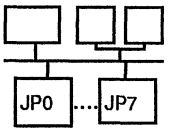
Data General

DG/UX Technical Brief

November 11, 1992

Information About AViiON® Systems from Data General's UNIX® Development Group



In This Issue:

8-Way Symmetric Multiprocessor Systems

Contents

Multiprocessor AViiON® systems provide users with a unique and powerful way of increasing their data processing throughput. However, we're concerned that there are misconceptions about how AViiON multiprocessing systems work and misconceptions about how to take advantage of a multiprocessing system's power.

Two questions that we hear often are "Am I really going to see a performance increase if I go from a single processor to a multiprocessor configuration?" and "What kinds of multiprocessor performance increases can I expect?" We want to answer these kinds of questions, so we'll devote this technical brief to discussing Data General's approach to the implementation of Symmetric Multiprocessor Systems (SMPs) in the AViiON series of computers. We'll explain how the AViiON SMPs work and how they provide a performance advantage. We'll try to clear up the common misconceptions about AViiON multiprocessor systems. Finally, we'll give you some hard facts that show the kinds of performance advantage that can be obtained when you use two-, four-, and eight-processor AViiON computer systems.

Please Note—This Technical Brief contains updated test results of the new AViiON AV6280 system, running with one, two, four, and eight processors. This Technical Brief replaces the July 31, 1991 Technical Brief (012-003886), which contained information for single-, dual-, and quad-processor systems.

Produced on a Data General AViiON AV4000 with FrameMaker® 3.1X.

012-004246-00

Multiprocessor Myths

Let's start by dispelling some common misconceptions about multiprocessing. We'll talk more about these issues in the rest of this brief.

Myth: *Multiprocessing systems don't provide a significant performance advantage.*

Not true—multiprocessor systems provide real-world performance advantages; they are not just a theoretically interesting experiment. In fact, multiprocessor AViiON systems have been in customers' hands since April 1989.

Myth: *I need to change my application code to take advantage of a multiprocessor system.*

Not true—you don't need to change your application software to run it on an AViiON multiprocessor system. You can run the exact same applications on single-processor systems and multiprocessor systems.

Myth: *Parallel processing and multiprocessing are the same thing.*

Also not true—parallel processing systems may use special compilers or special system calls to produce parallelized programs. These special, non-standard system calls can affect application portability. In contrast, AViiON SMP systems support UNIX® standards such as System V, BSD, and POSIX.

Myth: *I can achieve significant performance gains just by adding another processor to my system.*

Not necessarily true—you will see significant performance gains if you add a processor to a system that is CPU bound. However, adding a processor to a system that is memory bound or I/O bound will not improve performance significantly. You must scale up the system's I/O or memory subsystems to eliminate any bottlenecks before you can take advantage of another processor.

Myth: *A single-processor DG/UX system, which uses the DG/UX multiprocessor kernel, does not perform as well as a single processor system that uses a traditional kernel.*

Not true—the DG/UX kernel was designed from the start to work with both single and multiprocessor systems. The DG/UX kernel database locking mechanisms are as efficient as the traditional "no pre-emption" locking strategy.

AViiON Multiprocessor Concepts

An AViiON multiprocessor system is a *Symmetric Multiprocessing* (SMP) system. The key word here is *symmetric*, which means that the CPUs, the Job Processors, in an AViiON SMP system are seen by user programs and the DG/UX operating system as equivalent (Figure 1).

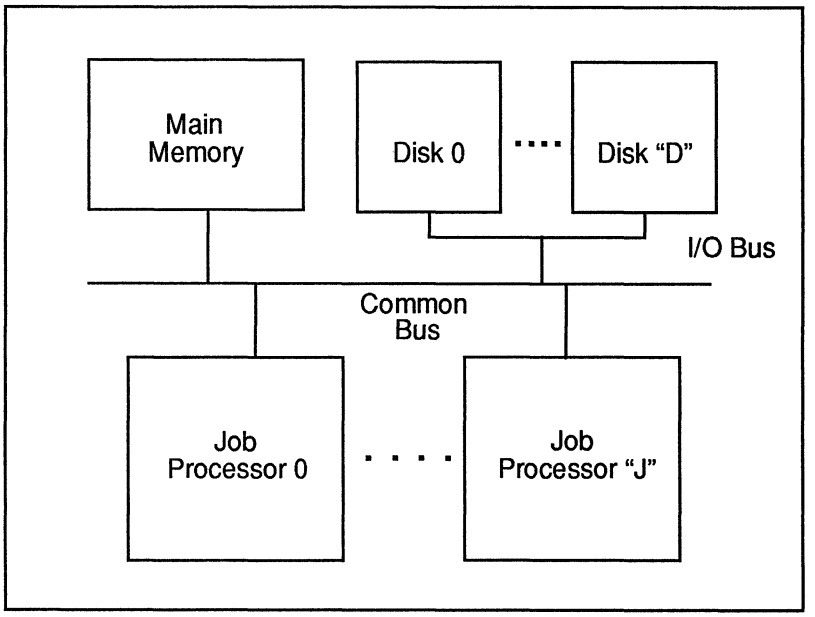


Figure 1 The AViiON Symmetric Multiprocessing System

In a multiprocessing AViiON system:

