Virtual Supercomputering on Macintosh Desktop Computers

Ken Krovchuck
Wayne State University
Computations Department - LLNL

Prepared in partial fulfillment of the requirements of the Science and Engineering Research Semester under the direction of John Wade, Research Mentor, in the Lawrence Livermore National Laboratory.

*This research was supported in part by an appointment to the U.S. Department of Energy and Engineering Research Semester (hereinafter called SERS) program administered by LLNL under Contract W-7405-Eng-48 with Lawrence Livermore National Laboratory.

Abstract. Many computing problems of today require supercomputer performance, but do not justify the costs needed to run such applications on supercomputers. In order to fill this need, networks of high-end workstations are often linked together to act as a single virtual parallel supercomputer. This project attempts to develop software that will allow less expensive 'desktop' computers to emulate a parallel supercomputer. To demonstrate the viability of the software, it is being integrated with *POV*, a retracing package that is both computationally expensive and easily modified for parallel systems.

The software was developed using the MetroWerks CodeWarrier Version 6.0 compiler on a Power Macintosh 7500 computer. The software is designed to run on a cluster of Power Macs running system 7.1 or greater on an ethernet network. Currently, because of limitations of both the operating system and the Metrowerks compiler, the software is forced to make use of slower, high level communication interfaces. Both the operating system and the compiler software are under revision however, and these revisions will increase the performance of the system as a whole.

December 12, 1995
Revision 2

Introduction

The wide availability of networked desktop computers has made them a viable option to supercomputers. The power of a large numbers of small computers can be combined to provide speeds comparable to larger, more expensive parallel computers. The maximum speed, however, is limited by the bandwidth of the network, and is also a function of the type of application involved.

Currently, several library interfaces are available for Unix workstations that allow them to emulate a parallel machine. A very popular and freely available interface is known as PVM, or Parallel Virtual Machine. This interface has been ported to supercomputers such as the Cray T3D. Since it is available on both workstations and supercomputers, programmers are able to test and debug code on the less expensive workstations, and then run the production version on a supercomputer. PVM is not available for Macintosh desktop computers. These computers are more widely available than Unix workstations in many workplaces The newer models also provide nearly the same computational power, particularly with respect to floating point operations.

This paper will describe the initial development of a software package designed to create a virtual parallel machine when run across a network of Macintosh computers. This software is called the Parallel Chicken Interface[1], hereafter referred to as PCI. As a proof of concept, this software was combined with a ray tracing package called Pov-Ray. The result, called Parallel Pov, distributes the ray-tracing processes across a network, significantly improving the time required to complete a ray-traced image.

AppleTalk

The Macintosh OS provides networking support through an interface and specification known as AppleTalk. AppleTalk provides an interface to the network, insulating applications from the type of network used, whether it be ethernet, a token network, or other network standard. AppleTalk provides both a low level interface and higher level interfaces to the network. The low level interface simply provides a method to send and receive packets over the network. There is

[1] This name comes from a common analogy used to describe parallel computers, A supercomputer, (the bull) is able to do much more work than a single parallel processor (the chicken), but given a large number of chickens, the total work that can be done can exceed that of the bull.

no checking to see if the packets actually arrive, or arrive more than once. Order of the packets is also not guaranteed. This interface, known as Data Delivery Protocol, or DDP, is also the fastest method of sending packets, (excluding calls made directly to the network hardware). The high level interfaces provide error checking, and packet ordering. They also allow data to be sent as bidirectional streams. These interfaces are known as ADSP, (AppleTalk Data Stream Protocol) and ASDSP (AppleTalk Secure Data Stream Protocol) . Using DDP requires procedures that are called during interrupt time, this in turn requires that the procedures have assembly language entry and exit points which save and restore the CPU registers. This code must be emulated 68020 code on the Power Macintosh computers. The compiler used to develop the PCI software , (MetroWerks CodeWarrier 6.0,) does not allow mixing of 680xx assembly language with PPC601 C language code. In order to utilize the speed of the Power Macintosh computers, applications must used the native, PPC601 instruction set. This has forced the initial version of PCI to use the slower, (but simpler) ADSP AppleTalk interface.

