

MULTIPROCESSOR SOFTWARE DEVELOPMENT FOR  
AN UNMANNED RESEARCH VEHICLE

Daniel B. Thompson  
Air Force Wright Aeronautical Laboratories (AFWAL/FIGL)  
Wright-Patterson Air Force Base, OH

Abstract

The Control Systems Development Branch of the Air Force Flight Dynamics Laboratory has been utilizing Unmanned Research Vehicles as low cost flight testbeds for in-house developed flight control concepts. Recently, the limitations of the previous aircraft and control system led to the decision to develop a new low cost research vehicle and complementary multiprocessor avionics/control system architecture.

Effective use of the testbed multiprocessor requires that the applications software of the system must be programmable by applications engineers. These engineers, however, may not be experienced in parallel programming, and may not have detailed knowledge of the underlying architecture. The emphasis of this paper will be on the communications protocols and a parallel software design methodology used to develop the applications function for the first phase prototype.

(1) Introduction

Over the past several years, the Control Systems Development Branch (AFWAL/FIGL) of the Air Force Flight Dynamics Laboratory has been conducting research utilizing Unmanned Research Vehicles (URVs). In 1982, an XBQM-106 remotely piloted vehicle was modified to utilize an autopilot based on an automotive emissions and fuel economy control microprocessor, the Intel/Ford 8061. This autopilot was developed to demonstrate the low cost/low real estate benefits of oncoming VLSI microprocessor technology to URV's, missiles, and even manned flight vehicles.

Following this effort, it was determined that the 8061-controlled vehicle, TN17, could be utilized by the Flight Dynamics Laboratory for low cost flight testing of in-house developed flight control concepts. Such a vehicle eliminates the risk factors and subsequent high costs of flying concepts on modified manned aircraft. As a result, applications which previously would not have been considered for flight test could be verified and demonstrated under more

realistic conditions.

As a recent example, the concept of control law reconfiguration was demonstrated in-flight on TN17. In the experiment, an algorithm to detect a surface failure and modify the control surface gains to compensate for the failure was added to the 8061 autopilot. A control surface was physically separated from the aircraft in-flight to prove the concept.

Although such a vehicle provides many possibilities for low cost flight tests, one main problem exists. The XBQM-106 vehicle was designed for a specific mission profile, not for general purpose tests. The aircraft is heavy, slow, and not very maneuverable. In addition, the XBQM-106 was not designed to carry embedded electronics. The bulkhead barely provides sufficient room for the 8061 autopilot and telemetry electronics.

This last factor is of particular importance to the use of the URV as a flexible research testbed. Many of the potential applications for the URV require computational power beyond that available with the 8061. Two possibilities exist to remedy the situation. First, a more powerful ground-based system could be used, with commands uplinked through upgraded telemetry hardware. Given the inflexibilities of the XBQM-106 vehicle, however, the second option, designing a new vehicle and embedded electronics, has been started.

The potential of multiprocessor control architectures has been demonstrated previously by AFWAL/FIGL. In an in-house program running concurrently with the URV efforts, the Continuously Reconfiguring Multi-Microprocessor Flight Control System (CRMMFCS) [1] was successfully developed and demonstrated. This program spawned new efforts in the research of microprocessor applications in flight control. One such effort, the Advanced Multiprocessor Control Architecture Definition (AMCAD) program [2], is currently producing a laboratory testbed system. These programs have provided in-house experience and development capabilities for multiprocessor technology. As such, the development of an embedded multiprocessor avionics/control

system for the new URV system, TN21, was undertaken.

## (2) Development of the URV TN21 Multiprocessor

The main design decisions of the TN21 multiprocessor were derived directly from the goals of the overall aircraft system. As with the previous URV system, low cost was a primary consideration. This applies to development, maintenance, and modification cost factors. Flexibility was another important consideration. The system is expected to take on a variety of applications or airframe configurations. Ease of adaptation is required to accomplish these changes in a timely manner. Also, the multiprocessor needs to be expandable to meet more computationally intensive applications. Fault tolerance was not a primary consideration due to two factors. First, the URV systems are unmanned and only flown in controlled areas where system failures pose minimal risks. Also, the low cost and high capability requirements of the vehicle would be compromised if redundancy were added.

