A GENERIC APPLICATIONS SUBROUTINE LIBRARY FOR THE MPP

Michael L. Gough, W. David Wildenhain
Science Applications Research
Lanham, MD

ABSTRACT

A new methodology to increase the utility of the MPP has been developed, and will be presented here as an addition to the current methods of using the MPP. This methodology provides for the development of an MPP-side abstraction layer that is callable from any host-side high level language. Routines in the abstraction layer have the option of using a powerful software tool for accessing the stager as "virtual memory". An additional abstraction layer that allows for remote access to the MPP via DECnet will be discussed. This integrated approach to programming the MPP is a valuable tool for the implementation of interactive user driven systems that require the computational capabilities of the MPP as well as a controlled "user view". It is expected that this methodology will be used to integrate the MPP into many such systems, and thus promote greater use of the MPP by scientific researchers who are accustomed to user friendly environments.

Keywords: Software Design, Virtual Memory, Network Communications, Interactive Systems

BACKGROUND

Although the architecture of the MPP is a departure from the standard Von Neumann architecture, there is no algorithm that can be implemented on the MPP that cannot be implemented on a Von Neumann machine. The only benefit of using the MPP for scientific applications is the dramatic increase in execution speed that is gained for some algorithms. Much research has gone into transforming algorithms so that they can take advantage of the MPP's parallel architecture. With the speed increase that is attainable through the use of the MPP, it is possible to perform heavy computational tasks interactively. It is clear that the scientific community could greatly benefit from such interactive computing power.

This effort represents the first use of the MPP as part of a user friendly system which allows a researcher to activate tasks on the MPP transparently. By using the MPP in this manner, it is possible to provide the researcher with a set of generic software tools which perform large computational tasks within a user friendly interactive environment. The MPP can thus provide a valuable service to the large established group of users who are accustomed to using pre-packaged systems for scientific investigation.

The motivation for the work which led to this methodology was the desire to use the MPP as part of an interactive system which produces animated graphics on a specialized graphics device. This project involved the development of a generic software package that would transform any scientific data set into a uniform quadrilateral mesh. It became apparent that it was impractical to implement such an algorithm on the VAX, and that the MPP was well suited for this algorithm, but it was not clear that the MPP could be integrated into a large operational interactive system.

ENVIRONMENT

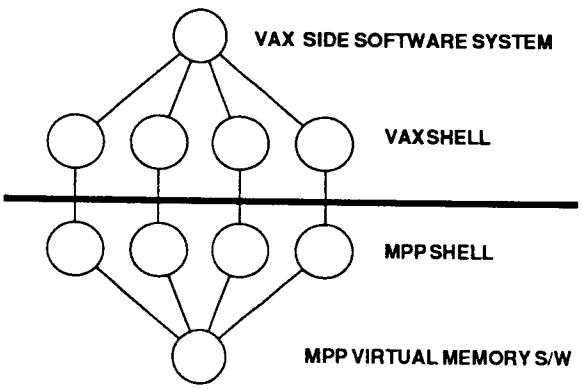
The MPP consists of a 128 x 128 array of microprocessors, each of which is equipped with 1/8k bytes of memory, and an MCU (Main Control Unit) which commands the processors in the array. The MCU has access to 64k of memory, which is used for executable code, as well as data. An auxilliary memory called the stager provides an additional 2k per processor in the array. Early in this investigation it was decided that the algorithms to be implemented on the MPP could not be limited to the 1/8k memory per processor and that the stager would be used, despite the obvious performance cost.

The front end machine for the MPP is a VAX 11/780. The VAX is well suited for user interaction, network access, and I/O to a wide variety of devices. It can also support large software systems that can not be ported to the MPP. Typical MPP applications are loaded into the MCU by the VAX, and activated. Control is not returned to the VAX until the completion of the application running in the MCU. In order to support truly interactive systems, a mechanism had to be devised to invoke various MPP primitives and return control to the VAX for further user interaction. At the discretion of the user, the MPP would be re-activated to perform another task. The overhead for MCU activation is 40 ms. This is negligible when used to return control to the host machine for user interaction.

