Engineering Excellence: DEC OSF/1 Symmetric Multi-Processing

Jon A. Hall
Senior Manager
UNIX Software Group
maddog@zk3.dec.com

Digital Equipment Corporation
Nashua, New Hampshire

In this paper, the term "SMP" will refer to "Tightly Coupled" systems, where CPUs can share all or a part of main memory with other CPUs, and have equal access to all peripherals. There are also "Loosely Coupled SMP" systems and "Closely Coupled SMP" systems, but these will not be discussed.

The specific implementation of SMP in DEC OSF/1 V3.0 will be discussed and why it is a superb implementation of SMP.

1. What is SMP, and what are its uses?

Multiprocessing (MP) is the ability for an operating system to use multiple processors (which usually means multiple CPUs) in a single enclosure and sharing all or part of main memory and access to peripherals. There are two kinds: Asymmetric MP (ASMP) and Symmetric MP (SMP).

Typically under ASMP one CPU does all of a certain kind of work. For example, one CPU (usually called the "master") may execute all of the system calls, scheduling and I/O. When any of the other CPUs want to do I/O or a system call they are blocked, and the master CPU will schedule another process for them. Then the master CPU must do the rest of the I/O or system call. The problem with ASMP is that under most types of loads, the "master" CPU becomes too busy to schedule all of the other processors. Not only can it not keep all of the other processors busy, but it can not even do all of its own work or its own share of user code. The master CPU becomes a bottleneck for total throughput of the system.

In SMP any of the CPUs can do any task, including scheduling work inside the kernel. All processors have equal access to the operating system services. This way, no one single processor becomes overloaded.

SMP is particularly valuable in CPU bound systems where there are a large number of individual processes or threads of execution. A process in DEC OSF/1 is actually a single thread of execution which is associated with a piece of virtual memory. In DEC OSF/1, more threads of execution may be spawned from the original thread of execution, and each thread may have its own priority and scheduling algorithm.

SMP is useful to the customer, since a CPU-bound system may easily be given extra capacity by adding additional CPU boards. Thus if a customer sizes and purchases a system, but later needs additional CPU capacity, it is normally very easy and cost-effective to add another CPU to an already existing SMP system.

It is important to note, however, that CPU-bound systems with a single-threaded application bottleneck will typically not benefit from an SMP solution. Therefore SMP implementations usually benefit servers and time-sharing machines more than they benefit the traditional single-user workstation.

Only when the CPU bound application in a single-user workstation has been decomposed into multiple threads of execution will that application benefit from SMP. Some sophisticated current-day applications do benefit from SMP on a workstation. These applications may be doing some sophisticated work with the X Window System, and keep their data in a distributed relational database. For these applications, there are at least three processes involved in the running of the application, and therefore three threads of execution which can be assigned to individual CPUs.

In UNIX there has traditionally been the use of fork(2) to create another thread of execution in "C" programs, and now the advent of FORTRAN 90 language directives (part of the High Performance FORTRAN industry standard extensions to FORTRAN 90) which enhance semi-automatic decomposition of FORTRAN programs. Therefore SMP becomes even more valuable as a means of obtaining additional CPU resources.

---

† UNIX is a registered trademark licensed exclusively by X/Open Company Ltd.
† Alpha AXP is a trademark of Digital Equipment Corporation

2. Issues in Implementing SMP

Customers question why implementing SMP is so hard. Why do computer companies make such a big deal over it, and why are some implementations considered better than others.

The main reason for implementing SMP is to get extra system performance out of adding an additional CPU. The amount of extra performance that you get out of adding an extra CPU is an indication of how well the system "scales". If by adding an extra CPU, you were able to put twice the number of processes on the system and run them in the same wall-clock time, you would say that you have "100% scaling". Or if you put in a second CPU, and the amount of time that it took to run a program in a CPU-bound system was only 1/2 the time, you would say that you received "100% scaling".

