PLURIBUS DOCUMENT 1: OVERVIEW

May 1975

Sponsored by:

Advanced Research Projects Agency
Contract No. F08606-75-C-0032

OVERVIEW

Update History:

Originally written by Severo M. Ornstein, May 1975

INTRODUCTION

Pluribus is more than a machine; it is an architecture and a set of modules for putting together multiprocessor systems. It was originally developed to provide a reliable and modular high-speed packet-switching node for the ARPA Network. The approach taken is quite general, however, and is suitable for many kinds of applications. Pluribus provides a cost-effective way to build computer systems in which reliability, speed, and modularity, or any combination of these, are of importance.

GENERAL PROBLEMS

Traditional computers consist of a central memory system, a processor to execute instructions, and some sort of I/O system. Numerous variations have been developed in attempts to increase speed and efficiency: cache memories, multi-ported memories, hierarchical memories with drums and disks, pipelined processors with look-ahead capability, processors with specialized instructions, and elaborate systems with peripheral processors for handling I/O. In all of these systems the main job is handled by a central processing unit (CPU) working in conjunction with main memory; most embellishments attempt to get more work through this pair of units.

Some of the embellishments — for example, combining peripheral processors with multiported memories — attempt to get various parts of the problem (I/O and the main program) flowing concurrently. Such attempts to achieve parallelism, together with the downward trend of minicomputer prices, have led to experiments in multi-computer systems. Most of these consist of loosely coupled machines where each performs a specific part of the overall job. This solution has weaknesses in areas of reliability and flexibility of load sharing.

If one processor breaks, it typically takes down the entire system until it is repaired or replaced, thus reducing system reliability. Furthermore, in systems of this kind, each Input/Output device is generally associated with a particular function and is accordingly attached to and serviced by a particular processor. When that processor goes down, all access to the device is lost. Manual switchover capabilities can be provided, but usually require several minutes and can involve program reloading. Such interruptions are unacceptable for many real-time control environments.

Limited flexibility of load sharing is also a characteristic of dedicated multiprocessor systems. Try as one will to segment a problem sensibly, certain parts of the system form bottlenecks while others loaf along lightly loaded. Worse still, as loads vary in real time, the bottlenecks will shift from one part to another. Because of the specialization of the processors, lightly loaded ones cannot conveniently help heavily loaded ones (e.g., in servicing I/O devices) with the result that dynamic load sharing is difficult or impossible.

A third general problem area in system architectures is growth. In large machines, expansion rack space, power, address space, etc., are often provided. The cost represents a small fraction of overall system cost. In smaller computers with no massive cost to overshadow such options, all too often one finds oneself suddenly up against hard boundaries — no more I/O channels, no more memory address space, no more power, etc. Furthermore, processing bandwidth (long felt to be the central costly resource) is usually matched carefully to the problem: a system which has a factor of two excess bandwidth to allow for tomorrow's increased demands is simply too expensive a choice for today's problem.

THE PLURIBUS SOLUTION

The Pluribus architecture has been designed to address all of these problems. Before describing how it does this, we will give a brief description of the system and its mode of operation.

The system consists of processor units, memory units and I/O units. Each unit is in fact itself a communication bus providing physical housing, power and cooling, and a primitive communications discipline provided by a "bus arbiter" card for devices on that bus. The number of busses of each type and their exact contents will vary depending upon the application's requirements for bandwidth, reliability, and fan-in/fan-out (I/O). These bus units are coupled together to allow devices on one bus to access devices on another. All processor busses are coupled to all memory busses and all I/O busses; all memory busses are coupled to all I/O busses — as, for example, in the following figure:

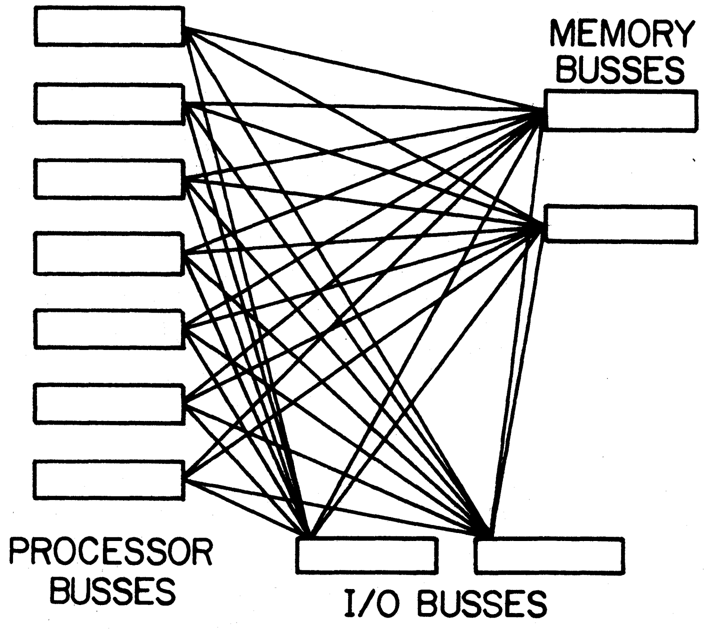

Figure 1 Communication Paths in a Typical Pluribus Configuration

It is characteristic of many programs that relatively small portions form time-critical parts. This is where the program spends most of its time — in so-called "inner loops". In recognition of this situation, a small amount of memory (up to 8K 16-bit words) is provided with each processor on its own bus. Accesses to this "local memory" do not suffer the switching or contention delays that occur in accessing the "common memories". A separate copy of the inner-loop code is typically stored in each processor's local memory.

The tasks to be performed by the processors are generated either by I/O devices calling for attention (completion of block transfers, timeout of a clock, etc.) or by processors spawning further tasks from those already being serviced. In a uniprocessor these are generally announced directly to the processor via a priority interrupt system. In a multiprocessor, however, it is difficult or impossible to select the most suitable processor to interrupt. Furthermore, interrupts, interacting with the resource interlocking mechanism (required in multiprocessors to avoid interference), can produce deadlocks. The Pluribus therefore handles task disbursing differently. Each task is assigned a priority and is associated with a particular flag in a (hardware) priority ordered task disburser known as a PID. When a task is to be performed, its flag is set by whatever unit generates the task. All code is broken carefully into pieces known as strips and each time a processor completes execution of a strip, it returns to query the PID for the most important task to perform next. A single instruction obtains the number of the highest priority waiting task and also erases that task from the PID. I/O devices are assigned specific flags in the PID and set these directly (instead of causing an interrupt). Processors also set

PID flags as the servicing of one task spawns others. PIDs are provided on every I/O bus.

Let us now see how this structure and method of operation relate to the issues of flexibility, speed and reliability.

With regard to flexibility and expandability, it is evident that the general structure is extremely modular. First of all, the busses themselves are modular in that a variable number of units can be plugged into each bus. Bus extender units permit busses to be lengthened to house more units. Furthermore, with a modular bus interconnection scheme, realized via separate bus couplers as opposed to a centralized "cross bar" switch, the number of busses in a system can expand or contract to suit needs. Although large numbers are not often necessary, one theoretically could incorporate dozens of processor busses and memory busses and up to four I/O busses into a system.

In configuring a Pluribus system of this sort, one must study the bandwidth requirements at critical places in the system. How much total I/O bandwidth (to and from common memory) will be required helps to establish how many I/O busses there should be. How much processing bandwidth is needed helps to determine the number of processor busses. The ratio of references to common memory vs. references to private memory establishes how many memory busses are required to support the selected number of processor busses. Thus, one can configure a Pluribus system so that the processors, memory units and I/O units are all internally matched for the application. Other considerations, such as reliability (discussed below), also affect configuration decisions. The point is that the system, consisting as it does of modular busses connected together by a modular switching scheme, forms a very flexible structure. The processors' address space is expanded (by mapping in the couplers) to half a million 16-bit words. Up to 1000 I/O devices are directly addressable by the processors. These limits (of address space) were deliberately set above any visible near-term requirements.

Unique flexibility of hardware utilization is added by the software. We have made it convenient for the program to search for and locate those hardware resources (memory, I/O devices, other processors, etc.), which are present in a system and to determine the type and parameters of those which are found. This makes it possible to construct programs which adapt to running in any of a variety of configurations. For instance, the program can include an algorithm for memory utilization. If enough memory is plugged into a system it will be possible to have ample buffer space, backup copies of all programs, certain helpful but not absolutely necessary programs, etc. Should some memory break, the program can adapt by shifting the utilization of the remaining memory in such a way as to sacrifice the least important functions. Alternately, as memory is added into a system, the program can be made to note the change, test the new resource, and absorb it into the system, utilizing it for the most important function then required. This sort of adaptive operation also relates to the issue of reliability discussed below.