- The resources of the system's configuration, such as memory and peripheral devices, are shared by all operating system and user processes.
- A user process, or any part of a user process, can run on any Job Processor. In other words, a user process that is temporarily suspended can run on a different Job Processor when it starts to run again.
- Similarly, a kernel process can run on any Job Processor.
- Processes executing kernel code can run concurrently on multiple Job Processors.
- Any I/O operation can run on any Job Processor, and Job Processors can execute I/O operations concurrently.
- The kernel's integrity is managed by internal locking mechanisms.

The SMP design, which uses each Job Processor equally, contrasts to other multiprocessor designs, such as master/slave or master/slave hybrids. In a master/slave configuration, the operating system runs only on a "master"

processor, which controls the other processors. As the number of slave processors increases, the master processor can become a bottleneck because it cannot share its load with the other processors in the system.

In a hybrid master/slave configuration, one processor may be responsible for controlling all of the system's peripheral devices. This I/O processor becomes a potential bottleneck. In a hybrid configuration, the kernel may be able to run on different processors, but it cannot run on multiple processors concurrently.

Bottlenecks

Since we've mentioned bottlenecks in master/slave configurations, we'll make it clear that SMP configurations can also have bottlenecks. Typically, the bottlenecks occur when there is too little memory or too few disk resources to service the multiple Job Processors. In this case, a processor may be idle while processes that it could run are waiting to access memory or a disk. This resource imbalance is the issue of "scale" that we mentioned earlier.

When you configure an SMP system, you should be sure that the system provides enough disk throughput and memory capacity to ensure that processes aren't contending for external resources. In Technical Brief 012-004054 (*The DG/UX 5.4 File System*) we provide tips for tuning a file system.

Processes, Virtual Processors, and Job Processors

Processes are programs in execution. On an AViiON system, processes run on Virtual Processors (VPs). VPs are software abstractions of the computer's real, physical Job Processors (JPs). Because they are software abstractions, VPs hide the implementation details of the underlying hardware from the processes. For example, processes don't even know if they're running on a machine that has a single JP or multiple JPs.

On an SMP system, a process can run on any Job Processor.

Figure 2 shows conceptually how processes, VPs, and JPs are arranged. In this hierarchy, there are typically more processes than VPs and more VPs than JPs. Therefore, processes compete for the services of the VPs and VPs compete for the services of the JPs.

