On The Shared Memory Verus Message Passing Argument

7th April 2003

Abstract

While shared-memory machines have traditionally been more popular than distributed-memory machines for High Performance Computing, distributed-memory machines are now seeing a comeback as clusters of inexpensive machines with enough software support to be useful for various server applications. These clusters have superiorities in terms of scalability, availability and graceful degradation as components fail. Here we consider the Internet Services subclass of High Performance Computing and take positions on the physical architecture for these services, as well as the programming model to use on top of that architecture. For the former, our position is to use a cluster of CC-NUMA (or SMP) nodes with high bandwidth interconnects. For the latter, it is to use a message passing programming style.

1 Introduction

For High Performance Computing, shared-memory machines have traditionally been more popular than distributed memory machines. This was because parallel applications could be easily written in shared-memory style, processes running on any CPU could access all I/O, and system administration was simple. However these systems had scaling limitations. Distributed-memory machines were difficult to manage and program in, while not offering a proportional cost advantage. But distributed-memory machines are now seeing a comeback, as clusters of inexpensive machines are gaining enough software support to be useful for various server applications. These clusters have superiorities in terms of scalability, availability and graceful degradation as components fail, because their failure modes are independent.

The positions in this paper are for architecture and programming style for Internet Services, a sub-class of High Performance Computing. We have two objectives; to find the right physical architecture for internet services, and to find the right programming model to use on top of that architecture. For the former, our position is to use a **cluster of CCNUMA (or SMP) nodes with high bandwidth interconnects**. For the latter, it is use a **message passing programming style**.

It is simple to design a message-based ensemble machine, since a lot of issues can be left to software. A shared-memory ensemble machine is desirable because sharing allows both simpler programming techniques and porting of existing serial code. The shared-memory machine is harder to design, but more valuable. It may also be more expensive[2]. The choice has typically been between lots of small nodes in a cluster and a very large shared memory system. The position we take for Internet Services is a cluster of CC-NUMA or SMP nodes with high bandwidth interconnects, such that the ratio of intra to inter node communication latency is not too large, between two and four, for example. Each node can scale up or down from a few to several hundred processors, and the cluster itself can scale up or down from a few nodes to tens of such nodes. Our main motivation is hence scalability of both individual nodes and the cluster. For programming style we favor message-passing because it is easier to build highly available services using this rather than shared-memory style. Our main motivation here is higher availability.

In the remainder of this section we define some ground terms and set out Internet Service design goals against which we will measure different architectures and programming models. For programming models there are two major styles - shared memory and message passing. For architectures, there are three, Uniform Memory Archtecture(UMA), Non Uniform Memory Architecture(NUMA) and the in-between CacheCoherent NUMA(CC-NUMA). In section 2 we elaborate on arguments for and against a shared memory architecture. In section 3 we discuss arguments for and against the message-passing programming model. Finally, we conclude in section 4.

In **Shared Memory** programming the abstraction is of a thread being able to access any word of memory without the assistance of another thread. An implicit assumption is that the placement of data does not significantly affect time to access it.

The **Message Passing** style is more explicit in terms of communication among processors. Partitioning of data among processors is also more important because there are no assumpions about shared memory any more.

In **UMA** systems, CPUs have coherent caches and identical access time to any part of memory. Also, this access time is orders of magnitude greater than cache access time. A shared memory style of programming is more natural on these architectures. It is simple however to simulate message passing on shared memory and so a message passing style of programmig could also be used.

In **NUMA** systems, memory is distributed and there is no direct access to remote memory. It would be expensive to simulate shared memory on such systems since it would have to be done entirely in software (remote latencies will be much higher than local). It is more natural to program in message passing style on these machines.

CCNUMA machines too have coherent caches among CPUs, but the memory is physically distributed. There is a cache consistency protocol that runs on top of this distributed memory. The assess time function is more complex. Remote memory access time is greater than that for local memory, which is itself orders of magnitude greater than that for cache. Also, remote memory access time may or may not vary much according to where it is.

Since our purpose is to take a position on the archtecture and programming style for Internet Services we need to set out objectives against which we will measure different approaches to the two problems[6]. We are concerned with (a) absolute and incremental **scalability**, where a system architecture is scalable if the deliverable performance is a linear function of the number of processors. A succcessful network service must scale to support a very large number of users(corresponds to large processing power), but since there is uncertainty and expense in growing a service, there is also need for incremental scaling that preserves and augments existing investments; (b) **high availability**, the service should match user expectation of twenty four hour availability; and (c) **low cost/performance** ratio.