Preemptive Threads

Prior to the introduction of a high speed, RISC based Macintosh computer (Power Macintosh), Apple Computer developed a OS extension known as the thread manager. The thread manager allowed for both cooperative, and preemptive threads. The thread manager does not support preemptive threads on the new Power Macintosh computers, however. Also, a new OS extension, OpenDoc, is being developed that will not be compatible with multithreaded programs. This has led Apple Computer to withdraw support for the preemptive capability of the thread manager and to only support cooperative threads. Cooperative threads require that a program make an explicit call in order for tasks to be switched. Threads must also be executing in the same program space; threads cannot yield time to other processes running on the same machine. This leaves little reason for PCI to make use of threads. Instead, PCI uses a polling loop to determine what process needs to be serviced next. If no process needs service, PCI yields time to other programs running on the machine, which is handled by a mechanism that is separate from the thread manager.

The next version of the Macintosh operating system will allow true, preemptive multitasking. PCI could be updated to use that mechanism at that time, if it proves to be more efficient.

PCI.

PCI, in its final form, will be a library interface that can be linked with programs that wish to use a parallel architecture. It will provide a message passing interface which is purposely similar to the ones provided by PVM and MPI. The similarity will allow programs (or at least their algorithms) to be ported to a Macintosh platform running PCI. Also, many supercomputers are designed to make use of message passing software, and a similar interface to the MPI standard would allow PCI programs, or even PCI itself, to be easily ported to large parallel machines. The PCI interface consists mostly of Send and Receive calls, which are use for the passing of messages to other processes. Other procedures would be provided for the spawning and killing of tasks, determining the ID of a task, and for sending and receiving blocking messages, etc.

The current version of PCI is implemented only enough to allow for the development of Parallel Pov. This was done so that the limitations of the hardware and Mac OS could be well understood before the interface design was frozen. Currently PCI provides for sending and receiving of non-blocking messages to any other process, but does not allow for multicasts. Also, process creation and destruction are not implemented. A small library for building lists of processes is provided, which allows for easier handling of multiple tasks in a non-preemptive environment.

Parallel Pov

Parallel Pov, developed as a test program for PCI, is organized around a master-slave architecture. Two separate programs exist. The pov-master, and one or more pov-slaves. The pov-master is the only program that actually interacts with the user. The pov-slaves, except for initial startup, are controlled internally by the pov-master. There can be up to 254 pov-slaves running simultaneously, though the bandwidth of the network will probably make this impractical.

The general operation of the pov-master program is the same as the original Pov-Ray program, where the user cycles through phases of editing the text description file of the scene to be rendered, and actually rendering the scene. The internal operation of both the pov-master and the pov-slave is considerably different from Pov-Ray.

When a pov-slave program first starts up, it calls routines from PCI which initialize the network interface, and publish the identity of the application to the network. The program then finishes its normal initialization, and makes an asynchronous call instructing the operating system to wait for a connection request. The program waits at this point until the pov-master establishes a connection with it.

When the pov-master program first starts, it scans the network looking for any other program that identifies itself as type 'chick', which corresponds to pov-slave processes which published their identity during initialization. Since this scan is currently done only at startup, all of the pov-slave programs must be started before pov-master is started, otherwise they will be ignored by the pov-master program. Once the identity of all of the processes is received, the master opens a connection with each of those processes (one per pov-slave).

A different list is maintained for each of the different possible states that a processes might be in. Whenever a response is received from a process , it is moved to the list that corresponds to its new state, for example, a process that responds to a connection request would be moved from the WaitingToConnect list to the ReadyForCommand list. Every network call is made in asynchronous mode, a feature of the Macintosh operating system that allows the program to continue operation while the network call is taking place (true multitasking is not available with this OS). The master program polls the status of each of these calls, and responds to them only when they have been completed by the operating system. This allows the master program to service the slave processes in an arbitrary order.

