Early Experiences with The 512 Processor Single System Image Origin2000

Robert B. Ciotti (ciotti@nas.nasa.gov),
James R. Taft (jtaft@nas.nasa.gov),
Numerical Aerospace Simulation (NAS)
NASA Ames Research Center, Moffett Field, CA 94035
Jens Petersohn (jkp@sgi.com),
SGI
1600 Amphitheater Parkway
Mountain View, CA 94043

ABSTRACT: NASA Ames installed a 512-processor single system image Origin 2000 system in October 1999. Over the past several months, NASA Ames has worked to develop operating system support, a batch-scheduling environment, programming techniques and libraries that serve to maximize the utility of large shared memory supercomputers. Early results suggest that large shared memory machines are viable parallel supercomputers and serve to provide a simplified and more effective programming alternative to MPI while also reducing the communication latency in parallel applications. Sustained CFD simulations exceeding 60 gigaflops or 13 times the performance of a 16-processor C90 system were achieved. MTTI and Gross Availability are shown and compared for several different systems and show good correlation between part counts and reliability – reliability of a 256p Origin2000 system is shown to be better than a 16p C90 vector system.

1 Introduction

NASA Ames installed a 512-processor Origin2000 system over the fall of 1999. This system was known to be a one of a kind and would never be offered as a product by SGI, but would serve as a prototype system to develop software (applications, batch systems, and operating systems) in preparation for the next generation Scalable Node (SN) system available in 2000. Further, it would serve as a proof of concept machine to show whether large shared memory systems could be built and operated at all.

Even though the Origin2000 architecture was capable of supporting configurations larger than 256 processors, there were several risk areas in pushing the technology to 512. 512 is the largest configuration this generation of machine could reasonably support, due to the complexity, latency, reliability, and cost

Significant technological risk was associated with the operating system and also in untested transistor logic that would be required in routing. Other issues such as the physical layout or the PROM topology discovery code were also significant hurdles to overcome.

All of these issues were successfully dealt with over the course of several months during the summer of 99. The prototype system built in Eagan consisted of 256 250mhz R10k processors and 256 300mhz R12k processors and was assembled in August 1999. 2 256-processor 300khz R12k systems were installed at NAS in July, waiting to be cabled together as a 512 scheduled for late September. This provided time to break in the systems and shake out problems that normally occur on any new system.

The 512-processor system came up in the first week in October and was soon running benchmarks (Linpack and Overflow). There were a few initial problems with stacklimits and MLDSETs. but these issues were quickly dealt with and the system was made available for application development within the first week.

2 Industry Trends

Generally, we see a trend in forming in shared memory computer systems. Computer industry momentum is toward larger CPU count symmetric Multi-Processing (SMP) systems built around Non-Uniform Memory Access (NUMA) memory sub-systems. Vendors have been forced to the NUMA approach for both technical and business reasons. Technically, the high clock rates found in systems today require very short point-to-point hardware communication paths. Bus or crossbar based memory designs are either too costly or simply cannot scale to large CPU counts within the physical space or path lengths required. NUMA is the technical solution to this problem, and will likely dominate commodity parallel system designs in the coming decade.

From a business perspective, vendors like the building block approach possible with NUMA designs in that

large systems are simple amalgams of many smaller identically manufactured building blocks that are connected together via inexpensive cables. This offers economies of scale in the manufacturing process and assures the volume necessary to sustain a viable commercial enterprise.

Having an operating system that can effectively deal with hundreds, or even thousands of processors is another story and represents the most significant technical challenge that the vendor community faces in building these systems.

Supporting shared memory in hardware without cache coherency is trivial. This functionality should be pushed out to the network interface and even standardized across vendor platforms. Of course, cache coherency is advantageous for application development in many instances and Origin2000 has shown it possible on large scale systems.

The decision to pursue this technology trend was based on benchmarking that demonstrated clear performance and price performance advantages over clustered and MPP systems (e.g. SP and T3E) for NASAs large scale computing problems. Further, SGI was and still is the leader in delivering high processor count, ccNUMA technology to market.

2.1 General Overview

The Origin2000 system is a cache-coherent non-uniform memory access (ccNUMA) computer. That is, all memory within a Single System Image (SSI) is globally addressable and cache coherent for all processors within that SSI. However, the processors far away from the memory they access (i.e. many hops) experience greater access latency than those processors that are close to their memory. In general, this latency is approximately 100ns/hop + 485ns. The best-case hop count on any Origin2000 system is 0 hops, and the worst-case hop count in a 512p system is 10 hops. So, bringing a clean/unshared (i.e. no invalidation necessary) cache line from memory into the processor can take anywhere from 485ns to 1485ns. This resulting factor of over 3x in latency differential has a major impact on how the operating system works, what applications must do to efficiently use the system and finally, how the batch queuing system allocates resources to user jobs.

However, the worst-case shared memory access latency is 10x faster than explicit messaging via MPI on Origin architectures. It is over 40x faster than MPI on clustered system like the IBM SP, Beowolf, and others. The efficiency of shared memory programming in terms of simplicity and latency make it a highly desirable and productive programming environment. It should also be noted that we observe large SMPs or ccNUMA systems provide the best platform for maintaining the highest level of algorithmic efficiency and parallel efficiency in many of our applications. For a good discussion of this see “How Moderate-Sized RISC Based SMPs can Outperform Much Larger Distributed Memory MPPs”, Pressel, Sturek, Sahu, and Heavy [1].

3 ccNUMA Memory Management

3.1 Parallel Programming Issues on ccNUMA

Any program, parallel or not, requires fast access, in terms of latency and bandwidth, to its local data, memory that is used primarily by the local execution thread. How sensitive a program will be to the access performance is also dependent on the program’s use of memory latency-hiding hardware assistance and the quality and capability of such hardware. Clearly, the more effective the program’s utilization of those latency-hiding techniques is, the better the program will perform if sufficient bandwidth is available.

CcNUMA systems are very good at hiding the details of which processor the execution thread is running on and exactly where the memory is that the thread requires. They greatly simplify applications in that much code is typically devoted to I/O processing or other tasks that take an insignificant amount of time. Nonetheless, the major computational kernel of highly parallel applications with hundreds of execution threads and thousand of memory pages, can very quickly create contention, introduce excess latency, and reduce effective memory bandwidth when thread and memory locality is not properly managed. Each of these factors typically have a major impact on scaling and overall performance.

Given that the current ccNUMA architecture from SGI supports two processors per memory unit (node), any data that is shared between more than two threads will experience at least half the smallest inter-memory latency possible. As the number of threads using a given piece of data grows, so will the minimum average latency. Combined with the previous argument, it becomes clear that (1) a program must consider the data it is sharing between threads carefully, and (2) such data should be laid out in the available memory optimally.

Therefore, data decomposition for arrays that are accessed heavily and in parallel by many processors must be carefully placed in the memory system as carefully as one would do in an explicit messaging code. Further, the CPU executing the thread should have the most direct access possible to the memory containing

the local data. Lacking this co-location will not prevent proper execution of the program on the ccNUMA architecture, but it will result in poor performance and poor scaling under most circumstances. In the case of repeated runs of an application, random placement of execution threads and memory results in large variations in application run time as large as 2-3x.

Optimally, frequently accessed arrays contain strictly thread local data, data that is not shared with any other thread. Conversely, programs will scale better if the globally shared data arrays are the least frequently accessed, assuming the parallelization is not limited by algorithmic reasons. Put another way, the most frequently accessed data should be accessible with the least latency. Careful control of this characteristic will result in improved performance.

Much of the work performed over the last two years on the large Origin systems is in ensuring the accurate and deterministic placement of data arrays, ensuring that threads don't migrate from processor to processor, and partitioning the machine in such a way as to prohibit one job interfering with other jobs running simultaneously on the system.

3.2 Problems associated with memory layout

3.2.1 Program Internal Causes

No operating system can anticipate the memory layout needs of every program; in fact it cannot really anticipate the needs of any program. The operating system in use on SGI ccNUMA systems (IRIX) is virtual memory based (VM). As a result, physical memory is committed only to the program's memory space when the program first uses it. A key component of a good memory domain distribution is largely a function of the order in which the program "touches" its memory unless other means, external or internal, are used to control the layout. External means are usually fairly coarse in their nature and do not permit the degree of control that even simple ordering of memory "touching" will produce.

3.2.2 Program External Causes

In order for a program to place its memory domains according to the most optimal method successfully, the memory resources that the program eventually requests must in fact be available. In a shared system, this may not be the case. Even in a controlled environment that attempts to prevent oversubscription (job management system), other programs may miscalculate resource requirements and occupy memory resources that the program anticipated as being available.

Another factor that may break the optimal memory layout is "thread migration". The SGI ccNUMA system can reschedule executing threads on other CPUs, which are not local to the current memory. The operating system attempts to prevent this from occurring, but is not always successful during phases of local or global CPU over subscription.