2 Architecture for Internet Services

The position we take here is a cluster of CC-NUMA or SMP nodes with high bandwidth interconnects, such that the ratio of intra and inter node communication latency is not too large, between two and four, for example. Each node can scale up or down from a few to several hundred processors, and the cluster itself can scale up or down from a few to tens of nodes. This is different from a cluster-based soluion using commodity machines suggested, among others, by [6]. We label and start each subsection with the arguments against our position. We elaborate first on the counter position follwed by an articulation of why (we think) our position is better.

2.1 Google doesn't do it

Google uses thousands of inexpensive PCs running Linux and organizes them into a scalable, reliable, high-performance computing system. It processes over 150 million queries a day, searching a multi-terabyte web index for every query, has an average response time of less than a quarter of a second, and has nearly one hundred percent uptime. If Google can do it with a commodity computer cluster, why can't I?

If you are like Google, in that you have (mostly) read only requests, precompute most of your results and have entirely unrelated accesses then you can use a cluster of commodity machines (And if you have the kind of software to match what Google runs on top of its cluster). By the way, none of Amazon, Ebay, or Yahoo, just to name a few, use commodity clusters. More generally, if there are mostly read-only access, relatively little interprocess communication, low-bandwidth I/O and your workload consists of classical transactions of unrelated accesses then almost any system design will work well. Like a cluster of commodity machines.

But this does not represent all or even a large fraction of internet workloads. If data is split into natural groupings, so all of the data that needs to be together is together, and the CPUs that work on it, all is well. Consider, for example, a system of N small nodes, and the database is split into N approximately equal pieces, and the transactions are retrievals of indpendent records[2]. There is usually some mechanism to start with an input key and route it to the appropriate node. Now if the amount of data is doubled, you could either double the number of nodes or make each node twice as big. Such a mechanism may produce too much imbalance and the disks may need to be rebalanced. In addition to simple retrievals if there are going to be joins on tables that happen to be on separate nodes, with low bandwidth in between, what is simple on a SMP or CC-NUMA is not so simple on a cluster.

More generally, it is important to adapt the shape of the hardware to the shape of the solution and not the other way round[2]. For instance, if one has a bunch of independent tasks, with high ratios of compute-to-communicate, a loosely-coupled cluster is a fine match. On the other hand, if one has a large RDBMS, with many tables and random queries and joins, then it may not be wise to split this data among nodes where the inter-node bandwidth is several times greater than the intra-node bandwidth. Likewise, if the applications is data-centric, and the parts of the database grow at different rates, getting good partitioning that persists is very difficult. A more general solution would be to set up a cluster arrangement dictated by the shape of the data access and interchange, and fault-resiliency arrangements, to start the individual nodes small, and then grow each one incrementally [this is where CC-NUMA comes in], just adding or subtracting CPUs, memory and disks.

Also, it is likely that it will be easier to coordinate a moderate number of CPUs with a large shared memory (for fine-grain sharing), and coordinate the resulting large chunks via message-passing, than it will be to coordinate a large number of individual CPUs with small memories.

2.2 Shared memory machines don't scale

The scalability of CC-NUMA architectures has only been demonstrated for up to 64 processors. And SMP for less than that.

This is true. It is in deference to this, and to get higher availabiliy, discussed in 2.4, that the middle-of-the-road, cluster of CC-NUMAs is suggested. As an example, individual nodes could be the SUN Ultra which scales upto 64 processors or the SGI Origins which can theoretically scale upto a 1024 processors, but are more often used in their 64 or 128 processor versions. The argument is not against using clusters. It is against claiming that commodity (non scalable) nodes should make up the cluster.

2.3 Shared memory machines are too expensive

The SUN Ultra with 64 processors is a quarter of a million dollars without any modules. Also, shared-memory automatically creates contention for frequently used data. Caching is a possible solution, but cache coherence protocols for fine grained machines become incredibly complex, and impact performance. Further, shared memory does not obey the natural laws of physics in a sense[2]. The abstract shared memory model pretends that there is a large number of communication paths to a single point. This isn't true, and additional hardware is required to provide this illusion to the programmer. This hardware costs time and money. Further still, parallel programs designed to reflect the underlying physical reality will operate faster than those that work in a virtual model.

