Multi-Threading in IDL

Copyright © 2001-2007 ITT Visual Information Solutions
All Rights Reserved
http://www.ittvis.com/

IDL® is a registered trademark of ITT Visual Information Solutions for the computer software described herein and its associated documentation. All other product names and/or logos are trademarks of their respective owners.

Table of Contents

I. Introduction ____ 3

Introduction to Multi-Threading in IDL ____ 3
What is Multi-Threading? ____ 3
Why Apply Multi-Threading to IDL? ____ 5
A Related Concept: Distributed Processing ____ 5

II. The Thread Pool ____ 7

Design and Implementation of the Thread Pool ____ 7
The IDL Thread Pool ____ 7
Why The Thread Pool Beats Alternatives ____ 9
Re-Architecture of IDL ____ 9
Automatically Parallelizing Compilers ____ 10
Threading at the User Level ____ 11

III. Performance of the Thread Pool ____ 13

Your Mileage May Vary ____ 13
Amdahl's Law ____ 13
Interpreting TIME_THREAD Plots ____ 15
Example System Results ____ 17
Platform Comparisons ____ 31

IV. Conclusions ____ 36

Summary ____ 36
Advice (So What Should I Buy?) ____ 37

V. Appendices ____ 39

Appendix A: IDL Routines that use the Thread Pool ____ 39
Appendix B: Additional Benchmark Results ____ 41
Appendix C: Glossary ____ 53

Introduction

Introduction to Multi-Threading in IDL

ITT-VIS has added support for using threads internally in IDL to accelerate specific numerical computations on multi-processor systems. Multi-processor capable hardware has finally become cheap and widely available. Most operating systems have support for SMP (Symmetric Multi-Processing), and IDL users are beginning to own such hardware. In the future, it is reasonable to imagine that most machines will have multiple CPUs.

The multi-threading capability, first added in IDL 5.5, applies to binary and unary operators, many core mathematical functions, and a number of image processing, array manipulation and type conversion routines. Although performance results will vary, the execution time of these computations can be significantly reduced on systems with multiple processors. The ability to exploit multiple CPUs will become very important in coming years, and the list of threaded routines is expected to grow with each release of IDL.

IDL users should be aware that ITT-VIS offers a Global Services Group (GSG) that can be hired to help optimize user-written code or to parallelize specific algorithms beyond those that use the thread pool.

The interface for controlling the IDL thread pool is simple, allowing immediate and measurable benefits with little effort. In addition, the IDL thread pool is safe and transparent on platforms that are unable to support threading. Those platforms that can benefit will use threads, and those that cannot will continue to produce correct results using a single thread, and with the same level of performance as previous versions of IDL.

This chapter provides background and motivation for IDL's multi-threading capability.

What is Multi-Threading?

The concept of multi-threading involves an operating system that is multi-thread capable allowing programs to split tasks between multiple execution threads. On a machine with multiple processors, these threads can execute concurrently, potentially speeding up the task significantly. Mathematical computations on large amounts of scientific data can be quite intensive and are ideal candidates for threading on systems with multiple CPUs.

The most common type of program is the single-threaded program. IDL has traditionally been single-threaded. When the program runs, this single thread starts at the main() function in the program, and runs until it either exits, or performs an illegal operation and is killed by the operating system. Since it is the only thread, it knows that anything that happens in this program is caused solely by it. Most modern operating systems time slice between various programs, so at any given time, a single thread is either running or sleeping. There are two reasons why it may be sleeping:

- It is waiting for a needed, but currently unavailable resource (e.g. memory, data (input, output)…).
- The operating system is letting some other program run.

This time slicing, which is usually preemptive multitasking, happens so quickly that the end user is fooled into thinking that everything is running simultaneously.

To move from single-threaded to multi-threaded (MT) programs requires a small conceptual generalization of the above. Instead of having only a single thread, we allow more than one thread in a single process. Each thread has its own program counter and stack, and is free to run, unimpeded by any other thread in the same program. All threads in the same program share any other resources, including code, data, and open files. The operating system still schedules which threads run and when, but instead of scheduling by process, it now schedules the individual threads within a process. If your system has a single CPU, preemptive multitasking still gives the illusion that more than one thing is going on simultaneously. If the system has more than one CPU, then more than one thread can be running on the system at a given time. It is even possible that more than one thread within a given program will run simultaneously. This is actual simultaneous execution, not the mere illusion of it as with a uniprocessor.