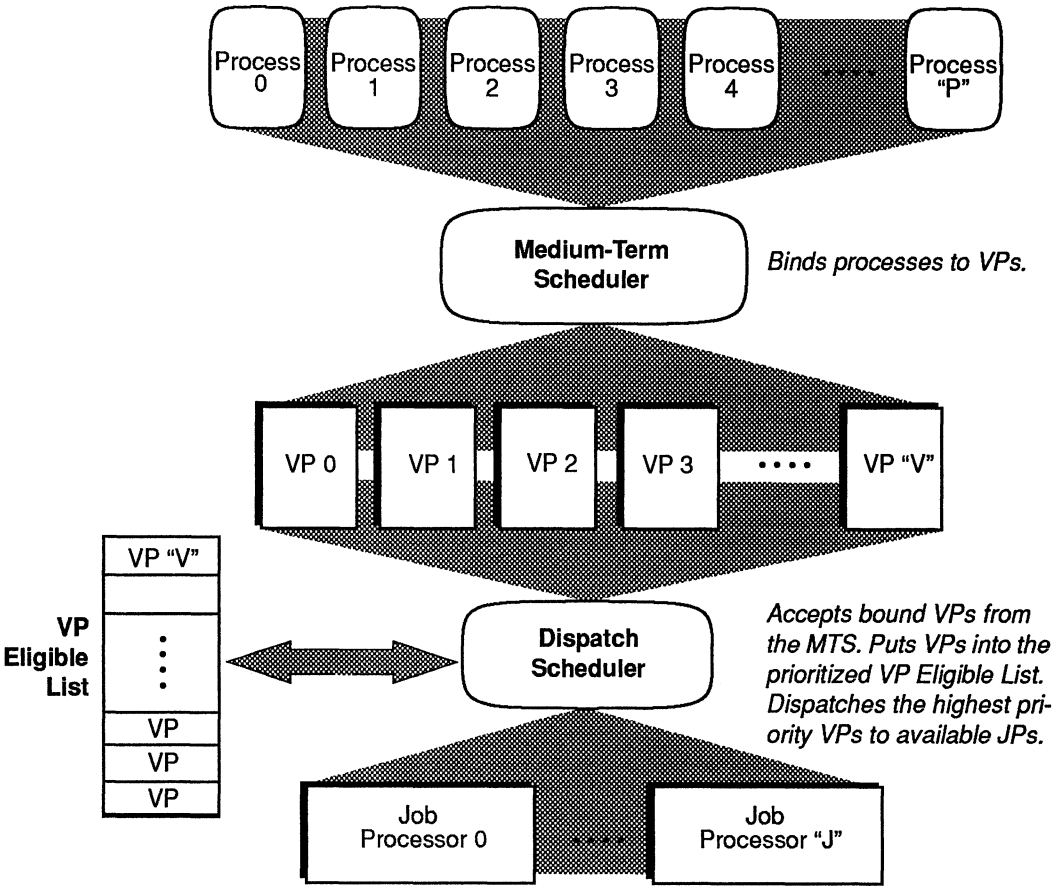


Figure 2 Scheduling Processes, Virtual Processors, and Job Processors

As shown in Figure 2, there are two levels of scheduling, which the DG/UX kernel manages. The kernel has a *Medium Term Scheduler* that it uses to assign processes to VPs and a lower-level *dispatcher* that it uses to schedule VPs onto JPs. The Medium Term Scheduler (MTS) establishes and implements timesharing scheduling policies. The MTS runs as a kernel process on its own dedicated VP. The dispatcher assigns VPs to JPs, based on the VPs' priorities. The code for the dispatch scheduler runs on each JP.

When a process is assigned to a VP, we say that it's "bound" to the VP. Only one process at a time can be bound to a VP. A VP that has a process bound to it can run on any available JP, because of the symmetry of the AViiON system's multiprocessor design.

A VP (with its process) can be removed from a JP when any of several events occur. Because the DG/UX operating system is a time-sharing system, a VP may have simply used up its allocated time slice and needs to be taken off of a JP to allow another VP to run. Other events include page faults, which occur when a process tries to access data that isn't already in memory.

When a VP can no longer run, the JP that was running the VP is available to run a different VP. Whenever a JP is free, its dispatch code says "I'm available to run a VP" and the dispatcher looks at the system-wide VP Eligible List to determine which VP to run next.

The VP Eligible List contains the status of all the system's VPs, including information about whether a VP can be run, as well as the VP's priority relative to the other VPs. A JP's dispatcher looks at this queue and loads the highest priority VP that is ready to run.

About Kernel Locks and Integrity

Designing and implementing a multiprocessor system requires an operating system that ensures that the kernel's internal data is protected. Consider the case where a process, running in kernel space, updates data in two kernel databases. If another process tries to access one of these databases, the kernel process could read or write invalid data. The same problem could occur if you are asking for your checking account balance from a teller machine while someone else in your family is making a deposit at another teller machine. Your account is temporarily "owned" by the first transaction and is released when the account's database is updated.

In most single-processor UNIX systems, the kernel's integrity is maintained primarily by a policy of *no pre-emption*. While a process runs in kernel mode (supervisor mode), the process cannot be pre-empted by another process—the kernel mode part of the process must run to completion. A process runs in kernel mode when its program makes a system call, or when the process takes a page fault.

This traditional, no pre-emption approach is not effective in a multiprocessor system. It makes no sense to have one Job Processor wait while another one completes an entire kernel-mode operation as occurs in master/slave configurations. Processes would be waiting needlessly when they could be doing work.

The DG/UX kernel's locking mechanisms ensure kernel integrity.

To solve the problem of under-utilized processors, the DG/UX operating system uses kernel database *locking* mechanisms. The operating system applies locks to critical kernel databases so that only one process at a time can access them. If a process tries to access a locked database, the process "waits on the lock" and can access the database after the first process has completed its read or write operations. That means that a process can enter kernel mode and perform any operation up to the point that it requires access to a locked database. The DG/UX kernel's locking mechanisms provide finer control of the kernel's shared resources than the no pre-emption strategy (which is really equivalent to a single, or global, lock on the kernel).

This DG/UX locking technique allows the operating system to perform operations on behalf of processes on more than one Job Processor—without corrupting system data. Consider, for example, two kernel processes: One is creating a file; the other wants to delete the same file. The kernel's locking mechanisms ensure that the operations are synchronized so that they cannot corrupt system databases.