The system goal of primary interest to this paper is usability. In order for the TN21 system to be a useful research tool, it must be accessible by applications engineers; not just the system designers. In the specific case of the multiprocessor, this means that the applications software must be programmable to a large extent by the applications programmer. However, this programmer may not have the experience in programming parallel processes. Practical utilization of the TN21 system dictated that this dilemma be addressed.

The development of the TN21 multiprocessor has reached the end of the first of several phases leading towards a flyable system. The first phase was intended to develop a hardware architecture and basic multiprocessor software operating system, and to demonstrate the prototype system with a representative applications function. In addition, the issue of multiprocessor programmability was addressed. This aspect is the main focus of the discussion to follow. First, however, a brief overview of the architecture and real time multiprocessor/multitasking operating system (RTMOS) will be given. Further details on the prototype, design rationale, and demonstration results can be obtained from [3].

## (3) Hardware Architecture

Although the 8061 proved to be a benefit in the single processor autopilot of TN17, it lacks several features required to make it a suitable element of a multiprocessor configuration, particularly considering the wide range of potential applications. The primary features lacking are floating point

support, sufficient memory addressing range, operating system support functions, and available software and hardware development/debugging tools. The MC68000 was selected instead due to the availability of the above features, the wide availability of parts based upon the processor, and the experience and tools possessed by AFWAL/FIGL for it.

The choice of interconnect was based largely on cost factors. An exotic switching circuit or network is not required on a system such as for TN21. The maximum number of processors was not foreseen to exceed eight to ten. Due to cost considerations, the decision was made to acquire an available parallel bus structure. Given the choice of the MC68000, the VME bus seemed a natural selection. An analysis of the capabilities of the bus confirmed its suitability.

The method to handle the input/output (I/O) requirements of the multiprocessor proved to be more difficult to determine. The 8061 internally contains the capabilities for multichannel analog-to-digital (A/D) conversion and pulse width modulated outputs, as used on the URV systems. Interfacing the equivalent circuitry to one (or multiple) MC68000 processors would add complexity to the design. The answer selected was to interface one (or more if needed) 8061(s) to the VME bus for I/O purposes. This choice has minimal impact on applications and does not require the 8061 to possess the above qualities of an element in a multiprocessor.

The resulting configuration is given in Figure 1. The VME backplane chassis and MC68000 processor boards were purchased hardware. The processor boards include dual ported RAM for VME communications. Each board's dual port section is uniquely addressed, creating a distributed shared memory structure. The 8061 and MC68881 floating point coprocessor hardware were added via attached wirewrap boards.

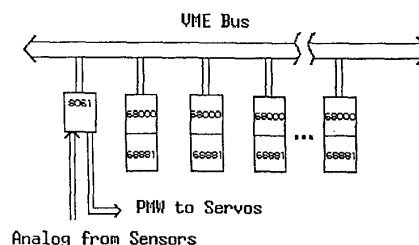


Figure 1

#### (4) Software Architecture

Much of the hardware used in the first phase prototype was comprised of off-the-shelf items. Similarly, the real time operating system kernel could have been off-the-shelf. A variety of vendors market skeletal kernels; however, only a couple directly support multiprocessor configurations. Still, several issues led to the decision to develop the URV RTMOS in-house. The primary factors were cost and multiprocessor programming.

The URV RTMOS extends a basic single processor, multitasking kernel model to multiprocessors. Each processor contains a copy of the multitasking kernel. The task load of the system is distributed to the available processors of the system.