When the user has finished editing the image description file and issues a render command, the master slave program parses the file looking for syntax errors, and if there are none, sends a copy of the text file across the network to each slave process. It also sends a copy of the currently selected options. Once a process has received a copy of the description file, the master sends it a set of coordinates describing the area that needs to be rendered. Each slave process renders a different portion of the image, and returns the result. The master then assembles these pieces into the final image. Theoretically, a speed up that is linear to the number of processors involved is possible, though the top speed of the network, and the constant time spent parsing the data file will reduce the overall speed of the system. Ray tracing is an ideal parallel processing job, since the bulk of the processing time is spent rendering pixels, and the computations for each pixel are independent of neighboring pixels. It was for these reasons that the application was chosen as an initial test-bed for the PCI software.

Implementing Parallel Pov

Even though Parallel-Pov was selected for its easy modification, some portions of the code have proven to be difficult to port to PCI. For example, Pov-Ray provides a mechanism to increase the quality of an image though anti-aliasing. For anti-âliasing to work, whenever a pixel differs significantly from either the pixel to the left of it or above it, that pixel is resampled. Resampling is done at random points very close to to the original pixel. These points are then averaged together to determine the new value of the pixel. The adjacent pixel that caused the anti-aliasing is also resampled. This has the effect of blending the color of the two pixels together, and reduces the jagged edges commonly seen on rasterized images. For PCI however, adjacent pixels are not necessarily known by the process that is calculating the current pixel. That process must also be able to change the value of the neighboring pixel. The most obvious solution to this problem would be to have the process send a message to the appropriate process asking for the value of the pixel. The sending process would then be forced to wait for an answer, or wait for the other process to actually calculate the value of that pixel. Both of these solutions can seriously degrade the performance of the rendering. A better solution would be for the process to ignore the anti-aliasing completely, and allow a separate, clean up process to scan areas that have already been completely rendered, and determine if any of those pixels need to be anti-aliased. That process would be able to adjust the pixels without waiting for any other task to complete.

Another difficulty in implementation occurs with the parsing of the file. Before rendering of the image begins, the text file describing the image is parsed and converted into a structure more usable by the ray tracing engine. Currently, this process is executed redundantly by each process. The algorithm used by Pov-Ray is not yet fully understood (by the author of PCI, at least), and the resulting data structure the parser produces is complex. Also, the parsing is not easily separated into discrete, separate tasks. Attempting to distribute the work would result in a large amount of message passing that would easily bottleneck the network. Instead, the entire text of the description file is sent to each process. The size of the description file is less than the resulting data structure, so transmission of the text can actually be faster that the transmission of the data structure. The current method was chosen for its simplicity of implementation. The other options should be more fully investigated, however, before any final solution is determined.

A third problem in modifying Pov-Ray results from the memory requirements of the rendering. While rendering, each pixel is saved as a set of 4 double precision values, for red, green, blue, and alpha channel data. For large images, this results in a very large buffer to hold all of the pixel values. Pov-Ray does not actually do this. Instead, only the enough information to allow for anti-aliasing to work is held in memory. This is the current and previous line rendered. Once one line is rendered it is stored onto disk and the memory is released. Ideally, Parallel-Pov would store the entire frame buffer. Since each process can return image data at varying times (the time required to render a pixel is not constant), some data can arrive out of sequence with other data. Storing the entire frame buffer allows for easy resequencing of the data. If memory requirements do not allow for this, virtual memory can be used, or synchronization points can be set up where all processes wait until a line of the image is completely rendered. Both of these options would slow the rendering process. The current version of Parallel-Pov buffers the entire image, but the pixel data is reduced to the accuracy of the displaying monitor, which is significantly less than four double float values normally stored.

The development of PCI and Parallel Pov has shown that a virtual machine interface for desktop computers is indeed possible, though the performance of such a system is not yet competitive with existing workstation implementations. This situation may change once the Macintosh platform is more fully ported to the RISC architecture, and emulated calls to the networking interface are no longer necessary. Continued development of PCI would provide an excellent experimental platform for designing parallel programming models, however. Problems that PCI and other message passing systems such as PVM do not address are the problems with parallel programming itself. Programmers are still forced to deal with the complexities of balancing the work load across the different processes evenly. This is partly due to the fact that these PCI and PVM are interfaces to C language programs, a language that was never designed for parallel computing. Development in this area is needed in order for parallel programs to be written in a more productive way.