3.2.3 Resource Allocation and Locality

It is desirable to have the access domains and execution threads placed on the system in such a way that the distance to each domain is minimized. This will guarantee the best possible latency for accessing each domain. In most cases neither the user nor the operating system will attempt to control this placement, usually resulting in a sub-optimal arrangement. The optimal scenario requires that the application specify its optimal thread and memory domain placement to the job management system, the job management system then guarantees these resources to the application. Then the operating system guarantees those resources to the job management system. When the application finally runs, the operating system provides support to the application in its desired thread and memory access domain placement.

3.2.4 Solutions for Locality Problems

Provided that a program performs the first touch of it's memory in a way that insures execution threads are local to the data they will access, an optimum memory layout will result if the operating system can guarantee the availability of the resources provided and guarantee that the memory resource is committed where it was first touched.

In the earlier IRIX releases, design issues and errors in the operating system components that select memory for the pages that a program touches and schedules threads to processors caused inefficiencies and sub optimal thread and memory placement. The operating system would not select the nearest memory in case the local memory overflowed. This resulted in degraded performance and interference with other programs running on the same system because the memory on nodes ostensibly assigned to those other programs would be erroneously depleted. The operating system would also frequently schedule the parent thread, not the newly created child thread, to a different processor that was not local to the current memory during thread creation. Threads also did not effectively migrate back to the processor closest to the memory. These issues created significant performance issues and resulted in significant run-time variations.

Several enhancements first developed at the NAS Facility have been incorporated into the main IRIX operating system release. These features attempt to guarantee consistent memory and thread placement and to prevent undesired memory allocation/stealing from occurring. First, a new feature that allows the job management system to provide a set of resources and communicate those resources to the operating system has been added. The operating system will then guarantee these resources to the job that is then attached to those resources. This is effective for both CPU and memory. The feature is called cpusets and memory limited cpusets. Put another way, a group of CPUs (say 20) can be allocated to Job. Those 20 CPUs and their associated memories are walled off from other Jobs running in the system. Should the job try to allocate pages beyond the memory capacity in those 20 CPUs, it will receive an error from the operating system and fail in the memory limited cpuset. Further, the job can then manage which threads run on which CPUs in a controlled way.

Other modifications force the operating system to be much more thorough in attempting to commit a page where it was touched. Only if it becomes impossible to do so will the page be committed somewhere else. In that case, the operating system will attempt to locate the nearest possible source of available pages to satisfy the request.

3.2.5 Improving Locality Issues

Once the granted resource are indeed guaranteed, several tools exist in the operating system to control the data access domain layout more carefully for each individual program. These tools exist as an extension in the page allocator component of the operating system IRIX and are known as Memory Locality Domains (MLDs) and Policy Modules (PMOs). Memory locality domains permit the user to create a blocks of memory that can be placed manually (specifying the physical target locations) or automatically, minimizing the distances in latency space between them for example. The Policy Modules then permit the user to attach data regions within the program to these memory blocks in several ways. The memory for a data region will be allocated out of the specified memory blocks (MLDs) per the policy established, until the memory in those memory blocks is exhausted. Several types of policies exist, the most notable being the “first touch” and “round robin” policies. The “first touch” policy will allocate memory from the nearest memory block specified via the MLD. Round robin will distribute subsequent allocations in time across the memory blocks in a round robin fashion.

Lastly, the user may also lock each thread to a specific CPU. This will prevent any type of thread migration from occurring.

These tools when used properly give the user very fine grain control over the exact placement of the access domains of the program. These tools should be used to achieve the maximum possible performance. The MLP Library routines described later in section 5.3.4 make use of these features and can effectively abstract this interface back up to the FORTRAN level.

The detailed programmatic interfaces are described in the man page mmci(5). Some of the functionality is also available via dplace without modifying the program itself. The man page dplace(1) describes the functionality of this tool.

The MLDs and PMOs are strictly limited by the cpuset mechanism described earlier when selecting physical memories (and processors). If logical placement is used, the selected memories will be confined to the memory limited cpuset. If physical placement is used and is outside of the bounds defined by the memory limited cpuset, an error will be returned.

3.2.6 Role of Job Management in Locality Issues

For optimal performance, the job management system (at NAS we use the Portable Batch System – PBS) must have an awareness of the physical topology of the machine. Due to the way the system discovers its configuration at boot time, the logical numbering of the CPUs from 1 to N does not imply that CPU #15 is physically close to CPU #16. This physical topology must be discovered from the router graph topology in /hw using a system call. Then, when a job specifies the number of processes and the amount of memory it requires, PBS determines the minimum number of nodes (a node is 2 processors and 600mb of memory) required to satisfy both the processor and memory requirement. PBS then tries to allocate physically close groups of processors as a memory limited CPUSET. This CPUSET is then passed on the job where is become the responsibility to the job the deal with its memory locality issues and the operating system to enforce resource allocation and ensure proper and deterministic behavior of memory allocation and thread placement.

Clearly, the functionality of the Job management system could be extended to deal with such resource allocation issues in a more dynamic way, or to better discover a jobs actual resource requirements, but is a problem of some complexity and in need of more research.

3.3 Other Enhancements

As most virtual memory based system do, IRIX will share text (executable code) between different programs executing. This becomes a problem for system libraries, which can be quite large. The first program to use a

specific library will cause that library to be copied (at least partially) into memories assigned to that program. Although memory limited cpusets will in-fact remove that text again if memory resources become exhausted in the granted set of resources, this consumes system resources in terms of disk I/O and processor cycles.

A solution locally developed at the NAS Facility is to preload the most commonly used library texts into a memory area set aside for this purpose. This will prevent the text from being copied into an area of memory that is part of a set of resources granted to a job. The overhead of removing and recopying the library text is thus avoided.

4 Reliability

Reliability of these large shared memory systems is a key concern. As in any complex system, at some point, there become too may parts to reliably operate the system for any reasonable length of time. Our target MTTI for the 512-processor system is 7 days. Historically, production supercomputers have at NAS have exhibited MTTI on the order of 14 days (figure 1).

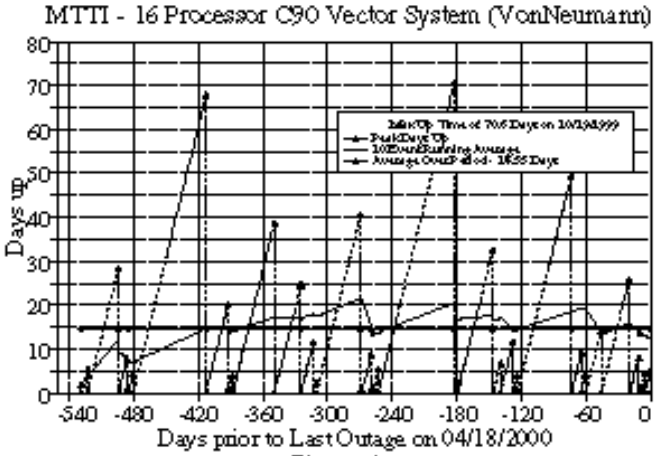


Figure 1

Several systems were analyzed over the past 1 to 2 years (depending on the system) for MTTI (mean time to interrupt) and GA (gross availability) and then compared to the 512p Lomax system.. The systems are continuously monitored 24 hours/day, 365 days/year for operational status. Detailed logging information is collected on all outages with an effort made to identify the source of the outage and log it. The duration of the outage is tracked and updated. Using the data collected, we define MTTI to include only unscheduled interrupts caused by:

- hardware failure
- software failure, hang or panic
- unknown cause

We do not count unscheduled interrupts caused by:

- tape device failures or hangs
- NFS failures or hangs

Further we do not include scheduled interrupts such as

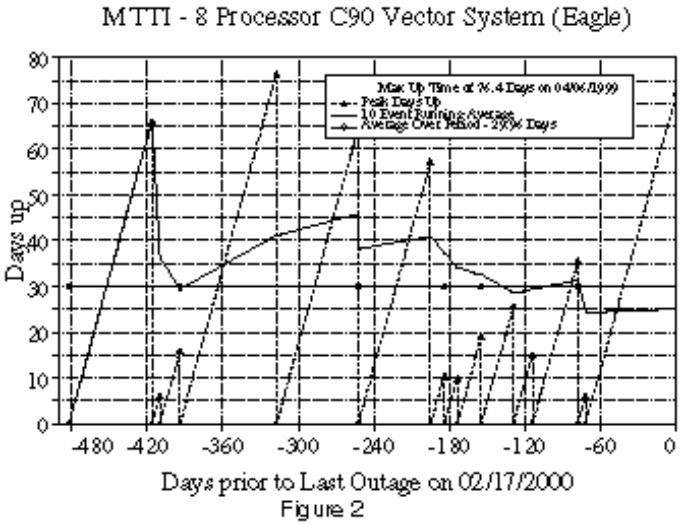


Figure 2

those for:

- dedicated time
- preventative maintenance

This particular treatment of the data provides relatively

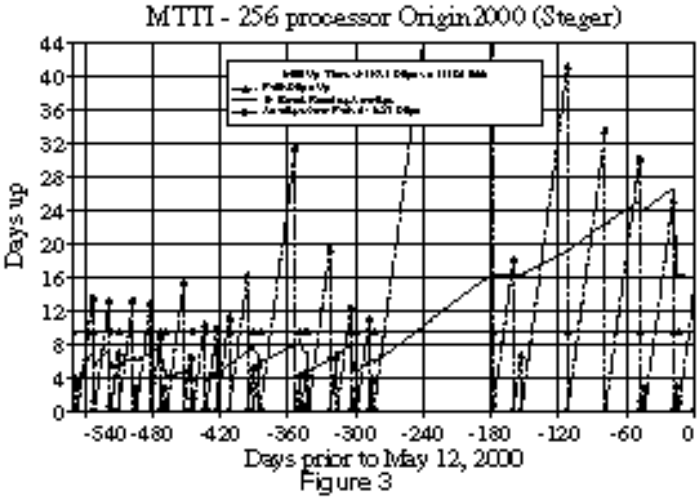


Figure 3

even ground for comparing several different machines, both parallel and vector, that are in production or research use at NAS.

