic view of data shared between processes also affects the communication interfaces. The most straightforward algorithm is double buffering using shared memory where only one reader and one writer grab complete control of the common memory until finished. A more complicated algorithm (n readers, one writer) allows several readers access to the memory at once, but only one writer at a time. For an example of an n-readers, 1-writer algorithm, consider the case were two levels in the hierarchy are communicating command and status to each other. Not only could one level read a command, but a diagnostic process and a data performance logger could also sample the command buffer with minimal overhead.

The client/server model is a widely used communication protocol. This method fits into a message passing type of interface. The server offers a level of abstraction from the user (i.e. the client) who queries the server. The server hides the client from the physical implementation details and allows the level of discourse to be of a logical and abstract nature. The client/server offers the strongest rationale for using a message passing scheme (even if it is simply a subroutine call). Within the control hierarchy, the world model acts as a server to its clients, the planner and the executor. Clients request service from the world model. If the world model is busy, this request is queued. When ready, a client is serviced by the world model. The world model translates all logical queries concerning the system into a corresponding physical representation and responds with an answer. Thus, the world model shields its clients from understanding its physical representation of the world.

Given these two communication interfaces, a general rule applies :

When one process has knowledge of the data structure used by the other processes, then access is direct. If the processes use different data structures or one views the data logically while the other handles physical implementation, a server is necessary in order to communicate information.

The amount of data moved between processes is another factor in the design. Moving large amounts of data or shifting processes to and from private processor memories can be cumbersome and slow. A shared memory scheme is more appropriate here since attempting to use a message passing technique as a server to mediate shared data can be much slower, by a factor of thirty, than direct access [KAZ87]. Thus, if the data structure is conceptually consistent across processors, shared memory improves performance.

5.0 Conclusion

This paper has addressed some design issues associated with developing a hierarchical control system for an intelligent machine composed of a broad set of functional requirements. This set of requirements is too large to handle in a conventional sequential programming environment because of the stringent timing constraint imposed on performance. The use of a hierarchical structuring of modules for a control system creates levels that operate in parallel and can be characterized as virtual control loops. Virtual control loops use software to emulate a feedback control loop by sampling as inputs the command from a neighboring upper level, comparing the input to the sensory sampled environment, computing a goal-directed action, and outputting this action to the neighboring lower level, all within a fixed response time. Mapping this hierarchical control system onto a parallel pipelined computing system is a reasonable method of implementation. It offers flexibility and reliability at a reasonable cost. Work has begun on the design and implementation of a concurrent hierarchical control system that includes planning, extensive world modeling, and high level sensory processing. This system will use many of the aforementioned communication features in a hybrid approach. The concept of the response time dictating the choice of communication schemes (based on a percentage of the response time devoted to communication) will be further explored as a possible aid to limit the iterative process of allocating processes to processors. By using a systematic ordering by time of the various communication primitives available, one can quickly determine the type of system support appropriate for a level, given the response time of that level. The discussion is beyond the scope of this paper, and will be the topic of future publications by the authors.

References

[ALB81] ALBUS, J.S., BARBERA, A.J., NAGEL, R.N., "Theory and Practice of Hierarchical Control", *Twenty-third IEEE Computer Society International Conference*, 1981, pp.18-39.

[ALB87] ALBUS J.S., McCAIN H.G., AND LUMIA R., "NASA/NBS Standard Reference Model For Telerobot Control System Architecture (NASREM)", NBS Technical Note 1235, National Bureau of Standards, Gaithersburg, Md., July 1987.

[BAR82] BARBERA, A.J., FITZGERALD, M.L., AND ALBUS, J.S., "Concepts for a Real-Time Sensory-Interactive Control System Architecture", *Proceedings of the 14th Southeastern Symposium on System Theory*, April 1982.

[DUB88] DUBOIS, M., SCHEURICH, C. AND BRIGGS, F.A., "Synchronization, Coherence, and Event Ordering in Multiprocessors", *IEEE Computer*, Februrary 1988, pp. 9-21.

[FITZ85] FITZGERALD, M.L., BARBERA, A.J., "A Low-Level Control Interface for Robot Manipulators", NBS-Navy NAV/SIM Workshop of Robots Standards, June 6-7, 1985.

[FRI86] FRIEDLANDER, C.B., AND WEDDE, H.F., "Distributed Processing Under the Dragon Slayer Operating System".

[GAG86] GAGLIANELLO, R.D., AND KATSEFF, H.P., "A Distributed Computing Environment for Robotics", *Proceedings of the IEEE International Conference on Robotics and Automation*, San Francisco, CA, April 1986, pp. 1890-1895.

[GAU87] GAUTHIER, D., FREEDMAN, P., CARAYANNIS, G., AND MALOWANY, A.S., "Interprocess Communication for Distributed Robotics", IEEE Journal of Robotics and Automation, Vol. RA03, No. 6, Dec. 1987, pp. 493-504.

[HOA78] HOARE, C. R., "Communicating Sequential Processes", *Communications of the ACM*, Vol. 21, No. 8, August 1978, pp. 666-677.

[KAZ87] KAZANZIDES P., WASTI H., AND WOLOVICH W.A., "A Multiprocessor System for Real-Time Robotic Control: Design and Applications", *Proceedings of the IEEE International Conference on Robotics and Automation*, New York, NY, 1987, pp. 1903-1908.

[KOR86] KOREIN, J.U., MAIER, G.E., TAYLOR, R.H., AND DURFEE, L.F., "A Configurable System for Automation Programming and Control", *Proceedings of the IEEE International Conference on Robotics and Automation*, San Francisco, CA, April 1986, pp. 1871-1877.

[KRU87] KRUSKAL, C.P., SMITH C.H., "On the Notion of Granularity", National Bureau of Standards Report, July 1987.

[LYO86] LYON, G., "Programming The Parallel Processor", Second Symposium on the Role of Language in Problem Solving, Applied Physics Laboratory of Johns Hopkins University, April 2-4, 1986.

[LYO87] LYON, G., "On Parallel Processing Benchmarks", National Bureau of Standards Report , NBSIR 87-3580, June 1987.

[MIC88] MICHALOSKI, J.L., WHEATLEY, T.E., AND LUMIA, R. "Exploiting Computational Parallelism with a Hierarchical Robot Control System", to be published.

[NAR86] NARASIMHAN, S., SIEGEL, D., HOLLERBACH, J.M. , BIGGERS, K., AND GERPHEIDE, G., "Implementation of Control Methodologies on the Computational Architecture of the Utah/MIT hand", *Proceedings of the IEEE International Conference on Robotics and Automation*, San Francisco, CA, April 1986, pp. 1884-1889.

[PAU86] PAUL, R.P, AND ZHANG, H., "Design of a Robot Force/Motion Server", *Proceedings of the IEEE International Conference on Robotics and Automation*, San Francisco, CA, April 1986, pp. 1878-1883.

[POS80] POSTEL, J., "Internetwork Protocol Approached", IEEE Transactions on Communications, Vol. COM-28, Number 4, April 1980, pp. 604-611.

[SCH85] SCHWAN, K, BIHARI,T, WEIDE, B, AND TAULBEE, G., "GEM: Operating System Primitives for Robots and Real-Time Control Systems", *Proceedings of the IEEE International Conference on Robotics and Automation*, 1985, pp. 807-813.

[SCH87] SCHWAN, K, BIHARI,T, WEIDE, B, AND TAULBEE, G., "High-Performance Operating System Primitives for Robotics and Real-Time Control Systems", *ACM Transactions on Computer Systems*, Vol. 5, No. 3, August 1987, pp. 189-231.