There may be reasons why you would not see this type of performance increase. Perhaps by adding the second CPU you relieved your CPU bottleneck, but your I/O subsystem was now at saturation. Or perhaps by adding the second CPU (and the additional processes) you now were bottlenecked by the amount of main memory you have, and your system will start paging and swapping. But given that you have enough memory, and given that your I/O subsystem can feed data to the programs fast enough, by adding a second CPU to the first, you should be able to double your load at 100% scaling.

However, now there are two CPUs which are looking at the same data in the kernel, executing the same code, trying to use the same devices. These CPUs will be trying to update the same tables, and they may interfere with each other. To keep the two CPUs from interfering with each other, different techniques are used, and these techniques reduce the efficiency of the total system but are required to maintain system integrity. In fact, you will see *always* less than 100% improvement. Perhaps it will be 99.9%, or 95%, or even as low as 80%, but it will *always* be less than 100%, due to the overhead of managing it.

It is also true that different types of loads will see different scalings. For example, highly CPU-bound technical computing jobs which do very little I/O may see very high scaling factors, while job streams which are high in I/O may see very low scaling factors. In fact, in at least one case which happened (fortunately) a long time ago, adding a second CPU to a job load which was high in I/O actually saw *less* performance from the system than with a single CPU. In other words, the system performed better when you turned the second CPU off than when you had it turned on. This is particularly true of ASMP implementations.

Another issue with SMP systems is that of "deadlock". One of the ways that CPUs keep out of each other's way in an SMP implementation is by "locking" resources. Devices and data all may be "locked" by a CPU in order to update some table or control structure inside the kernel. Unfortunately, if these devices are not allocated and de-allocated properly, one CPU may have devices that a second CPU needs, and vice versa. Neither CPU will give up what they have, and neither can proceed forward until they have what is being held by the other CPU. This is known as "deadlock", and in worst cases all throughput in the system eventually halts, requiring a re-boot.

In the early days of UNIX operating systems, UNIX kernels were single CPU, and non-preemptive. The second term means that the kernel would not be interrupted from whatever it was doing until that process or thread was done. This made kernel writing "simple" (or at least as simple as kernel writing ever would be), since as soon as the kernel was entered, the interrupt level was raised very high, and remained high until the kernel was exited. Therefore all the table updates inside the kernel would be completed without some other thread of control coming along and interfering. Unfortunately this meant that real-time activity or high-priority tasks could not be scheduled until the thread that was already inside the kernel was finished, and had returned to user space.

Later in UNIX history, a second CPU was added in what was known as "Asymmetric Multiprocessing". Using this technique, one CPU did all the scheduling of tasks, devices, I/O and "kernel" things, while the second (or third or fourth) CPU worked strictly on user-level processes. Unfortunately in a lot of cases, the "master" CPU became hopelessly overloaded very quickly and the throughput of the system rarely scaled past two or three processors (and in many cases actually decreased as mentioned above). The "master" CPU was also practicing a technique known as "funneling", which meant that certain tasks inside the kernel (such as scheduling or I/O) are always performed by one CPU.

In recent years some UNIX kernels have become preemptive in order to do realtime work. This means that a stream of execution going through the kernel *could* be interrupted from what it is doing, and have to return to that place later to finish what it was doing. In these cases (with a single CPU) there are "must finish" places inside the kernel where certain work must be done before the CPU can be interrupted. These places have to be identified, and locks executed to make sure that the tables will not be updated until the first thread of execution has a chance to finish what it was doing.

However tricky the preemptive kernels are to code, SMP is an order of magnitude above that. First of all, in a single-CPU system, you are guaranteed of not having cache consistency problems between CPUs. After all, you only have one CPU, and its cache is always consistent with itself. However, with multiple CPUs, you may have one CPU which has a piece of critical data in its cache, and another CPU is looking at bad data. Secondly, there are now multiple CPUs trying to access the same sets of data structures inside the kernel "at one time", and the ability to keep all of them coordinated is strained.