First off we have the traditional Cray C90 vector systems (figures 1 and 2). These systems are viewed as reliable workhorse machines with acceptable MTTI and uptime characteristics.

Shown in the X-axis is the number of days prior to the most recent outage. In figure 1 we see that the last recorded outage occurred on 4/18/00 with data extending back over the previous 540 days. Shown in the Y-axis is the number of days the system stays up

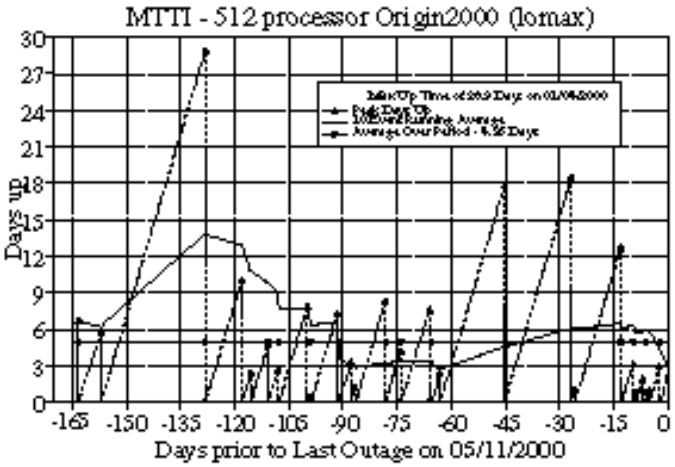


Figure 4

without an unscheduled interrupt with the up triangles marking the peak number of days up. The x-axis below the up-triangle marks the date of the outage. Also shown

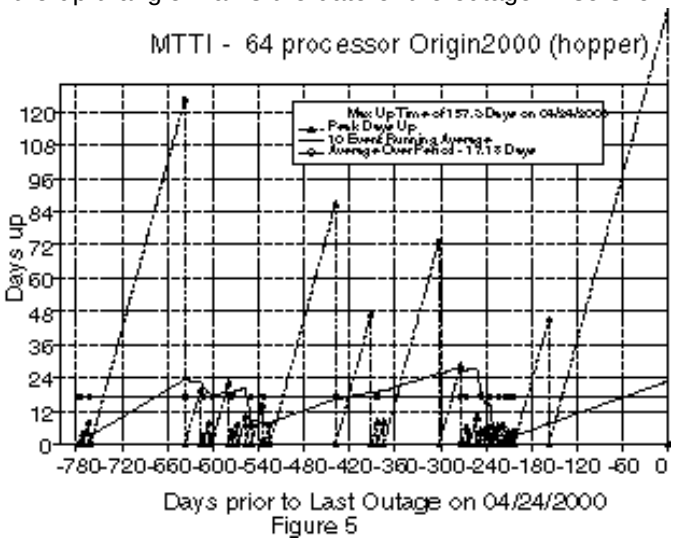


Figure 5

is the “mean time to interrupt” or straight average for the entire time period plotted. In figure 1, the average over the past 540 days was 14.5 days between unscheduled interrupts. Also included is the 10 event running window average, averaging the previous 5 and next 5 interrupts. The MTTI is also noted in the legend (14.55 for figure 1) along with the maximum number of days without interrupt (70.6 days for figure 1).

We see that the reliability data is directly correlated via the complexity of the two machines. Vn has twice the number of processors, roughly twice the number of I/O components and 4 times the memory as compared to Eagle, yet roughly ½ the MTTI (14.5 vs. 29.9 days). As expected, this strong correlation holds for all systems that were analyzed.

Figures 3 and 4 show a 256p Origin2000 system (Steger) and a 512p Origin2000 system (Lomax).

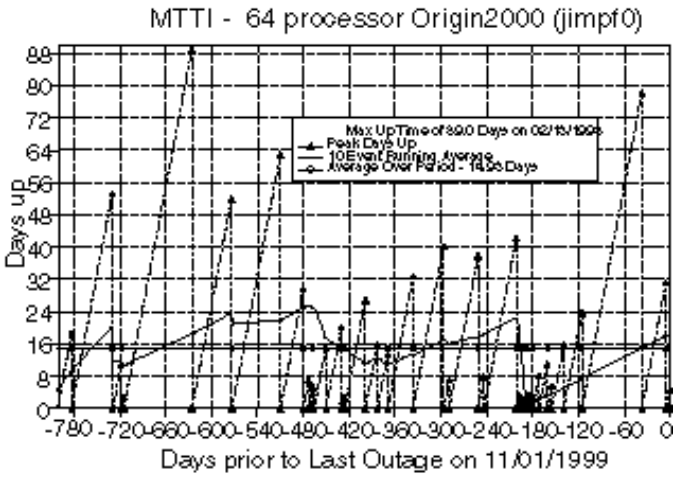


Figure 7

Again we show good correlation between part count and MTTI, with the 256p at 9.27 and the 512 at 4.9. However it is meaningful to point out the environments in which these machines operate and that the role operating system stability has a major impact. From approximately 9.5 months ago (-285 days on figure 3), the 256p origin2000 Steger was transitioned from a “research and development” platform to a “production” resource. The approach to managing the system changes from aggressive integration and testing of new software, to

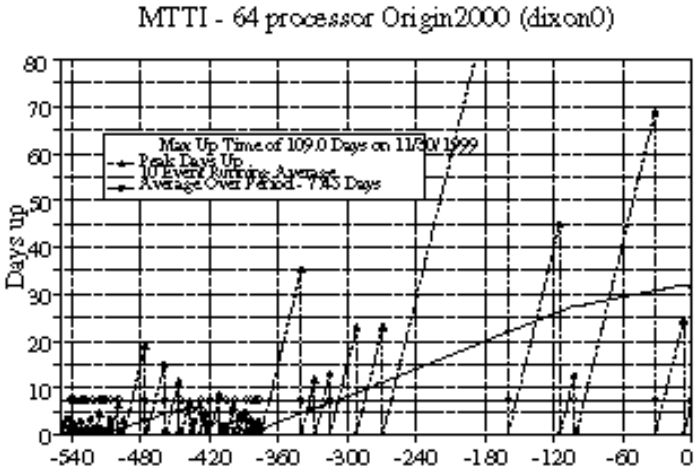


Figure 8

conservative integration of feature and function that are well tested in dedicated time and only put into the production environment when know to work well. As shown the running average MTTI moves into the 20+ day range with the MTTI over the period besting that of the 16 processor C90 (table 2) at 21.7 days vs. 14.9 days. Additionally, as problems with the operating

system are located and fixed a substantial reduction in software interruptm rate is realized.

Figure 5 show MTTI data for a 64p Origin2000 (Hopper). system. This figure shows repeated outages in the -240 day range, with many failures occurring within a relatively short period of time. MTTI seems to still correlate well with figures 3 and 4 hopefully sustaining their upward trend. Figures 6, 7 and 8 are 3 additional

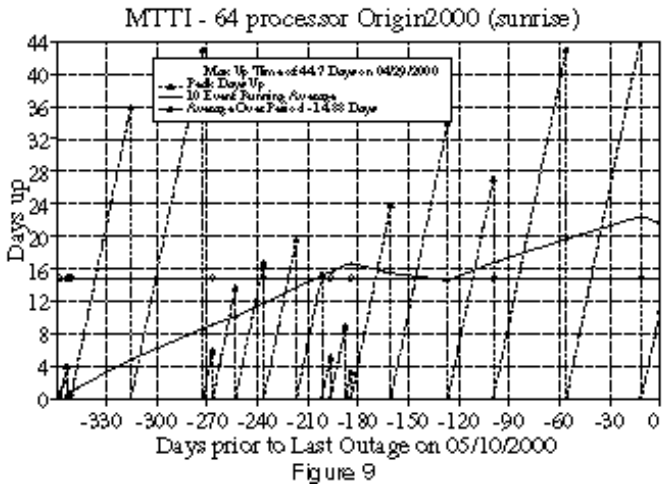


Figure 9

64p Origin2000 systems that also shown as additional background data points.