We counter this in two ways. Firstly adding function to an existing chip may be free, from the manufacturing viewpoint. Increasing a design's complexity doesn't necessarily increase its price. Secondly comparison of price must include the actual usabe resources and flexibility of programming[2].

Lets say we want to compare the usable memory in a cluster of 36 single-cpu boxes with 128 MB RAM each, versus the usable memory of a single 36-cpu SMP machine with 4.5 GB of RAM. On the cluster, you subtract at least 16 MB per machine for operating system overhead and you are also very likely to have memory load balancing problems, making it very difficult to pack the memory on each machine. For example, you could end up with with 32 MB free memory on each machine, and no small jobs to use those 36 extra pieces. Even if all the memory is used on the machines, there are likely to be replicated data structures that would not be needed in a shared memory environment. In this case, you are paying for more memory than you actually use. The SMP machine, on the other hand, allows you to use all of the memory with no extra work required for memory load balancing. In addition, if the SMP machine has a 64-bit operating system you can use the entire 4.5 GB of memory for a single job. That job can be serial or parallel, message-passing or shared-memory. Clusters are hence less flexible than shared memory machines, and the costs of this inflexibility need to be accounted for in comparing prices of SMP and clustered solutions.

In subsection 2.1, we saw that when the computaion size exceeded the size of a node (and an individual node cannot be scaled), there were problems. Here we see that if the computation size is just smaller than the size of individual nodes, we get fragmentation, where there is free but unusable memory. In the first case the problem is one of scalability and in the second, one of cost.

As an aside, large CC-NUMA machines can be built with only a modest cost disadvantage relative to small SMPs of comparable capability. If the NUMA interconnect is built into the memory controller, the silicon cost is smaller than having a separate I/O-based cluster controller, and the physical interconnect can be the same as in a cluster, so that cost need not differ[2]. Coupled with the greater scalability of CC-NUMA machines relative to SMPs, this is an argument for using clusters of CC-NUMAs over both cluster of SMPs and clusters of small nodes.

Finally, given scalable hardware, shared-memory programs can be made just as scalable as message-passing programs[1]. It is possible to make shared-memory programs slightly more efficient than message-passing programs because of the reduced need to copy data structures. The details can vary, depending on the hardware support for message off-loading and the latency profiles of the competing systems. In the next section, our position on using a Message Passing style has less to do with scalability or performance than about good programming practices and testability.

2.4 Shared memory machines have lower availability

On a shared memory system you have shared failure modes. The mean time to failure of large SMPs will be very small because a single CPU may corrupt memory and this may cascade to all processes. Components of a cluster have largely independent failure modes and this can be leveraged to get higher availability.

This is a valid argument. Together with scalability, this is the reason why the position is that we use a cluster of CC-NUMAS (or SMPs) rather than a single large CC-NUMA machine. It needs to be restated that the argument is not against using clusters but against using commodity nodes to build it.

2.5 You can get shared-memory support in clusters if you wanted.

You could start with a pure cluster, each node having large local memory, two communication links and a Memory Management Unit(MMU). The MMU can be used to fault on pages absent in local store and let the operating system or application decide how to get it from somewhere else[3]. The programmer could issue hints about the expected outgoing or incoming traffic. The only hardware facilities required are a decent MMU on each node, and page-sized messages. This is already done by some operating systems like the Mach.

Indeed, we may have network-based, page-oriented shared memory in clusters before long. But this would work only for applications with coarse enough granularity to tolerate several hundred microseconds of latency that is incurred in getting a page across a switched fast ethernet (or FDDI). For the same task, shared-memory access times of less than a microsecond is what is more suiltable for a much larger class of applications.

2.6 Arguments that are not made

Finally there are issues where there is seldom an argument. Like, what kind of platform is easier to program for? The shared memory programming style is easier. Easier in terms of amount of time, expertise, and money involved. And in any case shared-memory systems give you the flexibility of programming in a shared-memory or a message-passing style. In fact programming for shared memory is several orders of magnitude cheaper than programming for clusters. Also, a larger number of commerical third-party software applications have been fine-grain parallelized onto SMP systems compared to message-passing systems. Some new SMP systems already have more parallelized software than most message-passing systems that have been around for longer.