It is important to realize that the software concept of threading is an abstraction provided by the operating system, and it is available whether or not the underlying hardware has multi-processing (MP) capabilities. It is reasonable to run a MT program on a uniprocessor, unless your program requires actual simultaneous execution of multiple threads to work properly. For example, a program might use a thread to wait for incoming data from a slow source, while other threads manage the user interface and perform other tasks. Multi-processor hardware is not necessary for such a program. In contrast, if you are using threads to speed up a numerical computation, you will require actual MP hardware to see any benefit. On a uniprocessor, this program will work harder (MT code adds overhead) and will take essentially the same amount of time to complete as a single-threaded version. Common sense suggests that threading does not make a uniprocessor able to compute any faster.

Why Apply Multi-Threading to IDL?

Simply put, ITT-VIS has implemented multi-threading in IDL in order to allow users to harness additional CPUs to do more work in less time. Scientific data sets continue to grow in size faster than computers can process them. Multi-processors offer one way to handle larger problems.

Multi-processor hardware and Symmetric Multi-Processing (SMP) have become cheap and easily available. There are some powerful trends driving this change:

1. As transistor densities on processor chips increase with each generation, there is room for replicated processing units.

2. At any given point in time, the cost of the second most powerful CPU in production is much lower than the most powerful CPU. It makes sense that if you can harness multiple cheap, but only slightly less powerful, CPUs, you can do more work for less money.

3. There are physical limits that govern how fast a single CPU can possibly go, and we expect to hit those limits within a few (10-20, max) years. Once we hit this limit, the only way to increase computing power may be to add CPUs.

The development of SMP systems has been driven not by the need to run multi-threaded programs, but by a need to increase throughput on servers that run multiple single-threaded programs simultaneously (e.g., to serve files, mail and printing). Economies of scale allow computer vendors to apply this technology to desktop machines. It is becoming common for individuals to have such machines, and it appears that this trend will continue.

A Related Concept: Distributed Processing

Multi-threading is not the same distributed processing. While distributed processing, sometimes called parallel processing, and multi-threading are both techniques for achieving parallelism (and can be used in combination), they are fundamentally different. Multi-threading attacks the problem of doing more work faster at the micro level, while distributed processing attacks this at the macro level.

Multi-threading is a way to let programs do more than one thing at a time, implemented within a single program, and running on a single system. Multi-threading requires special support from the implementation language (or its support libraries) and the underlying operating system. Over the last 20 years, research in this area has matured to the point where stable, reasonably portable, system interfaces exist for writing threaded programs. Standardization of these interfaces ensures that we can use them knowing that they will have a long life and provide a stable basis for our work.

Distributed processing is a way for multiple programs, usually running on different systems connected over a fast local network, to cooperate in solving a single, usually large, problem. Distributed processing does not usually require any special language or OS support, but it requires a support framework (usually in a library that the programs can link to) that oversees the process of managing the communication of tasks to end nodes, the communication between them, and the pulling together of the final results. At this time, there are several different approaches to solving these sorts of problems, and standardization has not yet occurred. There are, however, some clear favorites, such as PVM (Parallel Virtual Machine) and MPI (Message Passing Interface).

Distributed processing does not require direct internal support within IDL. The current implementation of IDL is not preventing its use within a parallel processing framework. In fact, IDL users have already had success with distributed processing with IDL by taking advantage of larger existing frameworks such as PVM or MPI.

An example of software technology that provides a cluster (distributed) computing solution for IDL is the FastDL product available from Tech-X Corporation:

http://www.txcorp.com/products/FastDL/

The Thread Pool

Design and Implementation of the Thread Pool

Starting with version 5.5, IDL has the ability to use a thread pool to divide numerical computations among multiple CPUs. This multi-threading capability applies to arithmetic operators and mathematical functions, along with many image processing, array manipulation, and type conversion routines. Users can control the IDL thread pool to their advantage, through a simple interface that allows immediate and measurable benefits with very little effort.

ITT-VIS carefully considered other implementation options and chose the thread pool for a number of reasons. The IDL thread pool is a convenient implementation that allows IDL users to take advantage of multiple processors to speed numerical computations today--without having to wait for ITT-VIS to complete a resource-intensive and time-consuming re-architecture of IDL. With the design of the IDL thread pool, IDL maintains its single-threaded organization and uses threads only in focused and tightly controlled ways that will not lead to statistical bugs, locking problems, or other threading pitfalls.

This chapter discusses the design of the IDL thread pool and explains why the implementation beats the alternatives.

The IDL Thread Pool

The IDL thread pool provides a robust and simple mechanism for overlapping numerical computations to achieve potentially significant performance gains. It consists of a group of threads (excluding the main thread) that are created when IDL starts. On a system that supports N CPUs, the thread pool default is to have N-1 threads in the pool. Counting the main thread, this gives you one thread for each processor. While the thread pool sleeps, the main thread runs IDL much as it always has, as a single threaded application. When not involved in a calculation, the threads in the thread pool are inactive and consume little in the way of system resources. When IDL reaches a computation that can use the thread pool and which can benefit from parallel execution, the main thread assigns the N-1 threads of the thread pool work to do, and wakes them to run in parallel with the main thread. Once the helper threads finish their tasks and go back to sleep, the main thread continues. To the user, this looks and feels like a single threaded application that simply seems to run faster.

