Experiences with the SGI/Cray Origin 2000 256 Processor System Installed at the NAS Facility of the NASA Ames Research Center

Jens Petersohn and Karl Schilke
NAS Facility NASA Ames Research Center
jkp@nas.nasa.gov
rat@nas.nasa.gov

ABSTRACT: The NAS division of the NASA Ames Research Center installed a 256 processor single system image Origin 2000 system, the first at a customer site, during the latter half of October 1998. Due to the large number of processors, the system exhibits different operational characteristics than the smaller Origin 2000 systems. The causes of the observed behavior are discussed along with progress towards solving or reducing the effect of problem areas.

Characterizing the Machine

NUMA systems, specifically shared memory NUMA systems, offer the tantalizing benefits of both the ability to construct large systems from a building block system architecture and a common address space in which software may operate.

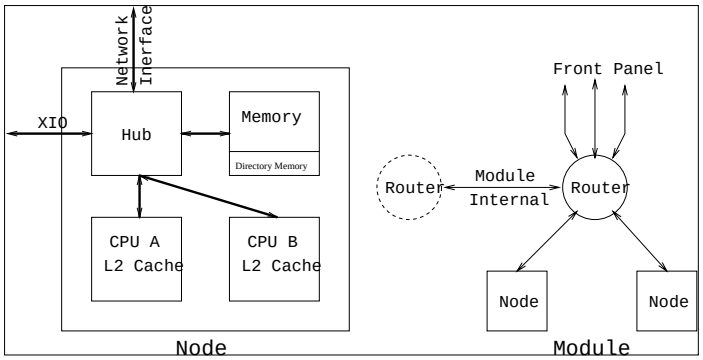

Figure 1 – Basic organization of the Origin 2000 computer.

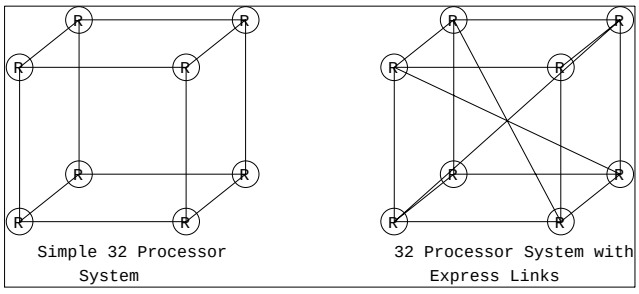

Figure 2 – 32 processor systems.

Additionally the shared memory NUMA system seemingly solves the problem of memory buss contention in true symmetric designs. The SGI (formerly Silicon Graphics, Inc.) Origin 2000 series computer, along with the Hewlett-Packard Exemplar and the soon forthcoming Sun "Starfire" system, are the first generation of these NUMA systems that attempt to exploit the above mentioned benefits. The dynamics of using these systems, especially for the larger processing element counts, from a software/programming and a system administrative perspective, are relatively poorly understood.

There are a number of possible designs that manufacturers may choose to implement the NUMA philosophy. These can range from a SMP emulation to relatively loosely coupled architectures. This paper concentrates on the SGI Origin 2000 system, which is of interest because it was the first NUMA design to appear on the market that provides an SMP like architecture. The Origin 2000 may in fact be used/programmed exactly like an SMP machine, however the fallacy of doing so will be discussed in more detail.

The Origin 2000 is a shared address space, cache-coherent, non-uniform memory architecture machine. The shared address space and cache coherency bestow the SMP-like appearance on the system. The term "shared address space" implies that all memory in the system, and even the I/O address space, is visible from all other compute elements in the system. The system even supports the use of "headless nodes", which contain memory but no processing elements. This configuration is, however, ill suited for efficient execution of computationally intensive software in most cases.

The system is constructed by an assemblage of nodes, which provide a memory element (32 to 4096 MB) and zero to two processing elements. The most common processing element at the time

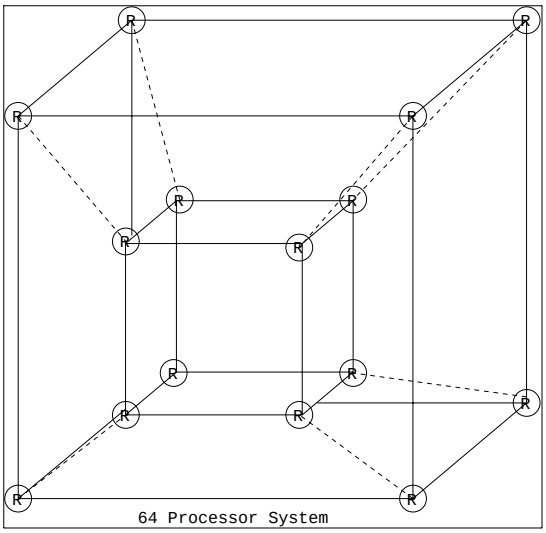

Figure 3 – 64 processor system.