The RTMOS is designed under the assumption that any task can operate on any processor. This assumption helps to eliminate processor dependencies during the programming process. The intent is to allow the programmer to specify tasks and intertask relationships independent of the hardware configuration. To accomplish this goal, the distributed shared memory structure is used. Tasks communicate to other tasks via the shared memory, not by direct processor-to-processor messages. As such, the task model is identical whether implemented on a uniprocessor or multiprocessor.

The task communications process takes place utilizing a mailbox structure through which data is passed. A producer/consumer flag scheme is built into the structure to enforce a unilateral rendezvous. The producer of data places the data (or a pointer to it), in the mailbox, sets the flag, and continues. The consumer waits until the flag is set, consumes the data, resets the flag, and continues. The rendezvous is unilateral since the producer does not wait on the flag. Each mailbox structure is protected for mutual exclusion by a binary semaphore. Timeouts are enforced on exchanges so that proper timing windows are met.

Tasks are synchronized in two ways. First, the above rendezvous process enforces sequential dependencies. A task requiring data, or a signal to start, waits on a rendezvous. Also, to insure that tasks do not clutter the queues of the system while waiting for a rendezvous to occur, a timer queue is used in each processor. These queues are used to hold tasks in a "sleep" state until their time of expected execution, thereby limiting exchange wait times and allowing timeout periods to be measured.

#### (5) Multiprocessor Software Design Issues

Multiprocessor software design inherently is different than that for sequential, single processor software. Some tasks have sequential dependencies

with other tasks. Some may be executed concurrently with others. Effective use of parallel hardware dictates efficient parallelization of a job onto the available resources. However, a requirement that the programmer have explicit knowledge of numbers and types of hardware, current task loading, and the such would bring the state of programming back to the "dark ages" of the back room programmer with his bag of software tricks. A compromise has to be met where modern software engineering techniques can be combined with the means to express parallelism in such a way that applications programmers, with limited knowledge of the underlying architecture, can effectively develop software.

Rendezvous techniques, such as addressed above or as used in Ada, help to some degree. Flow diagrams can be utilized to identify tasks and their interdependencies. Still, how does the applications programmer attack and divide the problem? How does one determine what comprises a task?

A technique used in the development of the applications software for the URV multiprocessor prototype is based on sequential, uniprocessor software development. Using a hierarchical/top-down methodology, a program specification leads to a mainline routine which contains calls to subroutines; which, in turn, may call other subroutines. This process of abstraction develops a hierarchy (or tree) of procedure specifications until the program is adequately defined.

The term remote procedure call [4] refers to the usage of tasks, possibly on separate processors, as subroutines to other tasks. If this concept is expanded to include the capability for multiple remote procedure calls to be made concurrently, parallelism can be obtained. Referring to the hierarchical diagram addressed above, if parallelism can be determined and expressed at any level of the tree, parallel remote procedure calls can be used to realize concurrent operation on multiple processors. The main benefit is achieved through the use of conventional sequential design techniques.

#### (6) URV RTMOS Communications Protocols and Parallel Software Design

As addressed earlier, the URV multiprocessor utilizes a rendezvous communications protocol based upon a producer/consumer exchange. The routines developed to accomplish this protocol, p.poll and c.poll, can be used to form remote procedure calls. Figure 2 gives the p.poll and c.poll algorithms.

From a producer (calling routine) task's viewpoint, these routines can be used either to specify sequential or parallel procedure calls. P.poll is used to initiate a procedure call and pass data if necessary. C.poll's, used to retrieve

```

Producer
P[var(sem)]
If var(P/C) <> C then
  Report Error to System
  var(P/C)=C
Else
  Put var(data)
  var(P/C)=P
End if
V[var(sem)]

Consumer
P[var(sem)]
Save registers
While (var(P/C)=C) and
(not timeout) do
  V[var(sem)]
  Wait
  P[var(sem)]
End While
If var(P/C) = P then
  var(P/C) = C
Else
  var(P/C) = F
  Report Error to System
End if
Get var(data)
Retrieve registers
V[var(sem)]

```

Figure 2

results or signals of procedure completion, are placed for sequential dependencies. A p.poll followed immediately by a corresponding c.poll specifies a sequential call. The calling routine does not continue until the called routine completes and signals. Multiple (grouped) p.poll's followed by multiple c.poll's specify parallel calls.

