Can We Make A Good Commodity Parallel Computer?

Peter A. Dinda
Advisor: David R. O'Hallaron

August 26, 1994

A workstation cluster, an isolated set of workstations connected via a high speed network, is a very promising platform for parallel computing. Because clusters are constructed from workstations and off-the-shelf networking hardware, economies of scale make their peak MFLOP/s per dollar ratio especially attractive compared to conventional parallel computers. Further, the cost of a workstation cluster can be amortized over a longer period of time than a conventional parallel computer because when its performance is no longer adequate, it can be disassembled and its nodes used as personal workstations. In many organizations, high end computers are purchased for "power users" and are inherited by less and less demanding users over time. However, the nodes of a conventional parallel computer cannnot be used as personal workstations. This makes the purchase of such a machine difficult to justify unless it is absoluteley necessary, not just desirable. On the other hand, workstation clusters are a no-risk to little-risk way for organizations to acquire parallel computers; thus workstation clusters have the potential to greatly increase the size of the parallel computing marketplace and make parallel computers as much of a commodity as personal workstations.

Are workstation clusters adequate parallel computers? In terms of node hardware, it seems so. In fact, a node on a typical conventional parallel computer looks surprisingly like a workstation. For example, an Intel Paragon node runs applications on an i860 tied to 32 MB of RAM and a second i860 that's used a communication coprocessor. Its peak performance is 75 MFLOP/s. Even more surprisingly, an IBM SP2 node is essentially a rack-mounted POWER2 workstation. For comparison, a typical DEC 3000/400 workstation runs application on a 133 MHz Alpha tied to 64 MB of RAM. It peaks at 133 MFLOP/s. In terms of network performance, workstation clusters are still catching up to conventional parallel computers, but they are getting close. Compared to the Paragon's hardware link bandwidth of 200 MB/s and the SP2's 45 MB/s, switched FDDI's 10 MB/s seems inadequate. However, ATM can provide over 70 MB/s at OC-12 service levels, and Gigabit Nectar will provide a 100 MB/s bandwidth. Further, Gigabit Nectar CABs and some proposed ATM CABs provide what amount to sophisticated communications coprocessors.

Why aren't more people using workstation clusters as parallel computers? One reason is that parallel computers in general are difficult to program. A typical parallel application is explicitly coded as a set of single node programs that communicate with each other via message passing routines. Unfortunately, writing applications in this way (and debugging them!) is more familiar to systems programmers than scientific users. Further, data and work distribution must be handled manually and explicitly coded for, which is difficult, error prone and can limit scalability and portability. Although distributions may scale, they may be non-ideal for a different target machine and changing them is non-trivial. A better way to program parallel computers is to use a language that allows the programmer to clearly express her application in an overall sequential manner while making parallelism explicit and changes to data and work distribution easy.

One such language is High Performance Fortran (HPF.) HPF extends Fortran 90, a language with which most scientific users are familiar, with data distribution directives and a parallel do loop. Data parallelism is expressed using the parallel do loop and array statements. Changing data distribution is trivial in HPF programs, making scalability and portability easier. At CMU, we have developed Fx [1], a parallel Fortran compiler platform for research into the issues of exploiting parallel computers for scientific and signal processing applications. The Fx language combines HPF's data layout directives and array statements with a parallel do and merge construct to allow powerful expression of data parallelism. Integrated into this data parallel framework is a construct for expressing task parallelism among data parallel tasks. For example, a multidimensional FFT can be expressed as a pipeline of purely data parallel stages each of which run 1-D FFTs along a dimension. Neither the Fx language constructs nor the experience gained in implementing and using them are tied to the Fortran base language.

[1] See /afs/cs/project/iwarp/member/fx/public/www/fx.html

The original target platform for Fx was the CMU/Intel iWarp system. I implemented a version of the compiler and run-time library that uses the Parallel Virtual Machine (PVM) library for communication and process control. The base PVM interface has been implemented on a wide variety of hardware. This has allowed us to quickly get Fx programs running on many parallel platforms, including workstation clusters, the Intel Paragon, the IBM SP2, and the Cray T3D. The exercise has also demonstrated the stability and generality of the compiler-generated data parallel communication we generate.

The performance of data parallel Fx applications on workstation clusters proved to be unexpectedly mediocre. This highlights another reason why workstation clusters aren't more commonly used as parallel computers: their actual performance on parallel applications is often nowhere near what would be expected from the hardware. Exactly why this is the case is an open question. I suspect that one reason is the layers of software that isolate the parallel application from the hardware. These layers can severely affect latency and even bandwidth. For example, on the iWarp, the Fx run-time library directly manipulates the communication hardware to achieve high bandwidth, low latency communication. On a workstation cluster, the Fx run-time library exchanges messages using PVM. PVM sends and receives messages via TCP sockets. In short, the entire TCP/IP stack plus the PVM layer is being used to exchange messages between workstations that are only a few feet apart. Measurements of bandwidth and latency on Ethernet and switched FDDI show that the PVM layer itself can eat substantial amounts of bandwidth and latency. Determining the optimal layer at which a compiled parallel program should interact with the networking software is one of my research directions.

Beyond slow networks, it is questionable whether using conventional workstation operating systems, even combined with job entry control, is the right approach to using a workstation cluster as a platform for parallel applications. Many parallel applications are tightly coupled, which implies that indeterminacy on one node affects all the others. For example, if, during a compute phase, one node takes longer than the others because a daemon needed to be run or a page fault occurred, all the other nodes will have to wait. Indeed, a cascading wait could occur if a fast node is waiting on a receive from a slow node and the operating system chooses to switch to some other unimportant, but ready, task. I am currently trying to isolate the impact of operating system overhead and indeterminacy on parallel programs running on workstation clusters.

Task parallelism may decrease the effect of operating system interderminacy because it effectively decouples groups of nodes so that the effect of slow nodes is more localized. Additionally, performance can also be increased by combining the communication locality of small data parallel tasks with pipelining. Our research has demonstrated that in addition to finding the right data distribution, finding the right mix between data and task parallelism is key to maximizing the performance of an application on a parallel computer. Further, combining task and data parallelism can increase the scalability of applications which are scalability limited by one or more routines in a purely data parallel implementation.

Currently, Fx task parallelism is static — everything about a task including how many and which particular nodes it runs on, and whether it shares those nodes with other tasks is determined at compile time. This works well on single user, single task, single thread machines such as iWarp, but may be undesirable on a more complex and unpredictable machine such as a workstation cluster. However, Fx task semantics makes possible coarse grain run-time scheduling because task boundaries are at procedure calls. For example, if after a data parallel task is finished, we notice that one of the nodes it was running on was slow, we could map a faster node to the next invocation of the task. We could also change the number of nodes the task runs on. In order to support research along these lines, my implementation of Fx tasking on PVM computes which array elements to send and receive during inter-task communication of distributed arrays at run-time. Although this is much slower than compiler-generated communication when distributions are known at compile time, inter-task communication has high temporal locality, so caching can be used to amortize the cost of the initial computation.

I am convinced that workstation clusters with fast networks, low overhead communication, and simple and predictable operating systems can make good commodity parallel computers. To a large extent, the hardware to build such systems already exists, but is constrained by operating systems and network protocols that were not designed for parallel computing. Parallel languages and compilers can make programming such systems only marginally more difficult than uniprocessors, but should not sacrifice the ability to express both data and task parallelism in favor of simplicity. The combination of workstation clusters and parallel compilers promises to greatly increase the number of users of parallel computing.