of writing is the 250 MHz MIPS R10000 chip. This chip has a 32 kB data and 32 kB instruction primary cache, and supports up to a 4 MB secondary cache, with a cache line size of 64 or 128 bytes. On the Origin 2000, a 128 byte cache line size is always used. In addition to the processors and memory element, each node is equipped with a hub chip and, optionally, external directory memory. The hub chip mediates access to memory from the secondary caches on the processing elements and also provides the network interface for communicating with other nodes. Note that after initialization the hub operates independently from the processing elements on the node. Access to memory is performed transparently to the CPU by

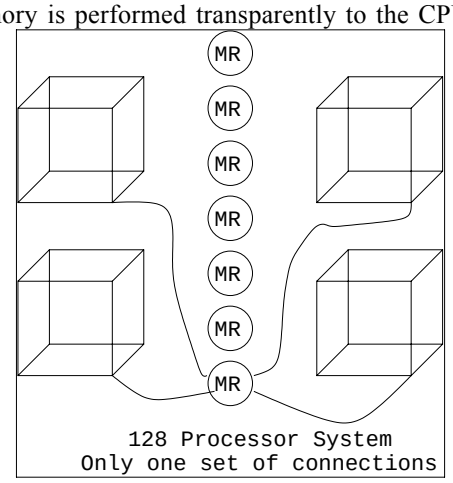

Figure 4 – 128 processor system.

retrieving the desired memory chunk, which is always cache line sized, and returning it to the cache of the requesting CPU. If the hub detects that the presented physical address falls outside of the range of addresses assigned to this node, it will communicate via the Craylink network with the node that contains the memory and place the result in the cache of the requesting CPU.

The directory memory, either stored in the main memory, or for larger systems, in a special directory memory, marks the "owner" and modes of a memory chunk. The directory memory is managed by the hub chip. When a memory chunk is loaded into the secondary cache of a processor, the node on which this processor resides and the access mode is recorded in the directory memory. The access mode (shared, exclusive, busy shared, busy exclusive, waiting) is used to determine the action required if a second access attempt is made to the same memory.

The hub chip connects to one of six ports on a router. A module, which can contain up to four nodes and two routers, is internally wired to connect two nodes to each router and the two routers to each other. (Refer to figure 1.) Thus, no external cabling is necessary for a single module of

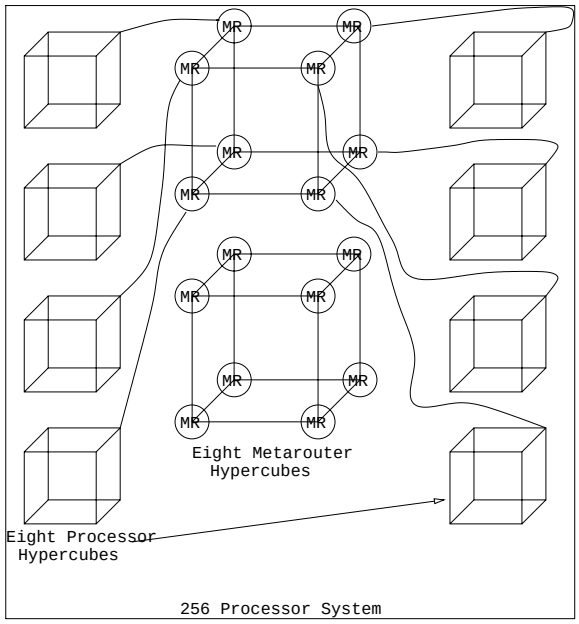

Figure 5 – 256 processor system.

up to eight processors. The remaining three ports on each router device are accessible via the front panel for external connections.

The nodes are connected to the router and the routers to each other via a special type of network termed the "Craylink" interconnect. This network operates with small variable sized messages which contain a source and a destination. Tables established at boot time inside each router determine the routing through the network. Because of this, a failure of a link, router or cable, can be recovered from by rebooting the system. The

routing through the network could potentially be modified dynamically, but this is not done at present.