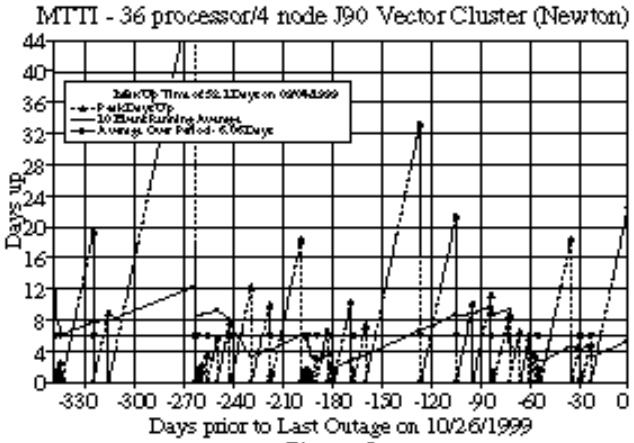


Figure 6

For completeness, Figure 6 shows MTTI data for J90 classic processor systems. The J90 cluster (Newton) consisted of 4 system, clustered together as a 16 processor system, a 12 processor system, and 2- 4 processor systems for a total of 36 processors.. Data for this cluster was not broken out by individual system, however, with failure rates being driven by complexity/part count, the data should give some indication of reliability. The data show a surprisingly low MTTI with a relatively high number of hardware failures (table 1).

Table 1 provides detailed data on several outage categories. One needs to be a little careful in a strict interpretation of the data as presented. Often it is difficult to determine the exact cause of a system crash, whether unwarranted software panic, or one the result of a failure in hardware. An effort is made to determine and log the cause after the fact and as such the data should provide good trend information and an accurate picture of combined hardware/software MTTI. Table data in presented as follows:

- Software Outage - Unscheduled outages typically caused by operating system panics, or hangs where the system is partially (or not) usabel and must be re-booted to clear the problem.

- Hardware Outage – Unscheduled outages where a hardware cause can be traced to.

- Other Caused Outage – Other cause that may or may not be system related, such as an analyst causing a reboot by accident.

- Tape Caused Outage – On VN in particular, there are D3 tape drives that fail on a daily basis. To clear these failures and get the tape system back to operational, the system may need to be rebooted. For an even comparison we choose to exclude these from the general hardware/software MTTI.

- NFS Caused Outage – The Origin systems are operated in a cluster environment. Frequently, NFS3 will hang and tracking down the source (whether client. Server, or network inbetween) is problematic as best.

- Network Caused Outage - Usually resulting from a problem in the network interface that requires replacement or reboot to clear.

- Unsched Facility Shutdown - Outages related to power failures , earthquake, cooling or other facility related problems.

- Sched Dedicated Time – Time set aside in advance for analysts to perform such tasks as software development and testing, performance testing, or application development.

- Sched Preventative Maint – Time set aside in advance for hardware maintenance of the system.

- Sched Facility Shutdown – Time set aside in advance for a facility related outage.

- System up with problem – Time a system is up and generally usable, but with a problem that is effecting some fraction of users. If the system is subsequently rebooted because the problem worsens or is

necessary to clear the problem, an outage will be indicated the appropriate category above.

- System up without problem – Time the system is up an available to run user workloads.

- Total of All Outages – Aggregates all outages over the given time period, including both unscheduled and scheduled. This probably the most relavent to the general user community as they often do not rally care why their tool is not available, just the fact that is is not.

- Total All Unsched Outages - Aggregates all unscheduled outages over the given time period. Indicates those failures that are out of the control of the analysts.

- Total of HW/SW Outages – Aggregates hardware and software outages for the purpose of comparing different systems in this paper.

Table 1 data has been collected over the past years as available and should be contrasted against that in table 2 which provides the same analysis, but only from 8/99 forward. Several conclusions are made from the provided data:

- The data show that the 256p processor Origin2000 system is more reliable than the 16p C90 system in MTTI and gross availability. The represents the first system at NAS that has outperformed a C90 in reliability, cost, performance, and price performance.

- IRIX operating stability has improved substantially over the past 3 years on all systems. IRIX reliability was one the most significant concern NAS had in moving into the Origin2000 project. Experience with the IRIX Operating system on Power Challenge Arrays had shown unacceptable stability. It is quite an accomplishment that this reliability problem has been fixed.

- Origin2000 hardware reliability has improved substantially over the past 3 years. Systems still seem to go through fits of outages, where a given system will experience multiple outages in a short period of time before stabilizing. It is believed that better tools are necessary to monitor, track and diagnose system problems as they occur. We are told that there is something in the works.

- IRIX operating stability on the 256p Origin2000 system has moved quickly from alpha-unreliable to production quality in 9 months. As with any first system, there are several problems, bugs, and features to work around. Good and steady progress was made as realized in MTTI better than a C90.

- Hardware MTTI for 512p Origin2000 SSI systems is within an acceptable range, software still needs to stabilize in order to reach desired overall MTTI. All indications are that MTTI will approach that of the C90.

- Hardware failures on the traditional vector systems take longer to bring back into service than do the O2k systems, largely due to O2k systems having the capability to be put back into service with some portion of the machine disabled and then repaired at a later (and scheduled) time

4.1 HPM Performance

At NAS we monitor HPM data for most machines that provide it. Figure 10 show a comparison between The 16p C90 VN and the 256p Origin2000 Steger. Data is missing in the latter half of 1999 for the 256p system. We see that the sustained workload performance of 256p significantly out performs the C916 by a factor of approximately 2.5x. This is significant in that this measures entire workload performance and not just a single benchmark code. We do not currently have HPM data for the 512p system due to it apparent corruption by unknown cause. The jump in performance of Steger between July 1999 and March 2000 is believed to be substantially a result of the operating system modifications to better control memory layout and thread affinity. The overall system utilization is in the 75-80% range making for substantial room for improvement in system management and adding a checkpoint/restart feature to the system.

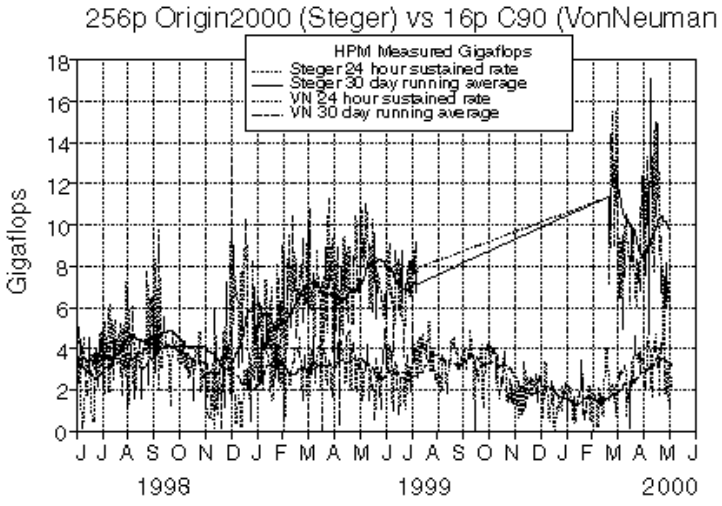


Figure 10

	512p Origin2000 (lomax)				256p Origin200 (steger)				64p Origin2000 (hopper)			
	Total #Days	. %Time	Evnt #	MTTI Days	Total #Days	. %Time	Evnt #	MTTI Days	Total #Days	. %Time	Evnt #	MTTI Days
Software Outage	1.28	0.8%	29	5.9	1.63	0.3%	33	17.1	0.59	0.1%	23	35.1
Hardware Outage	0.38	0.2%	6	28.3	5.94	1.1%	27	20.9	2.49	0.3%	23	35.1
Other Caused Outage	-	0.0%	1	170.0	0.38	0.1%	18	31.3	1.26	0.2%	28	28.8
Tape Caused Outage	-	0.0%	-	-	-	0.0%	-	-	-	0.0%	-	-
NFS Caused Outage	0.03	0.0%	1	170.0	0.13	0.0%	4	141.0	0.04	0.0%	3	269.1
Network Caused Outage	-	0.0%	-	-	-	0.0%	2	282.1	-	0.0%	1	807.2
Unsched Facility Shutdown	-	0.0%	-	-	1.47	0.3%	6	94.0	1.41	0.2%	5	161.4
Sched Dedicated Time	4.24	2.5%	34	5.0	46.38	8.2%	65	8.7	24.26	3.0%	101	8.0
Sched Preventative Maint	0.10	0.1%	1	170.0	1.21	0.2%	3	188.0	0.09	0.0%	2	403.6
Sched Facility Shutdown	3.64	2.1%	2	85.0	5.55	1.0%	4	141.0	5.89	0.7%	5	161.4
System up with problem	0.53	0.3%	15	11.3	0.61	0.1%	29	19.5	0.32	0.0%	21	38.4
System up without problem	159.79	94.0%	.	-	500.80	88.8%	.	-	770.83	95.5%	.	-
Total of All Outages	169.99	.	74	2.3	564.10	.	162	3.5	807.18	.	191	4.2
Total All Unsched Outages	169.99	.	37	4.6	564.10	.	90	6.3	807.18	.	83	9.7
Total of HW/SW Outages	169.99	.	35	**4.9**	564.10	.	60	**9.4**	807.18	.	46	**17.6**