Finally, we'll point out that this kernel database locking strategy costs little in terms of performance when used on a single-processor system. (The DG/UX operating system uses the same kernel on both single-processor and multiprocessor configurations.) Some people have wondered if there is too much overhead managing locks as compared to the traditional no pre-emption strategy. No—the DG/UX kernel's design is efficient because the kernel was designed from the start to work with both multiprocessor and single-processor systems. The DG/UX kernel provides a variety of locking mechanisms with different performance characteristics and functionality. This enabled the kernel designers to use the most efficient locks for the kernel's different databases.

SMP and Parallel Processing—Apples and Oranges

An area that deserves some attention is the difference between SMP systems and parallel processing systems. It's important to emphasize that the AViiON series of multiprocessor systems are not parallel processing systems.

A parallel processing system often depends on special programming techniques or special compilers. A parallel processing compiler automatically locates parallel paths in programs and generates code that will run concurrently on the multiple processors in a parallel system. A parallel system usually provides special-purpose, but non-standard, extensions and system calls. You can't make these special extensions invisible to the program.

On AViiON systems, the same program will run on single-processor and multiprocessor systems.

The important issue here is program portability. The software portability inherent in SMP systems is not available in parallel processing systems. When you port a *parallelized* program to a different parallel programming system, you'll probably have to modify or rewrite the code to use the target system's extensions. No standards are in place for writing portable code for different parallel systems.

In contrast, programs that adhere to the System V, BSD, or POSIX standards will run on any AViiON system. If a source program complies with one of these standards, the program will run, with minimal changes, on a single-processor AViiON system, on a multiprocessor AViiON system, or on other machines that adhere to the System V, BSD, or POSIX standards. Also, if an executable program adheres to the 88open Binary Compatibility Standard (BCS), the program will run, without changes, on any AViiON system.

Multiprocessor Performance Case Studies

We'll conclude this technical brief by looking at four examples of how multiprocessor systems provide a performance advantage over a single-processor system. In the tests, we used a new AViiON eight-processor system, and ran the same programs on one, two, four, and eight of the system's processors. We chose the examples to highlight the fact that multiprocessor performance gains are dependent on the kinds of applications that a system is running. We'll show you that compute-bound applications realize the most performance gains.

Test 1 demonstrates how a two-processor system, a four-processor system, and an eight-processor system perform at their theoretical limits of two times (2X), four times (4X), and eight times (8X) better than a one-processor system.

Test 2 shows you how two processors can actually perform over two times better than one processor, four processors over four times better, and how eight processors can perform over eight times better.

The third and fourth tests are based on practical, real world applications. In test 3, we compiled and linked a large program (the DG/UX 5.4.2 kernel). The results show the performance gains that can be achieved when you are running compute-intensive programs. In test 4, we ran a benchmark program that tests a system's performance in a multiuser, mixed job scenario.

In all of the tests, we used the same 25Mhz AViiON AV6280 series computer, running the DG/UX 5.4.2 operating system.

For the first three tests, the system was configured with 348 Mbytes of main memory and one 1 Gbyte SCSI disk. We configured the system to run with a single Job Processor (JP), then with two JPs, then with four JPs, and finally, with eight JPs. The operating system was configured with the default settings and we did no run-to-run tuning. Therefore, the machine and operating system configurations will not provide the optimum performance for a particular application or for this range of test applications.

Test 1—2X, 4X, and 8X Performance Gains

In this test, we ran a totally compute-bound application—an application that requires no I/O—on one, two, four, and eight processors. Although this isn't a "real" application, the test demonstrates that the 2X, 4X, and 8X scaling when running a compute-bound application.

This test application starts four processes and each process immediately forks another child process (for a total of eight processes). Each of the eight processes then counts to 100,000,000.

Figure 3 shows the time that it took to run the test on four different processor configurations. The right-hand column in the figure shows the speed-improvement ratios of the two, four, and eight processors versus one processor.