MPP SHELL

The MPP Shell is a subroutine library of VAX callable routines which activate Parallel Pascal subroutines that execute in the MPP. An MPP Shell subroutine called MPP_open is called to allocate the MPP, and perform required buffer initialization. A subroutine called MPP_close is used to perform all necessary termination, including the deallocation of the MPP. The MPP_load routine is used to load data into the array memory, and the MPP_unload routine is used to unload data from the array to the host memory. The remainder of the MPP Shell subroutines perform computational tasks.

Each Parallel Pascal subroutine is called via a host side header subroutine as pictured below, which issues a call to CAD$START to activate the MPP-side routine.



When the Parallel Pascal subroutine has completed its task, control is returned to the VAX via a call to a specially written MCL routine called HALT. Once control has been returned to the VAX, the applications program can interact with the user so that he can select the next function to be performed on the MPP.

When an MPP Shell subroutine is invoked, the VAX side header routine packs any parameters that are needed by the Parallel Pascal routine into a buffer. The header routine then transfers control to the Parallel Pascal routine (via CAD$START), which reads the parameters from the host using the standard MCU Host Memory Read subroutine. These parameters may include indicies that point to a static Parallel Pascal array. Thus, the result of a previous MPP Shell subroutine invocation can be referenced, and results of the present MPP Shell routine can be placed in the static array for use by other routines. A mechanism to reserve static memory via the Parallel Pascal compiler was developed but is not discussed here; interested persons are encouraged to contact the authors.

MPP-side subroutines that are not set up as VAX callable may be called by the Parallel Pascal portion of an MPP Shell subroutine. Thus, the large library of existing Parallel Pascal and MCL subroutines can be used, and eventually integrated into the MPP Shell Library.

A small FORTRAN code example is shown below, which illustrates the simplicity with which a host side applications program can allocate, use and deallocate the MPP.

```
CALL MPP_OPEN

CALL MPP_LOAD(VAX_real, R1)

CALL MPP_LOAD(VAX_imaginary, I1)

CALL FFT(R1,I1, R2, I2)

CALL MPP_UNLOAD(R2, VAX_real)

CALL MPP_UNLOAD(I2, VAX_imaginary)

CALL MPP_CLOSE
```

Where:

VAX_real and VAX_imaginary are 128 x 128 real arrays

R1, I1, R2, and I2 are integers that point to 128 x 128 real arrays in the static Parallel Pascal array

MPP VIRTUAL MEMORY

As mentioned previously, many applications require more than 1/8 k bytes of memory per processor. The MPP Virtual Memory software that was developed allows for the straightforward use of the staging memory as an extension to the array memory. For applications that cannot avoid the use of the stager, the MPP Virtual Memory package handles page swapping between the array memory and the stager. An LRU (Least Recently Used) algorithm was developed to control swapping of data to and from the array.

The static array mentioned above is used as a cache for the MPP Virtual Memory package. The MPP Virtual Memory package allows the programmer to use the stager and the cache memory as a single integerated memory. For the purposes of the present application, a granularity of 32 bits planes per page was chosen. Before an MPP Shell routine begins performing its task, it ensures that the desired page(s) of MPP virtual memory are "paged in" by issuing a call to "MPPVM". MPPVM checks a list of pages that are currently "paged in". If the desired page is not in the

cache, MPPVM performs all necessary array-stager data movement to put the page in memory. This may involve "paging out" a page that has not been used recently, in order to make room for the desired page. MPPVM returns the physical page number into which it has placed the desired page. If the desired page is already in memory, MPPVM returns immediately with the appropriate physical page number.

MPPVM is sensitive to the access mode that is required by the caller. Three modes are available: read, write, and update. By specifying the access mode, MPPVM can decide whether a given page needs to be physically moved, and thus prevent unnecessary data movement. The overhead incurred by MPPVM is negligible considering the ammount of time that is required by a single array-stager data move.

NETWORK ACCESS

Transparent network access to the MPP has been established through the use of interprocess mailbox communications supported by DECnet. A process on a remote node invokes a COM file on the MPP VAX which implicitly logs on to the MPP VAX and calls the MPP Shell. Packets are exchanged between the two VAXes to pass data and control information. This technique has been used to support interactive MPP applications accross the network.

CONCLUSION

The MPP is a valuable resource for heavy computational tasks which is needed by the scientific community. The tools that have been presented here promote the rapid implementation of user friendly systems that can access the MPP transparently. The power of the MPP can thus become more readily available to a larger user community as an integrated component of user friendly systems.