Routers are connected to each to other in a hypercube topology with routers forming the vertices. A full three dimensional hypercube is formed by eight routers, requiring four modules. Since each router can be connected (internally in the module) to two hubs, and each hub contains up to two processing elements, a full three dimensional hypercube consists of 32 processors. In this configuration, five of the six ports on the router are occupied (three internally in a module and two externally to interconnect the four modules). The remaining port is used for larger configurations or Express Links. Express Links offer additional connectivity inside a three dimensional or smaller hypercube by cross-connecting the opposing vertices. (Refer to figure 2.)

A larger system of 64 processors can be made by connecting the like vertices of two 32 processor hypercubes with the last remaining router port in a four dimensional hypercube. (Refer to figure 3.)

As all ports are exhausted by the 64 processor system, additional routing devices that are not connected to hubs, and may use all ports for routing interconnections, were introduced. To form a 128 processor system these routing devices, "metarouters", are used to provide a four way switch between four 32 processor hypercubes. Four of the front panel ports on each metarouter are connected to like vertices of the four processor hypercubes. In this configuration the metarouters are not interconnected with each other. (Refer to figure 4.)

The 256 processor Origin 2000 presents a departure from these basic configurations. In the 256 processor and larger systems, the metarouters are themselves interconnected in a hypercube configuration. Eight full three dimensional hypercubes are used, for a total of 64 metarouters. These metarouter hypercubes are not interconnected with each other. The vertices of the metarouter hypercube are connected to the same vertex of each processor hypercube. For example, each vertex in the metarouter hypercube "front top left corner" is connected the front top left corner of each processor hypercube. (Refer to figure 5.)

Metarouters have four exposed front panel connections and two internal connections. For the 256 processor system the four external connections are used exactly as on a 128 processor system. The two internal connections may be used to build an even larger system, but they have not yet been used in the field.

The complex network geometry of the 256 processing system necessitates the need for an alternate boot PROM that supports topology discovery of this complex network. SGI has termed this PROM the XXL PROM.

Bandwidth and Latency of the Craylink Network

The previous discussion is primarily intended to familiarize the reader with the architecture in preparation of this section. In order to understand the implications of large processor count Origin 2000 systems, it is necessary to understand the "construction" of those systems.

Although the hardware architecture offers substantial flexibility, it does not come without tradeoffs. The Craylink interconnect network used by these large systems is designed to scale the cross-section bandwidth linearly with the processor count, but it does so at the cost of higher latency.

In the most minimal configuration, a network message must travel through at least one router in order to reach it's nearest neighbor in the network. A load of a cache line from local memory requires approximately 485 ns to complete. Each router will add 50 ns each way to that time. Thus for an eight processor system, in which the most distant hub (and thus memory) is two hops away, the additional delay is only 30% of the total latency and these small systems provide a reasonably good emulation of a SMP system. Table 1 shows the worst case hop count and approximate additional latency for unowned memory.

Processor Count	Worst Case Hop Count	Additional Latency	Total Latency
Local Memory	0	0 ns	485 ns
4 or Less	1	100 ns	585 ns
5-8	2	200 ns	685 ns
9-16	3	300 ns	785 ns
17-32	4	400 ns	885 ns
33-64	5	500 ns	985 ns
65-128	6	600 ns	1085 ns
129-256	9*	900 ns	1385 ns
9-16 Express Link	2	200 ns	685 ns
17-32 Express Link	3	300 ns	785 ns

*Additional hop count caused by traversing metarouters.

Table 1 – Latencies of the Craylink network.

The worst case on a 256 processor system presents almost threefold increase in latency over a local memory access. For this reason, viewing the 256 processor system as a large SMP system is fallacious.

Additionally, there is the possibility for hotspots, links that are extremely busy because of two high-traffic activities occurring in different areas of the machine, to develop. The choice to maximize cross-section bandwidth does limit this contention in most cases. There are some simple tools available, such as linkstat(1), that can be used to monitor traffic if hotspots are suspected. Should hotspots be occurring, the above mentioned latencies may increase.