To design parallel software using these protocols, the hierarchical/top-down design process described previously can be used. However, at each level of the specification breakdown, the choice of procedures called at the next lower level is determined largely on the basis of parallel execution potential. The key to this process is the identification of independent elements of computation. Where the execution or outputs of one element affect (or are required by) another element, there is sequential dependency.

The determination of independent elements of computation is not a trivial job. Certain elements, such as operations on matrices, are obvious candidates. Others may not be. Algorithms may have to be restructured to take advantage of parallelism. It is not clear how a best, or optimal, breakdown can be determined for all cases. An important consideration, however, is the granularity of parallelism attempted. The URV multiprocessor was developed with coarse grained parallelism in mind. Attempting to divide all possible independent elements into separate tasks would yield ineffective results due to poor ratios of data passing to computation times. Although this is one aspect of programming the URV multiprocessor which requires the programmer to design according to architectural criteria, accounting for granularity is unavoidable. Simple rules-of-thumb can make the design criteria usable, however.

To demonstrate the ideas presented, an example follows. This example is a rough approximation of the process that was used to develop the applications software for the first phase URV multiprocessor prototype. Obviously, not

all aspects of the parallel software design process for the URV system have been worked out. The results presented here are from but one of several phases of development.

#### (7) An Applications Example

To adequately demonstrate the prototype hardware and operating system, a representative applications function was required. A control mixer algorithm for reconfiguration of control surface gains in the event of surface failures was selected. This was basically the application demonstrated previously on URV TN17. The derivation of the equation computed is not relevant to the discussion in this paper. Interested readers are referred to [3] or [5] for further details.

The equation to be computed is

$$K_i = (B_i' B_i)^{-1} B_i' B_o K_o$$

where  $K_i$  is the resulting aircraft control surface gain matrix. In the model used,  $B_i$  is a 5X4 matrix,  $K_i$  is a 5X3 matrix, and the quantity  $B_o K_o$  is a 5X3 matrix.

The first step taken is to develop a sequential algorithm to compute the above equation.

- (1) Take the transpose of  $B_i$  ( $= B_i'$ )
- (2) Multiply  $B_i'$  by  $B_i$  ( $= A$ )
- (3) Take inverse of  $A$  ( $= C$ )
- (4) Multiply  $B_i'$  by  $B_o K_o$  ( $= D$ )
- (5) Multiply  $C$  by  $D$  ( $= K_i$ )

For the purposes of this application,  $B_o K_o$  is a static quantity, precomputed and used as a single matrix.

Next, areas of potential parallelism are identified. Obviously, steps (2) and (3), which are dependent upon each other, are independent of step (4). At this level of abstraction, however, no other parallelism is apparent. Further specification of the algorithm is required.

Step (3) requires further definition. Many techniques exist for taking inverses or pseudoinverses of matrices. For the purposes of this exercise, the standard inverse definition

$$A^{-1} = (\text{cof } A)' / \det A$$

is used, where  $\text{cof } A$  is the cofactor matrix of  $A$  and  $\det A$  is the determinant of  $A$ . The algorithm selected to compute this is:

- (3a) Compute  $\det A$
- (3b) For each  $A(i,j)$
- (3c) Compute  $E = \det(\text{minor}(A(i,j)))$
- (3d) If  $i + j$  is odd then  $E = -E$
- (3e)  $E = E / \det A$
- (3f) Store  $E$  at  $C(j,i)$
- (3g) Next  $A(i,j)$

With this breakdown, steps (3b) through (3g) involve independent computations for each individual element of A. Each of the elements, therefore, can be handled concurrently.

This completes the breakdown necessary to specify the software task interactions. Further areas of parallelism exist, particularly in the computation of matrix multiplications and determinants. However, the sizes of the matrices do not warrant parallelism within those operations. The problem has been specified to an appropriate level.