This is because for SMPs you start with a static data layout, perhaps derived directly from the uniprocessor version, then parallelize it, and tune by looking at the dynamic characteristics of memory references. The process is often different when using message-passing where you have to first figure out how the static data layout will interact with the dynamic references and there are serious performance penalities for getting it wrong. This may involve considerable effort if the individual memories in a message-passing system are too small for the natural size of the data or computation. Layout of data for caches tends to be short-term, small-size, and dynamic hence easier to do than layout of data for multiple memory systems that tends to be a longer-term, larger-granularity issue, more difficult in the general case.

3 Programming style for Internet Services

The argument about what kind of hardware architecture to use is different from what programming primitives to expose to the programmer. Having motivated the need to have shared memory support, we now advocate a message-passing programming style. This argument is more along the lines of MPI(Message Passing Interface, a standard for message passing) versus OpenMP(a standard for shared memory programming). This is reasonable, since these are the dominant standards for programming in these two paradigms.

Part of the argument in favor of shared-memory support was the flexibility it affords to programming style. We take away some of that by our position on message passing. We feel the objectives for Internet Services (especially that of higher availability) mean we need software that can be easily tested and has well defined interfaces for individual modules. Using a shared-memory style can also lead to conflicts in accesses made by different processors[7]. Again, we label and start subsections with the arguments against our position followed by why our position better handles requirements of Internet Services.

3.1 What if there is large existing serial code and why use message pssing when you have hardware support for shared memory?

Clearly you can take serial code and run it on an SMP machine and get speedup (the compiler assists in identifying parallelism). Translation of serial code to exploit parallelism using message passing usually involves large code rewrites.

Actually, it is not so clear. There are fewer and fewer situations in which you can take code optimized for one architecture and make it run well on another without substantial rewrite.

The MPI model is simpler, because there is a fairly clean and simple separation between within-CPU costs and between-CPU costs[5]. The MPI model is not tied up so intimately with the compiler, so you don't have the problem that the costs can be largely in the overheads produced by the compiler to support OpenMP. Worse, if there is overhead due to support for OpenMP, there are usually no tools available to help you find out why and how, and so reduce them. The code MPI uses is interpreted by almost all compilation systems to be a memory synchronisation point (external function call). OpenMP, on the other hand, relies on unspecified extensions to language standards to synchronize the registers with memory. It must be menioned that this is an artifact of the current OpenMP implementation rather than something fundamental to shared-memory.

Message passing programs are also easier to test and maintain. This compares favorably with shard-memory programs where there are whole classes of bugs which are difficult to find with any tool. Mesage passing on shared memory architectures is discussed in[9]. There are other works such as [8]for combining the shared-memory and message passing paradigms.

4 Conclusion

While clusters of commodity systems are sufficient for a certain classes of Internet Services, there are others that require each node of the cluster to be scalable. Likewise, using a single SMP or CC-NUMA system to build the service doesn't scale and is not highly available. It is important to adapt the shape of the hardware to the shape of the solution and not the other way round. A more general solution is to set up a cluster where each node is a scalable CC-NUMA system. Programming for such machines should however be in message passing style. Message passing programs are easier to test, debug and maintain. and lend themselves well to building highly available Internet Services.

References

[1] Usenet comp.sys.super newsgroup. Message Passing versus Shared Memory thread. December 1995

[2] Usenet comp.arch newsgroup. Distributed Servers versus Big Iron thread. May-June 1996

[3] Usenet comp.arch newsgroup. Is CC-NUMA no longer a good idea thread. May-June 1997

[4] Usenet comp.arch newsgroup. Review of Forest Baskett's talk at UW thread. Nov 1997

[5] Usenet comp.arch newsgroup. Recurring thems of comp.arch thread. January 2003

[6] Lessons from Giant-Scale Services(DRAFT). Eric Brewer. 1999

[7] Role of Message Passing in Performance Oriented Parallel Programming. Vipin Kumar, George Karypis, Ananth Grama. UMN TR_97-040. 1997

[8] Shared Memory vs Message Passing in Shared Memory Multiprocessors. Thomas LeBlanc, Evalgelos Markatos. University of Rochester 1992

[9] The Stanford Flash Multiprocessor. J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simon, K. Gharachorloo, J. Chapin, D. Nakahira, J.Baxter, M.Horowitz, A. Gupta, M. Rosenblum, J. Hennessy. Proceedings of the 21st International Symposium on Computer Architecture. April 1994