In addition, over the years the separation between CPU speed, memory speed and I/O speed has grown larger. Techniques which were practical when CPUs were 1 SPEC, memories 4 MByte and disks 2 MByte/second are suspect when CPUs are 110 SPEC, memories are 96 MByte and disks (on a good day, with the wind at their back, and going downhill) are still only 20 MBytes/second†. Therefore it takes a good deal of study and analysis to engineer the type of locks which will give optimum performance in today's multiple CPU systems which can easily aggregate close to 1000 SPECs or beyond, when disks and main memory are "slow" and "small".

One item that makes it a lot easier (or looking at it another way, without this it would be a lot harder) to implement SMP is a highly structured, modern kernel. By having modularity in the kernel where messages are passed and locking points are well defined, it is a lot easier to implement SMP than with the older style of "onion-skin" kernel. The problem with an "onion-skin" kernel is that it is difficult to see how many threads of execution will be going through a section at one time, and where they will be coming from. With a modern kernel, and one that was *designed* for SMP, it is a lot easier for the engineering staff to identify locking points and areas of contention, then to have to retrofit this to an older style of kernel.

So what makes up a "good" SMP solution? Normally it is the proper application of a series of coding techniques after a long time studying a well-structured code over a wide variety of load conditions.

† Unless, of course you buy Velocitor disk subsystems from Digital, which can *sustain* data rates of 30 Megabytes/second *or more*. But this comment was aimed at the use of the fastest "fast" wide SCSI disks today.

Let's look at some of these techniques, and see how they may be used.

3. Techniques of SMP

Some of the techniques used in developing an SMP system are:

- o Funneling vs Locking
- o Coarse Granularity vs Fine Granularity
- o Simple Locks vs Complex Locks
- o Algorithm replacement or change

We will study each one in turn, but the reader should understand that in a good SMP system, *all* of these techniques may be used in one part of the kernel or another. The secret of a well-designed and implemented SMP system is using the *right* technique in the *right* place.

3.1. Funneling

Funneling is the technique of forcing execution of a particular set of code to one CPU. Funneling is the exclusive technique used in ASMP systems. Normally funneling is thought of as "bad", but in some cases funneling can actually be good, or at least "low impact".

Funneling is usually satisfactory for slow devices of low usage. As an example, the ISO 9660 file system (used with CDROMs) is funneled, since there typically are few of these devices on a system, and they are slow devices. The overhead of making this file system "SMP safe" would probably cause more CPU usage than would have been returned. Therefore the ISO 9660 file system was "funneled".

Another case of funneling is in the "reboot" system call. For reasons which are intuitively obvious, there was no "return on investment" of making this system call "SMP safe".

One additional area of funneling is interrupt handling. All interrupts are handled by one processor. However, immediately after fielding the interrupt, the processor schedules a thread to process that interrupt which may be run on any other processor (including the one that fielded the interrupt).

While some people in the field maintain that all interrupts being handled by one processor will degrade the system, Digital's engineering group did *exhaustive* study under different loads, and found no degradation to the over-all system performance by having one processor field all interrupts. To *ensure* that funneling interrupts to one processor will not degrade system throughput, the load balancing algorithm adjusts to reduce the tasks that the processor which is fielding the interrupts has to do based on the number of CPUs in the system, thus guaranteeing the CPU handling the interrupts the capacity it needs.

In addition, Digital's UNIX engineering group found that by *distributing* the interrupts, there was significant disruption of the warm cache in the CPUs between scheduling of threads. In other words, distributing the interrupts would have *degraded* over-all performance *more* than funneling does.

3.2. Locking

The "opposite" of funneling is locking. Here is where multiple threads of execution are allowed into the code, but certain areas of code or data are "locked" from another CPU updating (or in some cases even reading) it until the first CPU is finished.

Various types of locks are used for different purposes, and different "granularities" are also used. Let's speak of granularity first.

3.2.1. Fine-grain versus coarse-grain granularity

Inside the kernel are data structures which have to be shared between all of these CPUs. Sometimes these pieces of data are arranged in a table or inside some other structure (a linked list, for instance) and these data structures have to be updated all at one time. For instance, it would not do to have a linked list only partially linked before a second processor moved in to also manipulate that linked list.