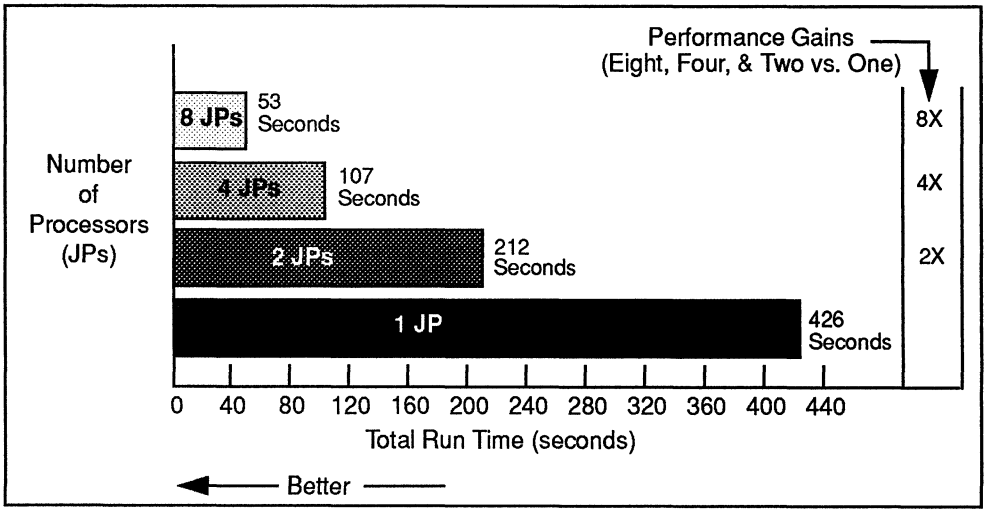


Figure 3 Compute-Bound Process Performance

Test 2—Performance Gains Greater Than 2X, 4X, and 8X

The second test shows that some applications can perform more than 2X better on two processors than on one processor, more than 4X faster on four processors, and more than 8X faster on eight processors.

Our test program passes data back and forth between two processes, using two pipes. The two programs are synchronized; that is, the second program processes the data from the first program exactly as fast as the data is received.

We first ran one copy of the application (two cooperating processes) on one, two, four, and eight processors. Then, on the same mix of processors, we ran two and four copies of the application (Figure 4).

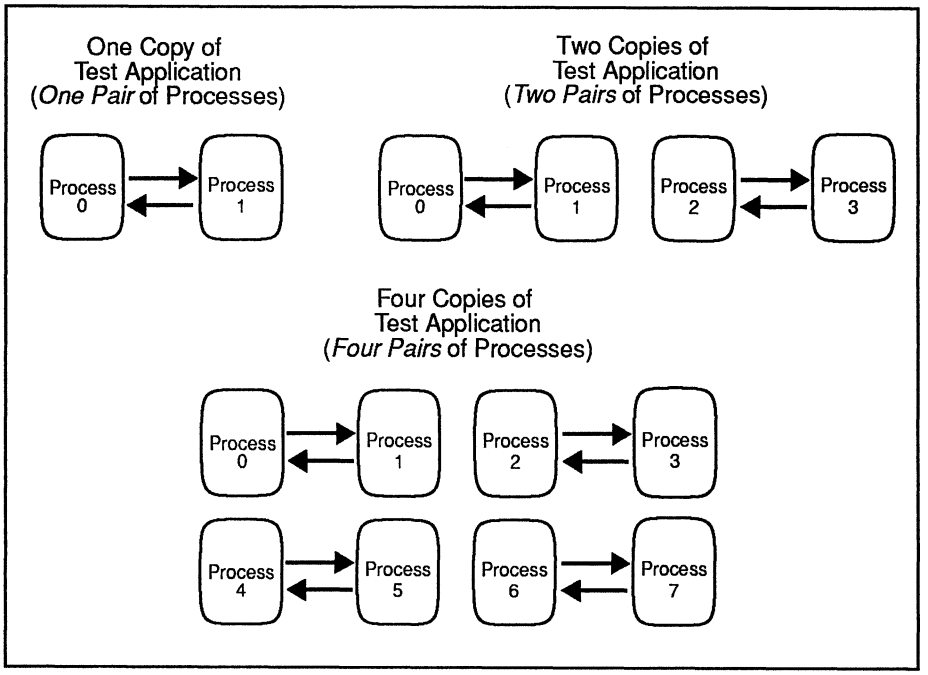


Figure 4 Cooperating Processes

When two copies of the application are running, there are four processes (two *independent pairs* of cooperating processes). When four copies of the application are running, there are eight processes (four pairs of independent processes).

Figure 5 shows the total number of operations/second when one, two, and four copies of the application were running on one, two, four, and eight processors. By total number of operations, we mean the number of data transfers between all of the process-pairs in a test. The performance gains shown in the right-hand column are calculated by dividing multiprocessor throughput by the throughput of a single processor.

When running one copy of the application (the bottom set of bars in the chart), two processors ran 2.52X more operations than one processor. Note that running one copy of the test program on a four- or eight-processor system provides little performance improvement over a two- processor system. That's because there are only two processes, so two processors (in a four processor system) are essentially idle.