	16p C90 Vector (vn)				8p C90 Vector (eagle)				36p J90 Vector Cluster (newton)			
	Total #Days	. %Time	Evnt #	MTTI Days	Total #Days	. %Time	Evnt #	MTTI Days	Total #Days	. %Time	Evnt #	MTTI Days
Software Outage	2.11	0.4%	14	42.1	0.42	0.1%	8	65.8	3.63	0.9%	40	9.7
Hardware Outage	4.76	0.8%	23	25.6	2.58	0.5%	9	58.5	2.95	0.8%	24	16.2
Other Caused Outage	0.20	0.0%	22	26.8	0.04	0.0%	2	263.2	0.12	0.0%	7	55.6
Tape Caused Outage	0.84	0.1%	18	32.7	0.05	0.0%	2	263.2	-	0.0%	-	-
NFS Caused Outage	-	0.0%	-	-	0.06	0.0%	2	263.2	0.04	0.0%	4	97.4
Network Caused Outage	0.06	0.0%	2	294.7	0.12	0.0%	2	263.2	0.02	0.0%	2	194.7
Unsched Facility Shutdown	3.35	0.6%	6	98.2	1.99	0.4%	7	75.2	0.28	0.1%	4	97.4
Sched Dedicated Time	3.52	0.6%	26	22.7	2.27	0.4%	13	40.5	2.05	0.5%	11	35.4
Sched Preventative Maint	1.87	0.3%	17	34.7	0.12	0.0%	2	263.2	0.14	0.0%	4	97.4
Sched Facility Shutdown	5.60	1.0%	5	117.9	6.18	1.2%	7	75.2	6.00	1.5%	8	48.7
System up with problem	25.55	4.3%	768	0.8	6.28	1.2%	198	2.7	0.89	0.2%	35	11.1
System up without problem	541.61	91.9%	.	-	506.36	96.2%	.	-	373.31	95.9%	.	-
Total of All Outages	589.47	.	133	4.4	526.47	.	54	9.8	389.43	.	104	3.7
Total All Unsched Outages	589.47	.	85	6.9	526.47	.	32	16.5	389.43	.	81	4.8
Total of HW/SW Outages	589.47	.	37	**15.9**	526.47	.	17	**31.0**	389.43	.	64	**6.1**

	64p Origin2000 (jimpf0)				64p Origin2000 (sunrise)				64p Origin2000 (kalnay)			
	Total #Days	. %Time	Evnt #	MTTI Days	Total #Days	. %Time	Evnt #	MTTI Days	Total #Days	. %Time	Evnt #	MTTI Days
Software Outage	0.52	0.1%	20	48.8	0.85	0.2%	16	23.3	0.05	0.0%	6	97.5
Hardware Outage	4.15	0.4%	35	27.9	0.74	0.2%	8	46.6	0.08	0.0%	4	146.2
Other Caused Outage	1.10	0.1%	4	244.1	0.57	0.2%	4	93.2	-	0.0%	-	-
Tape Caused Outage	-	0.0%	-	-	-	0.0%	-	-	-	0.0%	-	-
NFS Caused Outage	0.03	0.0%	2	488.3	-	0.0%	-	-	-	0.0%	-	-
Network Caused Outage	0.09	0.0%	5	195.3	-	0.0%	-	-	0.01	0.0%	1	584.9
Unsched Facility Shutdown	1.36	0.1%	6	162.8	0.82	0.2%	3	124.2	0.77	0.1%	3	195.0
Sched Dedicated Time	4.35	0.5%	31	31.5	1.16	0.3%	9	41.4	1.51	0.3%	12	48.7
Sched Preventative Maint	1.23	0.1%	17	57.4	0.20	0.1%	2	186.3	0.59	0.1%	6	97.5
Sched Facility Shutdown	10.25	1.1%	6	162.8	5.77	1.6%	5	74.5	5.30	0.9%	5	117.0
System up with problem	0.28	0.0%	13	75.1	0.11	0.0%	9	41.4	0.19	0.0%	6	97.5
System up without problem	953.16	97.6%	.	-	362.40	97.3%	-	-	576.36	98.6%	.	-
Total of All Outages	976.52	.	126	7.8	372.62	.	47	7.9	584.86	.	37	15.8
Total All Unsched Outages	976.52	.	72	13.6	372.62	.	146	12.0	584.86	.	14	41.8
Total of HW/SW Outages	976.52	.	55	**17.8**	372.62	.	134	**15.5**	584.86	.	10	**58.5**

Table 1 - Multi Year Data

	16p C90 Vector (vn)				256p Origin2000 (steger)				512p Origin200 (lomax)			
	Total #Days	. %Time	Evnt #	MTTI Days	Total #Days	. %Time	Evnt #	MTTI Days	Total #Days	. %Time	Evnt #	MTTI Days
Software Outage	1.08	0.4%	6	47.4	0.46	0.2%	5	56.6	1.28	0.8%	29	5.9
Hardware Outage	3.04	1.1%	13	21.9	2.10	0.7%	8	35.4	0.38	0.2%	6	28.3
Other Caused Outage	-	0.0%	10	28.5	0.07	0.0%	1	282.9	-	0.0%	1	170.0
Tape Caused Outage	0.35	0.1%	7	40.6	-	0.0%	-	-	-	0.0%	-	-
NFS Caused Outage	-	0.0%	-	-	-	0.0%	-	-	0.03	0.0%	1	170.0
Network Caused Outage	0.06	0.0%	2	142.3	-	0.0%	1	282.9	-	0.0%	-	-
Unsched Facility Shutdown	2.22	0.8%	4	71.1	0.91	0.3%	4	70.7	-	0.0%	-	-
Sched Dedicated Time	2.06	0.7%	14	20.3	1.39	0.5%	17	16.6	4.24	2.5%	34	5.0
Sched Preventative Maint	1.85	0.7%	15	19.0	-	0.0%	1	282.9	0.10	0.1%	1	170.0
Sched Facility Shutdown	4.46	1.6%	4	71.1	3.97	1.4%	3	94.3	3.64	2.1%	2	85.0
System up with problem	18.75	6.6%	604	0.5	0.57	0.2%	24	11.8	0.53	0.3%	15	11.3
System up without problem	250.67	88.1%	.	-	273.38	96.7%	.	-	159.79	94.0%	.	-
Total of All Outages	284.54	.	75	3.8	282.85	.	40	7.1	169.99	.	74	2.3
Total All Unsched Outages	284.54	.	42	6.8	282.85	.	19	14.9	169.99	.	37	4.6
Total of HW/SW Outages	284.54	.	19	**15.0**	282.85	.	13	**21.8**	169.99	.	35	**4.9**

	64p Origin2000 (jimpf0)				64p Origin2000 (dixon0)				64p Origin2000 (kalnay)			
	Total #Days	. %Time	Evnt #	MTTI Days	Total #Days	. %Time	Evnt #	MTTI Days	Total #Days	. %Time	Evnt #	MTTI Days
Software Outage	-	0.0%	-	-	0.07	0.0%	5	58.5	-	0.0%	2	163.4
Hardware Outage	2.02	0.8%	3	79.8	0.78	0.3%	5	58.5	0.02	0.0%	1	326.9
Other Caused Outage	0.49	0.2%	1	239.5	-	0.0%	1	292.4	-	0.0%	-	-
Tape Caused Outage	-	0.0%	-	-	-	0.0%	-	-	-	0.0%	-	-
NFS Caused Outage	-	0.0%	-	-	-	0.0%	-	-	-	0.0%	-	-
Network Caused Outage	-	0.0%	1	239.5	-	0.0%	-	-	0.01	0.0%	1	326.9
Unsched Facility Shutdown	0.75	0.3%	3	79.8	0.25	0.1%	2	146.2	0.77	0.2%	3	109.0
Sched Dedicated Time	0.46	0.2%	5	47.9	0.67	0.2%	6	48.7	0.84	0.3%	5	65.4
Sched Preventative Maint	-	0.0%	-	-	-	0.0%	-	-	0.17	0.1%	1	326.9
Sched Facility Shutdown	3.40	1.4%	3	79.8	3.44	1.2%	3	97.5	4.16	1.3%	4	81.7
System up with problem	0.25	0.1%	8	29.9	2.88	1.0%	19	15.4	0.01	0.0%	3	109.0
System up without problem	232.11	96.9%	.	-	284.35	97.2%	.	-	320.91	98.2%	.	-
Total of All Outages	239.48	.	16	15.0	292.44	.	22	13.3	326.89	.	17	19.2
Total All Unsched Outages	239.48	.	8	29.9	292.44	.	13	22.5	326.89	.	7	46.7
Total of HW/SW Outages	239.48	.	3	**79.8**	292.44	.	10	**29.2**	326.89	.	3	**109.0**

Table 2 - Data From 8/99 Forward

5 Parallel Programming 512 Processors

M As discussed earlier, non-uniform memory presents a challenge to the application programmer. Naïve use of simple loop level parallelism via OpenMP is not likely to scale beyond a few 10s of processors. Although we have seen good scaling to 128 processors on a very special case large single zone CFD grid, not all zones in CFD problems are that large, in fact they typically vary greatly in size from a few thousand points to millions within a single complex vehicle geometry. The individual zones also vary in complexity, flow characteristics and thus convergence rates. Not only does this pose initial static load balancing issues, dynamically throughout the computation effective load balancing can reduce the time to solution. To deal with architectural and application issues, NAS developed a new methodology for achieving very high levels of parallel efficiency on very large ccNUMA shared memory systems. This methodology is simple, general, and widely applicable to real-world production application codes in use at NASA and elsewhere. The new methodology is formally called shared memory Multi-Level Parallelism (MLP).