The reader at this point also has to understand the difference between an "outdated" piece of data, and an "invalid" piece of data. An outdated piece of data is one that was correct at one time, but now is technically incorrect, but still a valid number (such as a timestamp). Perhaps our linked-list data structure is being updated which makes the data value in question "outdated". The timestamp is still a valid number, and useful to report status to human beings, but from the view of the operating system, it is not "up to date". An example of an invalid piece of data would be a timestamp that was in the middle of being updated by one CPU when a second CPU comes along to read it. The second CPU would see a timestamp that was *never* correct, and if this time was reported (even to a human being), it would be useless.

In some cases it is not necessary to prevent two CPUs from "treating" the data. If one CPU is updating the data, and another is only reading it, it may be perfectly all right for the second CPU to get an "outdated" piece of data as long as it is a valid one. For example, the modification time of a file might be outdated by a microsecond, but since the granularity of the timestamp on the file is seconds, it may be all right for the system to get an "outdated" piece of data to use, as long as it was a valid time (e.g. not a negative number). Likewise it may be all right for a second CPU to read or manipulate some other part of a data structure while the first CPU is manipulating some other part.

The ability of a data structure to be manipulated by more than one CPU is called the "granularity" of the structure, or the "granularity" of the locking. This really refers to the parallelism which will be allowed through a subsystem that is manipulating these data structures. Typically the finer the "granularity", the greater the parallelism which can occur. However the finer the granularity, the more time spent locking and unlocking locks.

If a CPU is blocked from looking at a particular data structure, it can only do one of two things; either "wait" for that data structure to become available to it, or try to go off and do something else while the data structure is being updated (if there is something else the CPU can do).

Therefore it would seem best if the data structure was broken down into very, very small parts and each part was locked and unlocked as needed. This is known as "fine granularity". However, "fine granularity" has the issue that locking and unlocking locks uses CPU time that basically adds nothing to the solution of the end-user's problem. The finer the granularity, the more the CPUs spend time in locking and unlocking locks, and the SMP system becomes less efficient.

Course granularity is having few locks per data structure, therefore freeing up CPUs from locking and unlocking as many locks in doing their work, thus cutting down on overhead. So "coarse granularity" seems attractive. However, coarse granularity has the issue that large sections of code may be blocked from having a second, third, fourth or more CPUs accessing it. If that data structure is used a great deal inside the operating system and if the update is relatively slow with respect to wall-clock time then the other CPUs may be idle, and once again total system throughput suffers.

Therefore it is best to study the data structures inside the kernel, and use the correct granularity. Locks should allow read-access when permissible, even to re-designing data structures for better granularity with less locking overhead.

Which brings us to the topic of Locks, their types and efficiencies.

3.2.2. Types of Locks

The different types of locks are (engineers, please pardon the simplicity of this section) "spin locks" and "semaphores". Under semaphores (also known as "Mutexes") are "Fast Mutex", "Recursive Mutex" and "Blocking Mutex". Each one of these locks has its place inside the kernel.

3.2.2.1. Spin Lock

The simplest lock is the spin lock. When a CPU is starting to look at the data structure, it tests a piece of data common to all the CPUs to see if any of the other CPUs are looking at the data structure also. If none are, it sets that piece of data (the lock) to "set", and starts to manipulate the data. When another CPU starts to look at the data it sees the lock is set, and then loops, constantly looking at the lock to see if it is unlocked by the first CPU. When the first CPU is done, it clears the lock, and continues on. The waiting CPU then sees that the lock is cleared, sets the lock to indicate that the data structure is "busy", and accesses the data.