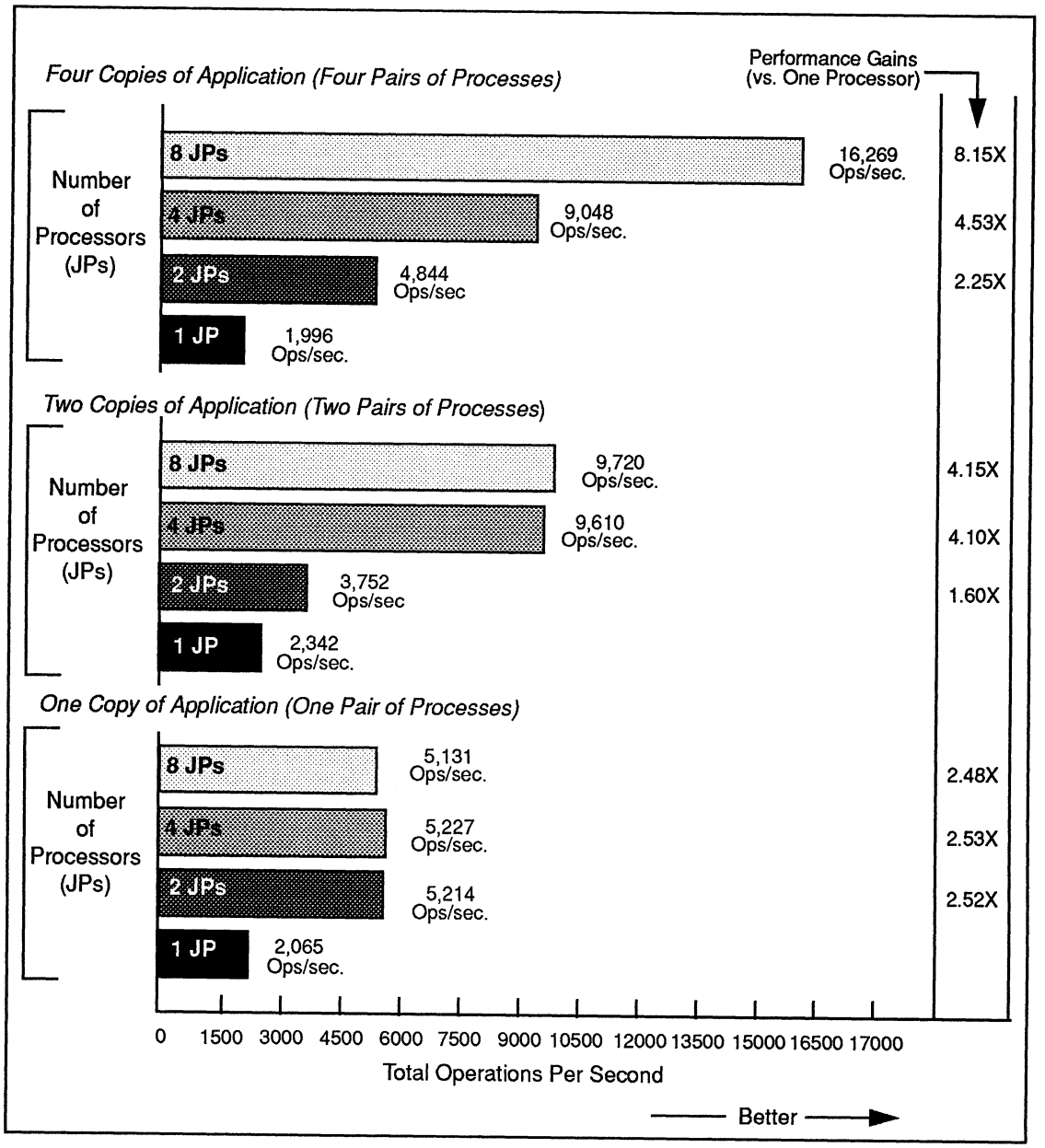


Figure 5 Exceeding Theoretical Multiprocessor Performance

In the one-copy test, a two-processor system completes 2.52X more operations than a one-processor system. Note that the four-processor system completes 6.49X more operations than one processor. The eight-processor system completes 8.15X more operations than a one-processor system. In this case, any of the eight processors can run any one of the eight processes.

How can a multiprocessor system have performance gains that are greater than the number of processors? Because of process scheduling. When you run this kind of synchronized application on one processor, the processor has to switch back and forth between the two programs as one program finishes its work and passes data to the other program. Therefore, the total time to run the application includes the time to process the data plus the time to schedule the processes that run the two programs.

When a single copy of the application runs on a two-processor system, the first program runs on JP0 and the second program runs on JP1. The programs can proceed in parallel—without the overhead of process scheduling. With no process scheduling, the effective net performance gain is more than 2.5X when the application is run on a two-processor system. Similar performance gains are seen when two copies of the application run on a four-processor system. When four copies of the application run on an eight-processor system, the net performance gain is more than 8X.

Process-scheduling overhead is the reason that there are fewer operations/second for two copies of the application on two processors (3,752) versus when two copies of the application runs on one processor (5,214). When two copies of the application run on two processors, there are four processes competing for the services of the two processors.

Even though this is a theoretical test, it has real-world implications when an application uses pipes between programs or when an application uses lots of operating system resources.