As previously stated, the latency times presented are in fact ideal cases, in as much that they assume that the memory chunk being retrieved is immediately available for transmission. This is not always the case. If the element is already "owned" by some other hub, that is one of the processors connected to a hub other than the one requesting the chunk, then action may have to be initiated to guarantee coherency of the memory chunk. Memory chunks are basically categorized into to "clean" and "dirty". A "clean" chunk has not been modified in any cache, thus the contents of all caches that contain a copy of that chunk match. Should any owner of a chunk modify it's copy of the chunk, the chunk becomes "dirty" and all other copies must be flushed from all caches that contain a copy of that chunk. If any hub tries to retrieve the chunk, the chunk must be "cleaned", that is written back to memory from the owner of the "dirty" copy, before the retrieve access can be completed. This scenario is frequently termed a "cache collision". This is somewhat of a simplification of the actual process (and memory ownership "modes"), but suffices for the purposes of this discussion.

Coarse Mode

Each cache line sized chunk of memory has a 64 bit vector associated with it. These vectors are stored in the directory memory along with "mode" bits. When a memory chunk is retrieved a bit corresponding to the hub which retrieved this chunk is set along with a mode bit to indicate the disposition of this memory chunk. For larger systems, those that have more than 64 hubs (nodes), such as a 256 processor system, the 64 bit vector is too small to hold all possible hubs. An alternate operating mode is used which is referred to as coarse mode, in which eight hubs are represented by a single bit in the owner vector. As a result, eight hubs will receive a cache invalidation message due to a "dirty" cache access and cause action to occur on eight caches, even if they do not actually contain a copy of the memory chunk in question. The invalidation request is tested against the contents of each cache and ignored if that cache does not contain a copy of the chunk. A response message is generated by all eight hubs and must be received and processed by the sender. Thus, access to a "dirty" chunk is considerably slower than access to a "clean" or unowned chunk.

A directory entry for a memory chunk enters coarse mode if any access to that memory is made outside of the 64 node piece of the machine that the memory resides in.

This expanded operating mode supports up to 512 hubs or nodes, for a maximum processor count of 1024. The standard mode addressing scheme used in the network messages however limits the maximum node count to 256 since the address field is eight bits in size.

The NAS 256 Processor Origin 2000

The system was installed in late October 1998. The base components consisting of two 128 processor systems had already been at the site for several months and operated as two systems. The metarouter cabinets and cards were delivered a few weeks prior to the installation of the 256 processor system.

As this system was constructed from existing smaller systems, workload requirements dictated that this system had to be operated similarly to the smaller systems at NAS and could not be dedicated to special purpose programs. This implied a highly varied workload with codes of very variable quality, especially with respect to the caching behavior and the amount of communications required between threads. Additionally, as little work had been performed on most codes to scale them to larger systems, and to meet the above mentioned work load requirements, multiple programs, jobs, had to be executed simultaneously. This presented a unique and interesting problem, both from a management and programming standpoint.

One of the most notable problems is that most codes exhibit highly variable runtimes. Investigation of this and other problems have pointed out issues in the system software, workload management and the user/programmer's perception of the system. Each of these will be discussed in more detail in the following sections.

Operating System Issues

The operating system version that was delivered with the system in October 1998 was IRIX 6.5.2XXL. This version of the operating system included special modifications to operate on a system with more than 64 nodes.

Surprisingly, the stability from both the operating system and the hardware standpoint has been better than that of the smaller systems that comprise the 256 processor system. Most of the exceptions revolve around special features or unusual operating conditions. The site has been actively integrating new operating system facilities such as Miser into the batch management system "Portable Batch System" (PBS). Unfortunately, until recently, enabling this capability resulted in operating system crashes. Another case that reliably crashed the system is a program that allocates a small memory on a single node and accesses this memory from many threads, such as 100 or more. This case will trigger various hardware bugs in the hubs and routers and cause the system to hang or crash. Outside of these two primary causes, there have been a few unexplained random hangs or crashes, but not occurring at a frequency were they cause concern.

As was previously mentioned, one of the major problems is the variation in execution time on the system. The primary symptom underlying these variations is poor memory layout predictability. Stated another way, the memory used by each thread is not predictably placed on the primary node on whose CPU(s) the thread is executing. This results from both memory placement during memory commit operations and unpredictable thread/CPU locality. Threads may execute on several CPUs, especially during their startup phase where they are allocating and first touching (and thus committing) memory. The MLD (Memory Locality Domain) facility in the kernel is meant to provide a programmatic interface to the memory management subsystem for user codes, but it is non-trivial to use and has not exhibited the desired robustness in guaranteeing the requested memory layout. Especially for codes that exhibit only marginal caching behavior, this is a serious problem. These issues are being actively pursued by both SGI and NAS personnel in an attempt to resolve them as rapidly as possible as they affect all uses of the system.