Spin Locks are very simple to implement, and are the least overhead of all the locks for data structures where the update time is very short. These are typically structures which reside completely in semiconductor main memory (not out on the disk) and will only take a machine cycle or two to update. Therefore they can be very efficient. However the efficiency of these locks has changed over the years. As was pointed out earlier, as CPUs got hundreds of times faster and disks were only two or three times faster, areas that had been locked with spin locks before now demanded a lock that would allow less idle CPU time. With a spin lock, the waiting CPU *is* idle. It would be nice if the waiting CPU could go off and do other useful work if it had to wait longer than a certain period of time, or a certain number of CPU cycles.

Enter the Mutex.

3.2.2.2. Mutexes

Mutexes allow the waiting CPU to re-schedule itself by "blocking", and going off to do other types of work. Later, when the data structure is unlocked, the CPU goes back and continues its work. In the case of threads waiting to unlock for updating the structure versus threads waiting to unlock for reading the structure, the threads which are waiting to read are given preference.

It is obvious that any mutex will take more overhead to set up than a simple spin lock. First the same type of lock testing has to be done to see if the data structure is free. Secondly the thread that is blocked has to re-schedule itself to do other work. Finally that thread has to come back and be re-scheduled again to complete the task it tried to do when it was blocked.

In order to cut down on this overhead, several different Mutexes were used: "Fast", "Recursive" and "Non-blocking". There are different names for these, but we will use these names for ease of understanding.

3.2.2.3. Fast Mutex

The Fast Mutex allows a single thread to lock the data structure. A second thread coming along will block as it tries to access the data structure by calling the same Mutex code. This Mutex has very little overhead, and is relatively "fast".

Unfortunately it has a problem. If the thread that executed the Fast Mutex in the first place (the one updating the data structure) for some reason calls the same Mutex code again before unlocking it, the entire thread will stop. Deadlock will occur. In situations where a thread *might* call a Mutex without unlocking it first, this is not acceptable.

3.2.2.4. Non-Blocking

To answer the issue that exists with the Fast Mutex, a non-blocking Mutex was created. Using slightly more overhead than the Fast Mutex, the Non-Blocking Mutex will simply deliver an error message if a thread of execution tries to enter an area already locked by that thread. Thus the thread can test for this error condition.

3.2.2.5. Recursive

From time to time it is desirable to have a Mutex that allows a thread to call itself, and to not allow any other thread to enter a section of a data structure until the first thread "returns" as many times as it was called. This is especially useful for traversing lists, strings, etc. A "Recursive Mutex" was created for this purpose.

While the overhead difference of these Mutexes may not seem significant, when executed hundreds of thousands of times each second the overhead can become considerable, perhaps even consuming a CPU of time in executing this "overhead code" which (in most cases) is not necessary except in an SMP environment. For example, the overhead of a Mutex is greater than two times the overhead of a spin lock, assuming that the waiting CPU does not "spin" very much.

Therefore good mutex selection is critical.

3.3. Algorithm Replacement

Even better than good lock selection in an SMP implementation is to re-design a kernel algorithm to reduce the lock overhead, or even eliminate a lock altogether.

As an example of lock elimination, the time structure in UNIX systems (used by the operating system many times each second) is called "timeval". On Alpha systems this happens to be a 64-bit structure. The Alpha architecture ensures atomic operations on 32 and 64-bit data structures. It was impossible for a CPU to pick up "half" of the timeval structure while another CPU was updating it. Therefore the necessity of having to lock the timeval structure while the CPU manipulated it disappeared.

In an example of algorithm replacement the process control structures and the methods of transversing them, were scattered between many tables. This made it difficult to allow fine preemption in a real-time environment, and also had a lot of overhead in an SMP environment. By re-designing the process control structures many of the fields used to "navigate" between the structures were eliminated, the structures had better "locality" (i.e. were closer together in main memory, therefore more likely to be in a cache of some type) and eliminated the need for several locks.

A final example of algorithm replacement is in the PID table. The PID table is used to find processes (and stands for "Process ID"). In most UNIX systems this table is static, and tends to be fairly large, with linked lists providing the method of searching. In DEC OSF/1 V3.0 this table was made to be dynamic, with the table to be searched being a much smaller "pidtab" array. The PID number is used as an index to the pidtab (making for *very* fast location of the entry) and the next available entry is pointed to by a "freepid" list. This also eliminated *seven* different locking points in an area of the kernel that is used extensively by every CPU.