5.1 Multi-Level Parallelism (MLP)

The implementation is based on shared memory access to global data while invoking two levels of parallelism for scaling efficiency. This new and very simple parallel programming approach was first developed in FY 1998.[3] MLP is a vastly simplified, and inherently more scalable

alternative to MPI. During the past year, this technique has been refined and improved. Executions of the production OVERFLOW CFD on the 512p system have demonstrated over 60 GFLOP/s of sustained performance for customer driven real world problems.

The MLP technique was developed under the Origin 2000 Optimization Effort that began in September, 1997. This effort was focused on demonstrating that systems based on RISC microprocessors could execute production CFD code at rates comparable to, or in excess of, a dedicated 16 CPU Cray C90 system. NASA was concerned that there was no apparent graceful migration path from the aging C90 systems that supported the bulk of their production computing. Previous experiments with microprocessor based systems had met with disappointing results. The Origin system was a novel new approach, and had some support for shared memory parallelism not previously available.

5.2 Overflow

The OVERFLOW CFD code is extensively used in the government and commercial aerospace communities to evaluate new aircraft designs. It is one of the largest consumers of NASA supercomputing cycles, and large simulations of highly resolved full aircraft are routinely undertaken. Typical large problems might require hundreds of Cray C90 CPU hours to complete. OVERFLOW was considered to be “a toughest case”. If OVERFLOW could be successfully converted, then conversion of virtually all multi-zonal production CFD codes at NASA would likely be successful.

The code consists of over 100,000 lines of FORTRAN and almost 1000 subroutines. It has an extensive selection of user inputs that allow various solvers, smoothers, turbulence models, etc. to be selected. It was essential for this effort that no functionality be lost and performance meet or exceed C90 capabilities. The details of the OVERFLOW optimization effort and the MLP parallelization approach are given in the sections below. We start with a general discussion on parallelization.

5.3 The Focus is on Parallelism

Simply put, parallelism is the central issue in achieving exceptional performance on large-scale applications. No matter what system is chosen, parallelism at some level will be an issue. For commodity based microprocessor system, it has been repeatedly shown that such CPUs routinely achieve only 10-20% of their peak performance on real scientific codes. This amounts to perhaps 100-200 MFLOP/s per CPU. On systems where commodity microprocessors are at the core of their design, an application must scale to hundreds of CPUs on such systems, if NASA is to achieve sustained performance levels in the 50-l00 GFLOP/s range necessary to stay competitive in high end computing today.

Parallelism is being aggressively pursued on two fronts. The first is a continuation of the classic message passing approach of clustered computing. The current leading technology for this approach is via the Message Passing Interface, commonly referred to as MPI[4]. The second philosophy simply expands on the original Cray Research Incorporated compiler directive approach. This has been recently codified into an industry standard set of directives under the banner OpenMP[5]. OpenMP is a widely supported open standard on most if not nearly all shared memory systems. Each parallelization approach has its advantages and disadvantages. We discuss each of these in turn, along with MLP, which appears to take the best from both.

5.3.1 Message Passing Interface (MPI) Parallelism

The MPI message passing approach is an outgrowth of the PVM message passing approach of the 80s[6]. It provides the user with the ability to spawn a series of identical processes, each of which is intended to do a fraction of the total work at hand. Each of the independent processes exchange information as necessary via packetized “messages”. These messages traverse the available hardware communication paths between processors, and are subject to the path’s inherent limitations. A major issue with the MPI approach is that it requires that the user perform all parallel decomposition of the problem at hand. In addition the user must perform all data packetization, communication, and process synchronization manually via extensive changes to his code, adding calls to the MPI library to do the functions required.

MPI has some advantage in that it is an industry recognized standard and allows code executions across clusters of machines. As of today, it also enjoys a large base of user-converted code. MPI however, suffers a number of serious drawbacks:

1) MPI is an arcane and complex library of about 100 subroutines, often with complex and non-intuitive arguments. A quote from the MPI documentation drives this home “As discussed in the previous subsection, the use of vanilla send and receive routines need careful consideration. Otherwise a program may easily end up deadlocking. Even more, some communication constructs may go well for some data, but deadlock with others. So, at first sight, this just looks like the very opposite of ease of use”.

2) Calls to MPI routines interspersed in the code often make it difficult to follow the original code logic, as synchronization points become muddled.

3) The process of debugging, profiling, and maintaining MPI based codes is well known to be a very difficult, error prone and labor-intensive task.

4) Very few tools for efficiently developing code based on the MPI programming model exist, even after almost two decades of work in this area.

5) Codes frequently grow substantially in size and complexity as calls to MPI routines are added (the MPI version of OVERFLOW is 10,000 lines larger than the vector MLP version).

6) Large latencies, particularly when accessing other systems in a cluster environment, often dramatically reduce overall scaling performance in real world applications. CFD in particular, is highly sensitive to message latencies and their effects on code scaling.

7) Many MPI versions of production codes have been forced to give up desirable physics, algorithmic efficiency (i.e. implicitization) and robustness in order to make the code more latency tolerant and scalable.

All of these issues together argue strongly against the use of MPI in any application except where absolutely necessary.

5.3.2 Loop Level Parallelism (OpenMP)

OpenMP is a vastly simpler method of parallelizing code. This approach allows users to explicitly request parallelism at the loop level by inserting compiler directives in front of each loop targeted for parallelization. Parallel sections are executed via the operating system's automatic spawning of lightweight threads, each of which performs a fraction of the work of the loop in parallel. The common implementation is to have each thread execute a subset of the total iteration count of the loop, with the subsets distributed automatically as evenly as possible across all CPUs in the system. The OpenMP approach avoids the need for the user to construct messages or explicitly synchronize processes, dramatically simplifying the code and shortening development time. Many production codes already contain Cray microtasking directives, and it is a simple matter to convert these directives to the OpenMP equivalents.

OpenMP requires no messaging as the system works entirely within a shared memory environment. Any exchange of data between any of the parallel threads is accomplished by memory referencing instructions only. Thus, communication time is on the order of hundreds of nanoseconds, not ten (in box) or even 50 (across boxes) microseconds typical of the best of MPI implementations. Also, because the user's code is executed as a single "a.out" all of the standard debugging and profiling tools can be used during the code development process. OpenMP is a recent development. It is by far the simplest method for developing parallel code. Unfortunately, it doesn't always scale as well as expected.

The OpenMP scaling issues are for the most part a direct consequence of executing an OpenMP application on a NUMA based memory architecture. That is, it is an effect intimately tied to the variability in the time needed to access a particular memory location from a particular CPU. If the user does not exercise extreme care in the layout of data in a NUMA system, overall scaling can be quickly inhibited. It is not uncommon for NASA application codes to fail to scale past 8 CPUs when ported from the Cray C90 systems; particularly when the porting effort only consists of naively converting Cray directives to OpenMP directives.

For the highest performance, OpenMP requires the developer to generate code that can be parallelized to hundreds of CPUs on a loop by loop basis. On NUMA architectures, this can be a difficult process. It is particularly difficult for large pre-existing production codes that have extensive data structures that are difficult to modify. In fact, OpenMP is generally only successful at very large CPU counts when codes have been completely rewritten with OpenMP in mind. This is a time consuming process at best, and is about as difficult as converting to MPI.

5.3.3 MLP Parallelism

MLP is differentiated from MPI in that all communication between processes is through native shared memory referencing instructions, not messages. It is different from OpenMP in that it still supports spawning of independent processes much like MPI, and uses those processes to invoke a second level of job parallelism. Because the technique relies on shared memory communication for all processes, latencies continue to be on the order of hundreds of nanoseconds, not tens of microseconds for all processes in the system. This permits very high levels of scaling efficiency for the largest CPU counts.

At NASA Ames MLP has been implemented in a user callable library, MLPlib. MLPlib consists of a total of three subroutines (totaling 150 lines of source code). This contrasts sharply with the highly complex MPI

approach with tens of thousands of lines of code comprising the MPI library routines.

MLPlib provides all of the functionality needed to support very high parallel scaling efficiencies to 512 CPUs and beyond once those platforms become available.

MLP programming is an open system design and has the following attributes:

- Two levels of parallelism
- Coarse grained parallelism provided by UNIX forked processes
- Fine grained parallelism provided at the loop level via OpenMP directives
- No explicit messaging – all communication is through globally shared data arrays that are memory mapped into the processes address space.