An example of pipelined programs taking advantage of multiprocessors is when you format text with the **troff** text processing program. You may first run the **tbl** program on the source text and pipe the output of **tbl** to **troff** and then to a post-processor (such as, **tbl | troff | pspost**). These pipelined processes will run faster on two processors, four processors, and faster yet on eight processors—not only because the processes can run in parallel, but because fewer process switches are needed.

A program that is making many operating system calls can run faster on a *Symmetric* Multiprocessor System. On a true SMP, such as AViiON systems, the operating system functions can run on any available processor while the application's code continues to run on another processor (or processors).

Test 3—Multiprocessors' Effects on a Large Compilation

In the third test, we compiled the more than 1,400 source-code files of the DG/UX kernel. The kernel-build compilations require significant I/O and computational resources. There are approximately 37 minutes of serial setup, then the independent compilation of the individual source-code files can proceed. On symmetric multiprocessor systems, the compilations can be performed in parallel.

We compiled the source-code files with one processor, two processors, four processors, and eight processors. Figure 6 shows the time that it took to run the kernel-build compilations on the four configurations.

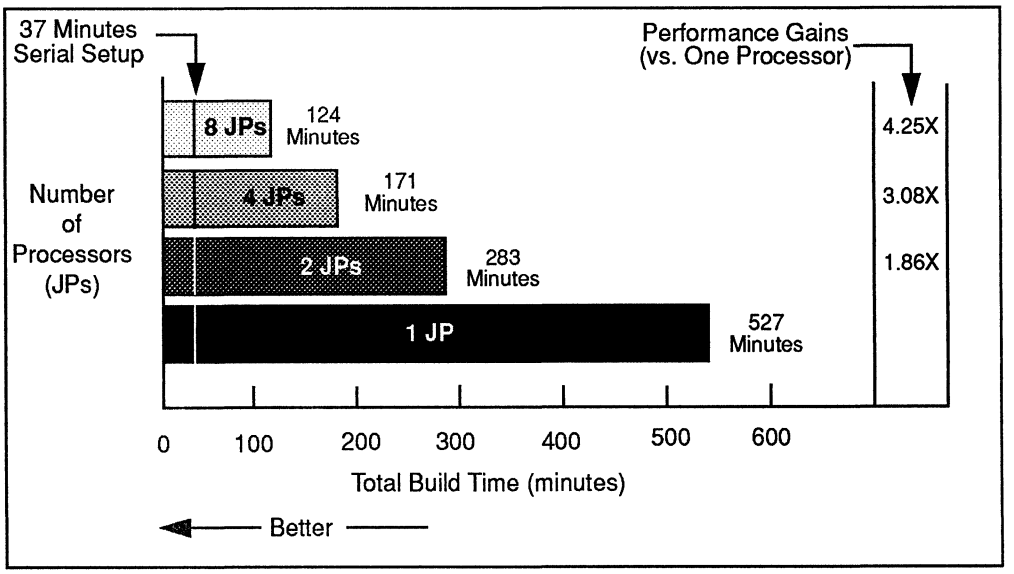


Figure 6 Kernel Build Times

There are two things to consider when you look at the performance gains of Figure 6.

- Only a small performance advantage is provided by multiple processors during the 37 minutes of serial setup. During the setup operations, only one process runs at a time. The shortest time in which this test can run is limited by this serial component; regardless of how many processors are in the system. This issue is discussed in more detail in the *Taking Advantage of Symmetric Multiprocessor Systems* technical brief (012-004177).

  Comparing the times of just the parallel parts of the test (without the serial setup time) show performance gains of 1.99X, 3.65X, and 5.63X for two, four, and eight processors.

- This is a real-world application that is not compute bound; it requires I/O operations to read the individual source-code files from disk. Our single-disk test system was not tuned to provide the optimum combination of disk resources for this test. The potential for I/O bottlenecks caused by using a single disk for this kind of application becomes more pronounced as the number of processors increases. As more processors are added to a system, more attention must be paid to balancing the system's I/O throughput with its processing capabilities.

Test 4—Multiprocessors' Effects on a Multiuser Benchmark

For the last test, we ran a standard multiuser benchmark that simulates multiple users, with each user running a mix of basic kernel activities. For this test, we increased the maximum number of processes to 1,024 from the default of 256. We added four SCSI disks, which were used as temporary (**tmp**) space.

Figure 7 shows the results of the benchmark tests, with the number of users plotted against the number of seconds to run the benchmark.