3.4. Different Strokes for Different Folks

In addition to the study of different SMP techniques, there is the application of these techniques to different job loads.

For instance, it is not necessary to have a lot of the locks in a single CPU situation. Therefore DEC OSF/1 has the ability to "patch out" these locks either as a configuration-time option, or as the operating system is booting in a single-CPU environment.

Likewise, if a customer is doing real-time in a single-CPU environment, the locks for preemption will be applied, but the locks for SMP will not. If a customer has an SMP system, but does not want real-time, the locks which are only relevant for real-time will be patched out. And finally, the customer that wants real-time and SMP will have an environment which includes both types of locks.

In all, DEC OSF/1 V3.0 has *five* levels of "lockmodes". To our knowledge at the writing of this paper Digital is unique in the industry in offering locking techniques configurable to the environment.

So far only four modes have been described. The fifth mode is "Lock Debug Mode", and has to deal with deadlock detection, which we will now discuss.

3.5. Deadlocks and SMP quality

An SMP system is useless unless it is a quality system. Digital, with its years of SMP engineering experience, designed quality in from the start.

The term "deadlock" was described earlier in this paper. One of the ways of avoiding deadlock is to lock the locks in a particular order (called "assertion ordering" or "hierarchy ordering") and to unlock them in reverse order. This means that it is impossible for two threads to have deadlock occur. However, in an operating system kernel this is easier said than done.

Even in a single-CPU operating system it is possible for deadlock to occur, or have the potential for deadlock to occur, and not experience it for years. When SMP systems ship, this potential escalates immeasurably. Digital's UNIX engineers built the original locks used for real-time preemption with debug code built into it. As engineering ran the kernel in the labs, they left the debug code turned on, which checked to make sure that the locks were locked and unlocked in the proper order. While the systems ran a great deal slower due to this checking, they uncovered hundreds of potential problems that might not have otherwise been uncovered until the code arrived in customer's hands.

In addition to checking the locking hierarchy. the same debugging code keeps count of lock usage, and allows the engineers to see how often a particular lock is used under different loads. This gives good indications of the next place where algorithm restructuring or lock replacement might be attempted.

Debug mode is also configurable by the customer, however it is not recommended in normal production environments due to the overhead it creates in the system. It is very, very useful if the customer starts to have problems, since the situation that creates a deadlock is usually long gone by the time the deadlock is evident. The debug mode logs the locking information necessary to find the problem.

3.6. Scheduler changes

Another issue in SMP is processor affinity. Customers often wish to apply processor affinity to one process or a particular set of processes. This may be to dedicate a certain amount of CPU power to a process, or (in the case of a real-time system) hope to cut down re-scheduling time by having the same processor (and therefore the same processor cache) run the application thread.

Processor affinity should not be confused with "funneling". Funneling is a technique typically used inside the kernel to allow only one CPU through a piece of code at one time for control purposes. Processor affinity is used outside the kernel to try to assign a processor or group of processors to a process or group of processes for better throughput.

DEC OSF/1 V3.0 allows "soft" processor affinity. This means that the customer can set up processor affinity for their CPUs, but if that CPU goes down, another CPU takes over. Also, if the assigned thread requires services only available from another processor, the affinity may be compromised for a short time. In addition, the operating system keeps statistics on jobs, and the operating system will try to assign the same threads to the same processors whenever possible.

Other than processor affinity being assigned (either by the customer or automatically by the operating system) the scheduler will attempt to distribute the processes across all available processors evenly.

In DEC OSF/1 V3.0, the system automatically tries to assign processor affinity, unless over-ridden by processor affinity set up by the systems administrator.

4. Field Testing and Quality Control

Despite the care taken in engineering DEC OSF/1 V3.0 for SMP, it is a complex task, and some errors will occur. It is for this reason that the Digital UNIX group has embarked upon one of the largest quality and field testing efforts in its history.