The above conditions also result in cross-job interference problems, even for codes that are nominally well behaved. Once memory locality is broken for one job and its memory is obtained from some other section of the machine, whatever is executing or will be executing on the remote part of the machine will be negatively impacted. This is also true for ill-behaved jobs, namely those which use more CPU or memory resources than the user indicated to the batch management system at submission time. Clearly better controls are required that allow for "hard" walls around individual jobs. Ideally these hard walls would be dynamically re-configurable, which implies that they must operate within a single system image. The Miser facility addresses this partially, but does not provide sufficient control over the memory component.

Workload Management

Much of the development effort of the Portable Batch System was aimed at providing workload management on homogeneous systems (distributed or SMP). Different codes have differing requirements in terms of their "connectivity" to sibling threads inside a given job. Codes that are

ported from SMP or vector systems tend to assume better connectivity than those originating in a cluster environment. In the ideal case, the batch management system should consider not only the amount of resources available, but also the "quality" of the resources. The user would communicate his or her requirements to the batch management system which would then attempt to make intelligent decisions by looking at the requirements of the codes awaiting execution, and the ability of the resources to meet these requirements now and in the future. Clearly this requires that the user understands the needs of the code that he/she is using, something that is not necessarily possible as many users use codes developed commercially or otherwise externally.

With limited development resources available, this area has not been addressed, and this deficiency is also largely masked by other problems.

Existing Codes

Resolving the memory layout issues mentioned only partially addresses the lack of performance and runtime variability problems that are observed.

The IRIX operating system considers all executing "normal" threads to be timeshared threads. This, combined with thread interference between jobs and unpredictable thread execution locations, results in highly variable execution rates of each thread. For instance, if the threads are forced into synchronous lockstep during their parallel loop iterations, the performance of the entire job is dictated by the last straggler thread that finally arrives at the barrier (explicit or implicit) that is synchronizing the threads. This has been frequently identified as a primary cause of scaling problems in codes that otherwise exhibit almost perfect behavior and are prime candidates for up-scaling the number of threads.

Ultimately, this problem needs to be addressed from the operating system perspective, so that more reliable loop iteration times can be guaranteed. At present, however, resolution of this problem requires reduction of the number of synchronization events in the code overall or re-distributing the work (and rewriting parts of the code) so synchronization is only required between threads in smaller independently operating groups. A model that has been applied successfully to this end is the MLP or multi-level parallelism concept, described below.

The code must also exhibit reasonably clean separation of memory objects between threads. If this is not true, then not only is caching behavior negatively impacted, but the memory from which the cache is loaded is more likely to be remote. Optimally, each thread should perform calculations within a block of memory that is page aligned and a multiple of the page size. If that is not the case, at least one thread will be accessing remote memory for the overlap page, which is a minor problem for small pages, but becomes a bigger problem if large pages are used, which is desirable to produce better Translation Lookaside Buffer (TLB) behavior.

Most importantly, the pages must also be touched by the thread that will eventually use them, and not by some parent or master thread. If memory is first committed by some parent or master thread, the individual threads will most definitely not have these pages local to their execution CPU. Page migration can theoretically mitigate the resultant non-locality problem, but it has not yet been proven of value at NAS. Better thread execution location stability is required in order to make page migration effective.

Programming Paradigms

The most frequently used programming paradigms at NAS are the SGI compiler parallelism and the Message Passing Interface (MPI) API (SGI version). The compiler parallelism is implemented similarly to the compiler parallelism found in the Cray UNICOS compilers. Because of this, it is frequently used in a fashion that reflects the Cray programming paradigm, namely that of small scale (micro) multi-tasking. There is no inherent requirement that it be used in this fashion, but as many codes were ported from Cray vector systems it is quite often the case. Aside from penalties associated with setup when entering a parallel section, and of course the implicit barrier and cleanup at exit, this programming model in many cases results in poor understanding of the memory access patterns by the programmer. This leads to large numbers of cache collisions and off-node memory accesses, causing poor scaling behavior on a large systems with reduced connectivity compared to smaller systems.

The MPI API does require the programmer to explicitly transfer the shared data across multiple threads, but the most frequently used MPI calls imply barriers since a normal receive operation does not return until the sender has transmitted the data and it has been placed into the receiver's memory area. If many blocking MPI calls are made, especially during tight loops, the implicit barriers will result in a significant performance degradation that can be avoided with asynchronous MPI calls. The cause of this is largely the

unpredictable loop iteration times previously discussed.