The typical MLP application will have the coarse grained decomposition of the typical MPI code, but will do so with the much simpler forking of processes under user control. From a user perspective he simply types "a.out". The code itself at some point will make the MLPlib call to fork other processes. There is no "mpirun" command. The forked processes provide the coarsest level of parallelism. For multi-zonal CFD codes the coarse grained processes are usually assigned to work on one or more 3D zones for the duration of the calculation.

As the calculation proceeds each of the MLP processes serially proceeds through the all of the zones assigned to it. Generally, this means taking a time step for a zone and proceeding to the next zone in the list. Each of the MLP processes can take advantage of loop level parallelism using OpenMP directives. This greatly expands the number of CPUs available to work on the total problem, yet does not require fine grained loop level parallelism to span more than a handful of CPUs. This is highly desirous in order to keep overall parallel scaling efficiency high.

It should be noted that within MLP, CPUs can be added or dropped dynamically within microseconds to effect better load balance as the problem proceeds. This is possible because explicit movement of data is unnecessary in the MLP shared memory environment. Dynamic load balancing is a well known difficult problem within the MPI or pure OpenMP environments. For MPI much data must be moved. Within OpenMP it is difficult to design a code to operate on different zones with differing and changing CPU counts at the same time. For MLP, dynamic load balancing becomes almost a trivial operation. One simply changes the values in an array holding the CPU counts per process.

As mentioned earlier MLP establishes a global shared memory arena to allow users to communicate information between processes. This arena can be considered to be a global "common" block from the FORTRAN programmer's view. All variables assigned to this arena are visible to all processes executing. Furthermore, any accesses of these variables by any process are performed by direct memory reference instructions identical to the normal access of any other variable. All synchronization of access to the data between processes is handled by the very fast hardware cache coherency mechanism. Thus, when any process writes data items in this area, all other processes are guaranteed to have rapid access to the latest value.

5.3.4 MLPlib – Building an MLP application

MLPlib was created at NASA Ames to allow the simple development of MLP applications. The library is consists of a total of three routines requiring approximately 150 lines of source code. The library requires only the most primitive of UNIX calls. As a result, it is UNIX platform independent. It currently resides on Sun and SGI systems at NASA Ames. The routines are described below:

Subroutine GETMEM(xarry,xpoint,numbyt)

The GETMEM routine is called for each variable the user declares to be in the "global" common that is visible to all processes. In this case the GETMEM routine allocates numbyt bytes to the array, xarray. The variable xarray will be visible to all MLP processes, and will be pointed to by the Cray pointer, xpoint. From the user's standpoint the programming model is truly that of a global common block that contains all shared variables.

MLPlib utilizes standard UNIX mmap calls to create the globally shared memory arena. The philosophy behind MLP is that each a.out will write data and read data from the arena whenever it needs to communicate information to another process. This occurs at full memory speeds just like any other memory access. There are no messages, since all transactions are memory referencing instructions. Latencies on Origin 2000 systems are on the order of 0.5 microseconds for this activity, not the 10 (in box) or 50 (across box) microseconds for MPI messaging.

Subroutine FORKIT(numpro,nowpro)

The FORKIT routine spawns additional MLP processes. A total of numpro processes are created. Nowpro is returned, and is the current process number. Under MLP, the user spawns a number of a.out's that occasionally share data via the global shared data

created with GETMEM. Otherwise they are independent and execute autonomously.

Subroutine BARRIER(numpro)

The BARRIER routine performs barrier synchronization between numpro MLP processes. BARRIER waits until numpro processes have arrived at the barrier, then all drop through.

Even though MLPlib is a very simple library, results provided later in this paper demonstrate its power to scale to large CPU counts. This library should be contrasted with the MPI library, which consists of around 100 routines and tens of thousands of lines of code containing many complicated UNIX system calls. Because of the reduced dependency on system calls, MLP is much more reliable than MPI. This is particularly true for applications executing on hundreds of CPUs.

5.3.5 MLP and Multi-Zonal CFD

NASA's multi-zonal CFD codes like OVERFLOW, CFL3D, TLNS3D, and LAURA to name a few, are ideal candidates for MLP parallelism. These codes represent a major fraction of the compute cycles expended on NASA's large compute servers on a yearly basis. They all decompose a large region of interest into many linked smaller 3D regions. These smaller regions can be solved mostly in parallel, with the occasional exchange of boundary information at the end of a time step. In short, the recipe for converting a multi-zonal CFD code to MLP is:

- Assign global variables to shared memory arena (i.e. boundry condition data)
- Spawn MLP parallel processes
- Assign groups of 3D zones to each MLP process (coarse parallelism)
- Assign groups of CPUs to each MLP process (fine parallelism)
- Start the time stepping
- Synchronize MLP processes at the end of a time step
- Update global data if necessary
- Perform I/O as needed
- Continue the time stepping

Basically, the converted MLP code is initialized via the master process which performs all initialization tasks. This process then allocates all variables that are to be global via GETMEM and then touches the global data arrays in an "optimal way" which may be as simple as round-robin across all processor memories in the run or whatever has been determine as optimal. The master process then spawns a number of other MLP processes groups allocating zones of work and a given the proper number of processor to each MLP process group to effectively load balance the problem. Each MLP process group is then tasked with properly touching memory in a way to again lay out the local data arrays in an "optimal way", which may be as simple as first touch. MLP process groups may also make local copies of data arrays in order to reduce contention and latency in accessing the global data.

Each individual MLP processes groups then takes a time step for each of the zones under their control. After all zones in a process have taken a step, the boundary data is updated in the global shared arrays, and the MLP process groups then barrier synchronize. I/O is performed if necessary (usually by the master process to keep it simple), and the processes continue to the next time step. This approach was inserted into the Cray C90 version of the OVERFLOW CFD code described below.

5.3.6 OVERFLOW Origin 2000 Optimization Effort

The Origin optimization effort for OVERFLOW began with executions of the standard C90 version of OVERFLOW on the Origin 2000. These early executions were conducted with OpenMP directives used in place of all Cray microtasking directives found in the code. Cray executions had shown that the code was capable of sustaining 4.6 GFLOP/s on a dedicated 16 CPU C90 system.

Early executions on the Origin 2000 were initially disappointing with sustainable performance approaching only about 1 GFLOP regardless of the CPU count used on the problem. All executions failed to scale past 8 CPUs, due to NUMA effects, and the inherent limitations of OpenMP loop level parallelism.

The major issue with the first version of the code was that it serially processed through all of the zones one at a time for each time step. This worked well on the moderately parallel C90, but was not efficient on the target Origin system as it implied that each zone had to execute well on large numbers of CPUs or the code would not scale overall. As mentioned earlier, it is particularly difficult to get OpenMP loops to scale to large CPU counts, unless great attention is paid to the allocation of data within the NUMA memory system. Fixing this to execute well on hundreds of CPUs would require a complete rewrite of the entire OVERFLOW code.

After reviewing the results of the initial Origin work, the first optimization effort focused on increasing the single CPU performance. Profiling of the code showed a very flat profile with no routine taking over 5% of the CPU time, and most taking less than 1%. This indicated that many routines would have to be changed in order to increase the overall performance of the code.

Efforts at single CPU performance were quickly abandoned as too invasive and dangerous from a code validation and verification standpoint. Highly modified code would not likely be accepted by the production community due to lack of extensive validation necessary to ensure the accuracy of the code. Further, the individual maintaining the code was not amenable to incorporating massive changes into the standard release which becomes a significant maintenance issue over successive releases of OVERFLOW.

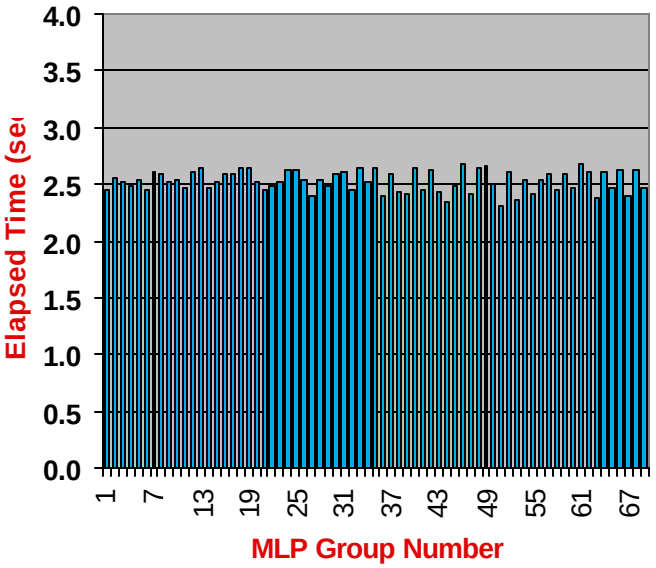
The second focus was to adopt the MPI data decomposition approach to the problem, and attempt to solve the multi-zone problem in a coarse-grained parallel fashion. An existing MPI version of the code was available, but suffered from scaling and correctness issues at the time. As a result, an alternative coarse-grained approach was developed. This became MLP.