SMP was running for over a year inside our laboratories in Nashua, N.H. before we released it to field test. The field test was in two stages, with Internal field test commencing in November of 1993, and External field testing starting in March of 1994. The SMP development group has been using V3.0 for their normal day-to-day work since February of 1994, not just "testing it" in the lab. An update to External field test was released in May of 1994, with volume ship planned for the July/August timeframe. In addition to the formal field test sites, Digital put forth a heavily monitored pre-release program to ISVs and large end users, including several large data-base customers who have been jointly working with Digital for the past two years in the design and implementation of our SMP.

Finally, the engineering group developed a multi-threaded, multi-call test suite to create thread races between different system calls at the same time in a random fashion to check out inter-system call issues.

Digital feels that the DEC OSF/1 V3.0 SMP implementation will be very stable from first ship.

5. SMP is not the kernel alone

A good implementation of SMP is not just the kernel. A good implementation requires supporting programs to make it successful.

Other vendors shipped their SMP version of the operating system, but did not ship tools for application development or debugging until six months (or more) after the SMP implementation was shipping.

The end-user customer did not see off-the-shelf applications by commercial vendors which could take advantage of SMP until much later (perhaps even a year after first ship).

In the world of "C" programming, software engineers have often used parallel programming techniques (through the use of threads and shared memory) to get better performance on both single CPU and SMP systems.

However there are products which make it easier for C and FORTRAN programmers to take advantage of multi-threading. KAP for DEC Fortran V2.0 and KAP for C V2.0 are being actively used in SMP benchmarks at all of the Digital benchmark centers. Features/benefits of the KAP preprocessors include:

- Automatic decomposition of FORTRAN 77 and C programs
- Optional directed decomposition
- RTL based on OSF/1 threads, providing good efficiency

- o Parallelization is combined with KAP's usual scalar optimizations
- o Tested, mature technology

These versions of the KAP development tools will be available with the shipping of the DEC OSF/1 V3.0 layered products consolidated CDROM.

Digital, through its High Performance Computing group, has also formulated a series of libraries and compilers that allow a customer to develop a multi-threaded FORTRAN program. In fact, Digital has extended its Parallel Software Environment (PSE) and DEC Fortran 90 compiler to embrace industry standard extensions (known as "High Performance FORTRAN or "HPF") to FORTRAN 90 allowing for directives which tell the compiler how to break up the program for best parallelism. This compiler suite is in field test now, shipping at the same time as DEC OSF/1 V3.0, allowing ISVs to develop high performance applications which will be available when DEC OSF/1 V3.0 ships, or slightly after. Digital was the first vendor to support both FORTRAN 90 and the High Performance FORTRAN extensions.

In addition, Digital has a debugger which has the capability of debugging multi-thread and multi-process applications and a profiler that can profile to the thread level. This will be shipping at the same time as V3.0, and some of the system management utilities shipped with DEC OSF/1 V3.0 will be modified to give information about each CPU's utilization.

6. Conclusion

This paper has talked about many techniques used in Symmetric Multi-Processing, most of which are well known in computer science. The point is not that new techniques are used in implementing SMP, but that the techniques used are used carefully, with the right technique being used in the right place for maximum throughput in the system. Digital engineering has made that investment of time and engineering skill studying and implementing DEC OSF/1 V3.0 for an efficient SMP implementation. In addition, this paper has shown that the OSF/1 base kernel from the Open Software Foundation was a good choice for implementing such a system due to its modularity.

I have not included the charts showing performance in this paper, simply because it continues to get better every day, and to embed the charts in this paper would not give the reader the current status. However, the charts are available, and will be placed out in the public directories on gatekeeper.pa.dec.com as they are updated.

DEC OSF/1 is a world-class UNIX operating system, and the V3.0 SMP implementation is a symbol of that class of engineering that is a characteristic of the Alpha Generation efforts.