Pluribus system design recognizes that specialization is anathema to reliability and that single copies of key resources are vulnerable points in any system. As opposed to specialization, the Pluribus architecture emphasizes equality. All processors can perform any system function; none is singled out (except momentarily) for a particular function. All processors have equal access to all programs, all I/O units, etc., so that the full power of the machine can be brought to bear on the part of the algorithm which is busiest at a given time. The program can be written to adapt to running with whatever processors are available so that capacity can be increased simply by adding processors and so that loss of a processor incurs loss of capacity but no loss of capability.

With all processors able to perform any system task and a centralized highly efficient mechanism for meting out tasks to processors in priority order, it is clear that service can respond quickly to shifts in demand. Just how quickly this response must be made will depend on the nature of the problem. A communications processor, servicing many high speed lines, requires unusually fast responsiveness. To achieve this, the length of code strips (i.e., the length of time a processor spends between tests for higher priority work) is kept to a few hundred microseconds. In many other systems this time can be relaxed to milliseconds or more without loss of performance.

Since the Pluribus approach to reliability is somewhat unusual, it is important to clarify what sort of reliability is meant. Pluribus systems are reliable in that, although they may suffer momentary outages, they will quickly recover without manual intervention and resume operation — at worst at reduced capacity but with no loss of function. The goal is to eliminate most outages and reduce even the bad ones to a matter of seconds.

In Pluribus systems, the strategy used to achieve reliability comes in two parts which parallel the traditional division into hardware and software. The first part provides hardware that will survive any single failure, even a solid one, in such a way as to leave a potentially runnable machine intact (potentially in that it may need resetting, reloading, etc.). The second part provides all of the software facilities necessary to survive any and all transients stemming from the failure and to adapt to running in the new hardware configuration.

There are two basic strategies in providing the hardware. The first is to include extra copies of every vital hardware resource. The second is to provide sufficient isolation between the copies so that any single component failure will impair only one copy.

To restore the algorithm to operation after a failure, a hierarchical system of software and hardware timers is coupled with a processor consensus system. In addition, a number of disciplines are carefully adhered to in programming which help to reduce vulnerability and limit the insidious effects of errors.

It is instructive to consider what happens when a PID fails. In order to avoid having the system collapse, each I/O bus is given a separate PID (or PIDs). The I/O devices on each I/O bus request service through their local PID and if the PID fails, those devices will be incapacitated just as they would if the power supply (for example) for that bus failed. The processors have equal access to the PIDs on all I/O busses. They typically use only part of one PID for software generated task disbursing and will switch to an alternate PID if the one they are using fails.

From the above, it is clear that loss of a resource central to an I/O bus, such as the PID or the power supply, results in loss of all I/O units depending on that bus. For certain sorts of devices, such degradation is not unreasonable — a section of the machine will be rendered unusable and certain lines or devices will cease to function but the rest of the machine will continue to operate normally. Since some devices are critical, however, and must not ever be lost, controllers and line interface units are designed so that devices can be double connected — i.e., to two controllers on separate I/O busses. In such a case the software will use only one and will switch to the alternate immediately in the event of trouble.

The above discussions highlight an important characteristic of Pluribus systems. Admitting the difficulty and enormous expense of building inherently reliable hardware, we have chosen a most cost-effective means of achieving system reliability — by shifting a major share of the burden of responsibility for both software and hardware reliability onto the software. Sections of the software are specifically devoted to coping with failures. Such "reliability software" concerns itself with failures stemming both from hardware and from software. In coping with hardware troubles (and in performing automatic trouble shooting to locate a problem) it utilizes the redundant hardware resources provided. To cope with active failures, as for example when some processor repeatedly overwrites memory, password-protected, program-controlled amputation switches are provided whereby an actively failing unit can be decoupled from the system.

APPLICATIONS

The Pluribus architecture directly addresses a number of system design requirements involving combinations of greater speed, greater flexibility (expandability) and greater reliability. The Pluribus makes possible systems in which one can more nearly keep up with these requirements without complete replacement or reprogramming by simply adding more parts into the system.

Gain in speed is dependent upon an ability to segment jobs into concurrently executable tasks. Presently this segmentation is part of the job of programming. Whether or not this process of segmentation can eventually be separated and performed automatically is a moot question. The answer will determine the ease with which general, multi-user systems with time sharing, operating systems, etc., can utilize this architecture. It may well be that, eventually, languages will come to include features which provide users with an easy means of describing parallelism in their programs.

In the meantime, the Pluribus architecture seems likely to be used primarily by those whose requirements for speed, expandability, and reliability override the need for general time sharing, higher level languages, etc. Such uses tend to appear in non user-programmed environments with "real time" requirements for speed and survivability. Communications processing, process control, and command and control systems are such environments.