The initial use of the IDL thread pool has been to thread IDL's array-oriented arithmetic operators and mathematical functions, along with many image processing,

array manipulation, and type conversion routines. The IDL thread pool has also been used to replace the existing implementation of multi-threading for volume rendering. For a complete listing of threaded routines, see the IDL Reference Guide.

The IDL thread pool is easy to use, providing an immediate and measurable benefit to the IDL user without requiring a special effort. IDL automatically determines when and how to employ multi-threading. When IDL encounters eligible computations, it determines whether or not to use the IDL thread pool to carry them out. This is based on the availability of multiple CPUs in the current system as well as on the number of data elements in the input array. The latter criterion is somewhat heuristic because IDL cannot know all of the information necessary to determine the effect multi-threading will have on performance. If a computation involves too few elements, the overhead involved in splitting a problem between threads may exceed the gain. If a computation involves too many elements for the system memory, the virtual memory system will be activated (paging), and threads may begin competing for access to memory. Both situations could result in poorer, not better, performance relative to the single-threaded alternative. Using the number of elements in the input array as a factor in deciding whether to employ the thread pool in a given computation is a good rule of thumb. As with all rules of thumb, there are situations in which it applies less well. There are also other reasons threading may not be desired. For instance, out of courtesy to other users on a multi-user system, or when the rounding of finite precision floating point types may produce different (although equally correct) results in algorithms that are sensitive to the order of operations. For all of these reasons, the IDL user is provided with a simple interface to control the parameters IDL uses when deciding to employ multi-threading:

- A read-only system variable named !CPU that reflects the current state of IDL's use of processor features. !CPU is initialized by IDL at startup with default values for the number of CPUs (threads) to use, as well as the minimum and maximum number of data elements. If you have more than one processor on your system, if your desired computation is able to use the IDL thread pool, and if the number of data elements in your computation falls into the allowed range (neither too few, nor too many), then IDL will employ the thread pool in that calculation.

- A system procedure named CPU which is used to alter the state of !CPU. With the CPU procedure, the user can control the minimum and maximum number of data elements for which IDL will use the thread pool, and the number of threads to use.

- Standard thread pool keywords accepted by all system routines that use the IDL thread pool. These keywords are used to override the defaults established by !CPU on a per-call basis.

The IDL thread pool is safe and transparent on platforms that are unable to support threading. Those platforms that can benefit will use threads, and those that cannot will continue to produce correct results using a single thread, and with the same level of performance as previous versions of IDL.

Why The Thread Pool Beats Alternatives

The IDL thread pool is a convenient way to apply multiple threads without re-architecting IDL's language and interpreter. It can potentially be applied, internally, to any task that does not call a non-reentrant function. While this limits the scope of possibilities, numerical computations fit this requirement well. Many IDL users are grappling with ever-growing data and can benefit from the ability to solve larger problems faster with multi-processors.

To enable the users of IDL to take advantage of powerful MP hardware in solving large computational problems, ITT-VIS had several options. IDL could have been completely re-architected to make the language and interpreter reentrant and thread-safe. Perhaps an auto-parallelizing compiler could have been employed to ease this task. Threads could have been exposed at the user level, instead of internally. ITT-VIS considered these options carefully and chose the IDL thread pool implementation for a number of reasons.

Re-Architecture of IDL

Since its inception more than twenty years ago, IDL has existed as a single-threaded program. The implementation of the IDL thread pool does not change this fact. With the thread pool implementation, IDL's internal use of threads is well-contained, allowing IDL to maintain its single-threaded organization. Retrofitting a single-threaded program to use threads is a resource-intensive endeavor, and the end-result is likely to be error-prone and may suffer poorer performance overall. Developers who take on the task face the following challenges:

- In a MT program, there can be more than one thread of execution simultaneously running within the same program and address space. Programs use this to produce all sorts of desirable effects. On the negative side, in a MT program, a given thread cannot assume that it is the only cause of change within the program. Threads must be careful not to change data at times when other threads are accessing it, or problems that are unpredictable and difficult to fix will result.

- Single-threaded code usually makes many implicit assumptions that simply do not hold in threaded code. Single-threaded code is rarely designed to be reentrant or with careful thought to how locking might be used to control access to critical sections. (Locking keeps multiple threads from colliding.) It is easy to understand why the authors of single-threaded code may not address these issues, as the solutions usually require additional time to design and implement, and often require some sacrifice in simplicity or performance.