The MLP technique was developed and inserted into the OVERFLOW CFD code within 6 months from receipt of the code. Four of these months were devoted to learning the code and the single CPU optimization effort. Only two months were needed to develop and insert MLP. Ultimately, only a few hundred lines of code were changed from the original C90 version of the code. Even today, the code is still 99% identical to the original C90 code with long vectors throughout. No single CPU optimizations have been performed on the code.

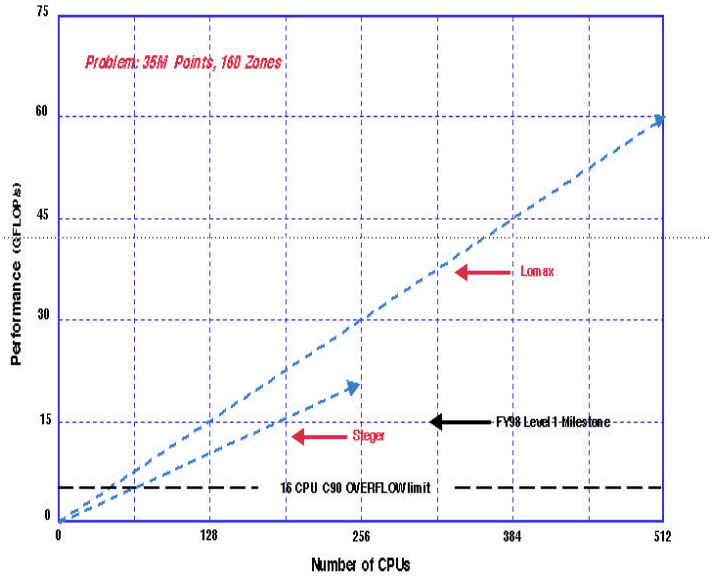
The end result of the optimization effort was a code that distributes the zones of a multi-zonal problem across a modest number of MLP processes (typically 8-64 depending on problem size). These processes then execute their assigned zones using fine-grained parallelism on 8-16 CPUs within each MLP process.

5.3.7 OVERFLOW-MLP Results

A test case was defined at the beginning of the Origin 2000 Optimization Effort to evaluate the performance of the OVERFLOW-MLP code. The case is a real life 35 million point problem describing a large transport aircraft configured for landing and is used to analyze the wake vortex issues of these large aircraft. It has impact in flight safety, airport capacity, and high lift systems. The 35 million points are distributed across 160 zones of widely varying size from 1.4 million to 11 thousand points. The problem requires about 2500 time steps to converge for moderate angles of attack. The solution requires hundreds of CPU hours on a Cray C90 system and sustains about 4.6 GFLOP/s on a dedicated 16 CPU system. This problem fully stress tests the MLP code and Origin 2000 system hardware and software.



The figure above presents histogram showing the results of one time step as executed on the lomax 512 CPU Origin 2000 System (R12K/300 MHz) at NASA Ames. A total of 69 MLP processes were defined for this execution. Each MLP process had a different number of CPUs assigned for the fine-grained parallel work, and each was assigned a different subset of zones to solve. The largest grid was assigned 32 CPUs and the smallest grid was assigned 1 CPU. Each vertical bar in the histogram presents the elapsed wall clock time in seconds required to solve one time step for all of the assigned zones in the given MLP process. The worst case time was 2.6 seconds, resulting an efficiency of over 90% in load balancing.



The figure above presents the results in a more dramatic format. This is a plot of GFLOP/s versus CPU count from 1 to 512 CPUs. The figure shows a remarkable linear speedup with increasing CPU count for OVERFLOW-MLP. Overall, the OVEFLOW-MLP code executes the 35 million point problem at a sustained rate of 60 GFLOP/s on 512 CPUs. This amounts to 13x faster than the dedicated C90 system, at a cost performance ratio in excess of 33x! As mentioned above, the execution time per time step was 2.6 seconds, resulting in a fully converged solution (2500 steps) in less than 2 hours of elapsed time.

Though not shown, the OVERFLOW-MPI results for this same problem cease to scale past 100 CPUs. This falloff in MPI scaling is typical for many of the MPI based production CFD codes at NASA Ames, and is a function of the relatively high latency found in the MPI based CFD codes. In this particular case the MLP approach is about 5x faster overall than the MPI version with scaling continuing to increase linearly at 512 CPUs.

The 512 CPU (lomax) results are interesting when contrasted with the 256 CPU (steger) performance numbers. In effect the execution took place on twice as many CPUs, but the performance went up by a factor of three. The reason is primarily because of the doubling of cache per CPU. Since OVERFLOW is memory bound, the fact that there is a total of four times the cache (twice the CPU count and twice the cache per CPU) on the 512 CPU system made a significant difference in performance.

Overall impact to the code in terms of modifications resulted in less than 1% (1000 lines) of code changes contained in 6 routines during the MLP optimization effort.

5.4 Future Work

Even though OVERFLOW-MLP performance is very high relative to the C90, its current performance is substantially less than could be obtained on the new microprocessor based systems. Historically, the entire focus on optimizing OVERFLOW has been to increase the efficiency of fine and coarse-grained parallelism. Virtually no single CPU optimizations have been performed. Examination of the code has shown that there is perhaps a factor of two in runtime reduction still available if this activity is undertaken. This work will be substantially more involved than the work to date, as many routines will change and the verification and validation effort will expand accordingly. Current plans are to work on generally applicable strategies to microprocessor optimization.

5.5 Substantial Improvement in Capability

Executing production CFD codes 13x faster than a dedicated 16 CPU C90 system is a major breakthrough in large scale computing within the United States. In particular, we can now finally imagine executing high fidelity RANS simulations of full and highly resolved aircraft in a matter of 100 minutes or less. We can now finally imagine invoking computational steering for real problems in which we fly full aircraft at high resolution in the ‘electronic” wind tunnel, changing angle of attack, etc. at will and get the answers back in a few hours. One can envision scenarios in which a team of engineers could schedule a system, such as NASA’s Lomax, for a week to evaluate a vast range of scenarios, cutting months or even years off the design cycle of new aircraft. This has been an elusive goal for decades. It appears that it is now about to be realized.

6 Summary and Conclusions

The recent addition of the lomax single system image 512 CPU Origin to the NASA Ames NAS facility has proven to be highly successful. Significant contributions and improvements to the state of the art have been made in operating systems, systems integration, application programming techniques and the resulting performance of the CFD code OVERFLOW that far exceeds any pre-existing simulation capability in the United States.

Significant progress has been made in the understanding operating systems issues with regard to high processor count ccNUMA systems. Several additional controls that significantly improve performance and reduce run time variability have been incorporated into the standard IRIX operating system.

NAS has successfully replaced its primary production computing resource with a system that is more reliable (C916 @ 15.0 days vs. 256p @ 20.8 days 512p @ 4.9 days), has higher sustained throughput (2.5x), has a substantially higher capability in large scale applications (256p @ 5x, 512p @ 13x), and at a substantially reduced cost (256p @ 1/5 and 512p @ 1/3).

The success with OVERFLOW-MLP demonstrates that that MLPlib and MLP programming techniques are an important step forward in improving parallel scaling efficiency and algorithmic efficiency while at the same time maintaining simplicity and elegance in its implementation in scientific application codes critical to NASA's continuing missions success. Further that the technique is generally applicable to many other science areas of NASA/National interest including weather modeling, molecular dynamics, and stellar dynamics,

This success proves the viability of large-scale ccNUMA architectures. While reliability remains a concern, future generations of the Origin architecture increase connectivity and integration thereby significantly reducing latency, part count and complexity, which should serve to support even larger configurations or much more reliable 512p systems.

Future directions should strive to continue to push the limits of SSI and eliminating single points of failure while maintaining the appearance from an application of a single cache coherent shared address space that spans the entire system.

7 References

1. D.M. Pressel, W.B. Sturek, J. Sahu, K. R. Heavy, How Moderate-Sized RISC-Based SMPs can Outperform Much Larger Distributed Memory MPPs., 41st Cray User Group Conference Proceeding, May 24-28 1999.

2. J. Petersohn, K. Schilke, Experiences with the SGI/Cray Origin2000 256 Processor System Installed at the NAS Facility of the NASA Ames Research Center, 41st Cray User Group Conference Proceeding, May 24-28 1999.

3. Taft, Multi-Level Parallelism, A Simple Highly Scalable Approach to Parallelism for CFD, HPCCP/CAS Workshop 98 Proceedings, Catherine Schulbach, editor.

4. W. Gropp, et al. Using MPI: Portable Parallel Programming with the Message Passing Interface, MIT Press, Cambridge, MA, 1994.

5. OpenMP Architecture Review Board, OpenMP. A Proposed Standard API for Shared Memory Programming. October, 1997.

6. Beguelin, J. Dongarra, G. A. Geist, et al. A user's guide to PVM: Parallel Virtual Machine. Technical Report TM-11826 Oak Ridge National Laboratory, 1991.