Clearly, if used correctly, either model should work well if the program exhibits good memory access patterns and minimizes implicit and explicit barriers. For this to be the case, the programmer must have substantial awareness of the characteristics of the machine. A new programming model that would implicitly enforce these requirements, while also providing the necessary low level code to assist the kernel (and ultimately the batch management system) in guaranteeing more predictable memory and thread layouts, would greatly ease the use of the NUMA architecture as a high performance computing platform.

For codes based on suitable algorithms, a multi-level parallelism approach works well. Essentially, the work performed by the program is broken into micro- and macro- components. Each macro-component is processed by a semi-autonomous group of threads that internally use micro-multitasking to process that component. If these groups of threads are laid out on the machine in such a way as to maximize the connectivity of the micro-multitasking threads, within one 32 processor hypercube or more ideally only one or two router hops away, they will operate almost as well as on a true SMP system. This is especially true if the micro-multitasked threads exhibit good caching and TLB behavior. The groups themselves operate autonomously from each other and only communicate rarely, such as to synchronize after the completion of the components by the micro-multitasked threads.

Unfortunately, this paradigm only works marginally well in the current batch environment, since no guarantees are made as to the connectivity and layout of the assigned nodes/CPUs. MLP is performed at present by either mixing two parallelism models (MPI and compiler for example) or by using operating system services directly to provide the top level parallelism. Although some of the necessary code could be packaged into a library, a more coherent single paradigm for programming is really needed. As mentioned above, the support code for such a new programming paradigm can be developed to interact with the batch management system and the operating system. This would provide for a much more robust execution environment for large thread count applications. The development of such a programming paradigm is being investigated by NAS personnel at this time.

Current Projects

The following areas are being actively pursued by SGI and NAS personnel. A rough estimate of the time to completion is provided.

Memory and thread placement and stability: This will insure better thread-to-memory locality. It is being heavily investigated by SGI personnel in Eagan and some improvement should be available in the short term.

Improved separation between unrelated jobs: Harder walls between unrelated jobs concurrently executing on the machine are needed. Optimally this should be dynamically adjustable by the batch management system. NAS personnel has integrated Miser CPUset functionality directly into PBS which will reduce the errant thread interference problem. Forthcoming fixes from SGI (and in the intermediate term NAS) for the operating system will extend that capability to memory as well.

Coherent Resource Management: This is being looked at to provide a more consistent view of resources to a batch/workload management system. SGI and NAS are looking at possible alternatives. This is a long term project.

“QoS” Resource Management: As large thread count codes will have specific requirements for location/layout on a machine, the batch/workload management system will need to have some type of “quality” system to insure the application receives the desired resource layout. Better understanding of the needs of applications and support from their multi-tasking method, as well as how to communicate these requirements to the batch management system and kernel, is required. This is also a longer term project.

Multi Level Parallelism: The possibility of extending MP/OpenMP is being looked at as a vehicle to provide a programming paradigm for multi level parallelism. This vehicle could provide the required support for the “QoS” resource management mentioned above.

Closing Remarks

The possibility of very large processing element count systems based on the NUMA architecture is certainly an intriguing one, especially in a single system image environment. There are definite advantages of such systems. From an application standpoint the connectivity, both in latency and

bandwidth, is a vast improvement over what is possible in more loosely coupled systems.

The single system image environment certainly eases the administrative burden and makes intelligent resource management a much simpler task.

Once the pressing issues that currently hamper the usability of the 256 processor Origin 2000 system have been resolved, and the techniques for programming for this architecture are better understood by application programmers, this type of system will provide an excellent high performance computing platform for the computational sciences. In many ways it does represent a sizable change from the traditional "bread and butter" Cray vector systems, but the much lower cost and the performance potential make the NUMA systems an attractive alternative.

References and Acknowledgments

Much of this information was obtained by observing the machine and the programs running on it. Information on the architecture, notably the messaging system and routers and hubs was gathered from lengthy conversations with SGI personnel, particularly Bron Nelson, Scott Emery, Jim Harrell, and Jack Steiner. Some was obtained by reading the appropriate design specifications for the hardware. Most information about the IRIX operating system was gathered by studying the source code.

Actual references to the SGI internal documents may be obtained (by SGI personnel, or with their permission) by contacting the authors.