Figure 3 diagrams the resulting sequential orderings and areas of parallelism. Figure 4 shows the algorithm as a hierarchical dataflow diagram. The numbers associated with the arcs represent the ordering of procedure calls within a given level. Calls with equal numbers are parallel.

As an example of p.poll/c.poll usage, the following sequence of instructions comprise the Ki mainline as shown in Figure 4:

```

p.poll      transpose.start
c.poll      transpose.end
p.poll      C.start
p.poll      multD.start
c.poll      multD.end
c.poll      C.end
p.poll      multKi.start
c.poll      multKi.end

```

Again, sequential p.poll's followed by corresponding c.poll's form a sequential call. Multiple p.poll's followed by their corresponding c.poll's form parallel calls.

Now that the application has been adequately specified, the "bubbles" of Figure 4 can be coded into tasks. In this exercise, a total of twenty four tasks were defined. The tasks can be loaded onto the available processors of the system and executed. Again, the URV RTMOS was developed such that any task can operate on any processor. From this, the software design above works for any number of processors in the configuration.

Given the degree of sequential dependencies inherent in the algorithm, the results of the applications software on the prototype were favorable. A 39 percent decrease in equation computation time was measured using three processors ( 1.64 times speed up ). The theoretical limit, given the algorithm, is a 58 percent decrease ( 2.36 speed up ). More realistic models, with more control surfaces and larger matrices, would yield even more parallelism and higher speed up factors.

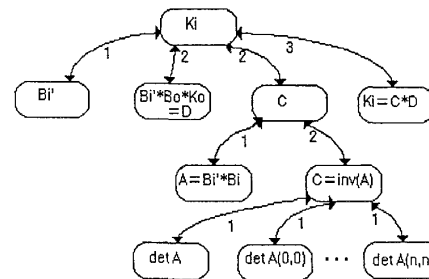


Figure 3

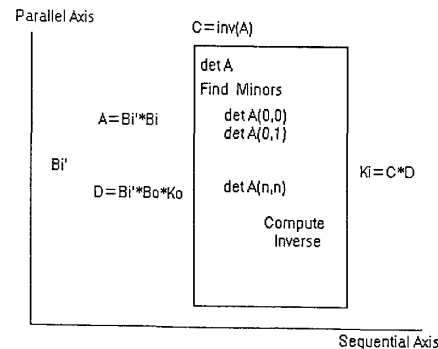


Figure 4

## (8) Conclusions

Parallel software design is not a trivial issue to be addressed after the formation of an architecture. Software engineering techniques have yet to conquer the single processor problem. The construction of parallel tasks into an effective mesh of computation, efficiently utilizing available resources, is a far more complex problem. The design of the URV multiprocessor and RTMOS has attempted to incorporate some basic techniques to assist in the development of parallel applications software.

Still, much more work remains to be done. High Order Language (HOL) issues, such as the usage of Ada, have to be addressed. Some sort of software development tool set or environment would greatly enhance programming effectiveness. Verification issues comprise another complex problem area. The techniques described herein only begin to scrape the surface.

#### References

- [1] S. Larimer et al, "A Continuously Reconfiguring Multi-Microprocessor Flight Control System", AFWAL-TR-81-3070, AFWAL/FIGL, May 1981
- [2] D. Thompson et al, "AF Multiprocessor Flight Control Architecture Developments: CRMMFCS and Beyond", NAECON 86 Proceedings, May 1986
- [3] D. Thompson, "A Multiprocessor Avionics System for an Unmanned Research Vehicle", AFWAL-TR-88-3003, AFWAL/FIGL, January 1988
- [4] G. Andrews et al, "Concepts and Notations for Concurrent Programming", ACM Computing Surveys, March 1983
- [5] K. Rattan, "Study of Control Mixer Concept for Reconfigurable Flight Control System", Final Report, Southeastern Center For Electrical Engineering Education / AFOSR, September 1984