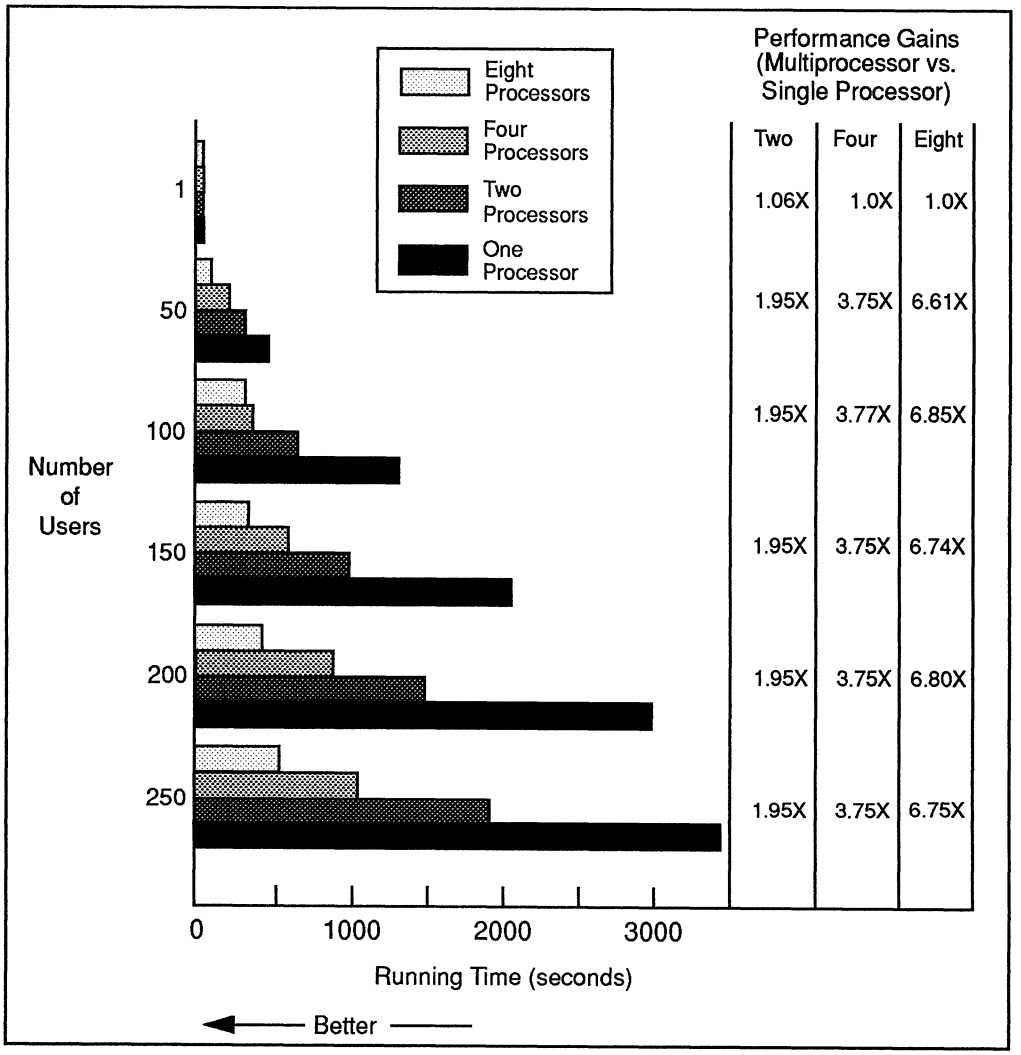


Figure 7 Multiuser Benchmark on One, Two, Four, and Eight Processors

As you can see, the two-processor configuration provides an average performance increase of 1.95X as soon as more than one user uses the system. With four processors, the increase jumps to an average of 3.75X.

With eight processors, the increase shows an average of 6.7X. The maximum multiplier for the two-processor system is 1.95X; for the four-processor system the maximum multiplier is 3.77X, and for the eight-processor system the maximum multiplier is 6.85X.

The better performance of the four and eight processors against two processors for a single user shows an example of operating system processes running on any available processor.

For More Information

Among the articles that discuss multiprocessor systems in more detail are:

DG/UX™ Technical Brief: *Taking Advantage of Symmetric Multiprocessor Systems* (012-004177), June 17, 1992, Data General Corporation

Multiprocessor Aspects of the DG/UX Kernel
USENIX Conference Proceedings—Winter 1989
Michael H. Kelley, Data General Corporation

Design of Tightly-Coupled Multiprocessing Programming
IBM Systems Journal—Vol. 13 No.1 (1974)
J.S. Arnold, D.P. Casey, and R.H. McKinstry

Multiprocessor Unix Operating Systems
AT&T Bell Laboratories Technical Journal—Vol. 63 No. 8 (October 1984)
M.J. Bach, S.J. Buroff

An Experimental Symmetric Multiprocessor Ultrix Kernel
USENIX Conference Proceedings—Winter 1988
Hamilton, Graham, and Daniel S. Conde