- UNIX programs of any complexity usually need to handle signals. In a single-threaded program, signals are delivered to the program as they occur. A process has a signal mask that controls if it is able to receive a given signal. Signals not currently allowed are quietly remembered by the system, and will be delivered if the signal mask should change to allow it. In a multi-threaded program, the situation is predictably more complex. Each thread has its own signal mask. Synchronous signals (such as division by zero) are always delivered to the thread that caused it, but asynchronous signals can be delivered to any thread that has a compatible signal mask. If more than one thread is currently allowing a given asynchronous signal, the system is free to deliver it to any of them.

- A single-threaded program has a single stack. Usually, the stack starts at the top of the address space, and grows down, while the process dynamically allocated memory (the heap) starts at a low address just beyond the program code and grows up. As long and these two memory segments do not collide and the system has sufficient virtual memory, there is no conflict and both segments can get very large. This means that deeply recursive algorithms can be supported in single-threaded programs. Multi-threaded programs require a separate stack for each thread. This means that the maximum stack size must be determined by the system when the thread is created, and it must be significantly smaller than that allowed in single-threaded programs. This can cause recursive algorithms that work fine in a single-threaded environment to overflow the stack in a multi-threaded one.

- Large software programs can contain a lot of code from third party sources (IDL definitely does). Such code is rarely thread-safe, and the idea of making it thread-safe is not realistic. Instead, it must be used in a single-threaded manner (for example, by putting a lock around all use of it to ensure only one thread at a time can access it).

Implementing the IDL thread pool allows ITT-VIS to provide the benefit of threaded computations quickly, without tying up resources in pursuing a threaded organization that could be error prone or result in poorer performance overall.

Automatically Parallelizing Compilers

There are some C compilers that can automatically examine input code and generate object code that uses threads to speed a computation. With a flick of a compiler switch, one can parallelize an entire program in a few minutes. Compare that with the approach of carefully writing threaded code by hand, and it's easy to see the appeal. Why didn't ITT-VIS use this approach to thread IDL, instead of implementing the thread pool?

It turns out that there are many serious problems with this approach:

- The automated approach to parallelizing is only able to detect fairly simple opportunities for using threads. To do a good job requires algorithmic change, and therefore, human intelligence. It is not a magic bullet.

- Auto-parallelizing compilers usually require you to add specially formatted comments or #pragma statements to the code to get the desired parallelization. These comments and pragma differ from compiler to compiler, which will create an unmanageable mess in the internal IDL code.

- IDL is a cross platform product, but the auto-parallelizing compilers are not uniformly available. On the platforms for which they are available, they produce code with varying degrees of improvement.

- IDL is not merely a standalone program. It is also a library, to which users link their own code. It is a long-lived program, and it is important that its performance from release to release be predictable and stable. For these reasons, the quality and consistency of the compiler used to build it is very important. This is why IDL is generally built using the standard compiler for each platform. These standard compilers are optimized by the vendor to get the best performance from the system. They are, however, not auto-parallelizing. Auto-parallelizing compilers are niche products, sold to a small number of people, and often produced by third party companies. The quality of such compilers is suspect compared to that of the system compilers, which see much wider use.

- Sometimes it is better not to use threads even when they are available. The size of the problem may not be sufficient to cover the overhead, or resources may not be available. Automatically parallelized code is difficult to tune in this way, and is generally an all or nothing proposition with little end user control.

- Some auto-parallelizing compilers place unacceptable restrictions on the resulting program. For example, they may not be able to be used with programs that use explicit threads for any reason. Since IDL has other needs for threads, and needs to be able to link to other third party code (that may or may not be threaded), this is unworkable.

In summary, auto-parallelizing compilers are ideal for simple standalone programs dedicated to a single task on a single machine. They are not suited to large, complex, portable programs like IDL that are long-lived and dependent on third-party code. Explicitly coding the IDL thread pool offers better cross-platform consistency while allowing IDL to continue to use the standard system compilers. It gives ITT-VIS and IDL users better control over the performance of threaded algorithms. In short, it ensures a more long-lived and stable solution.

Threading at the User Level

ITT-VIS considered a third option when implementing multi-threading: exposing threads directly to the IDL user through the language itself. The implementation could conceivably involve an IDL function that creates threads. When a thread is started, it begins running an IDL program that you specified to the creation function. The thread exits when it returns from the startup function. Mechanisms would have

to exist for communication, locking, forcing threads to exit, and recovering their exit values. It is easy to think of things one could do with this:

- Instead of multiplexing widget events together using XMANAGER, you could start one thread for each widget application to handle the events for that application, having it exit when the application is done. The main thread remains free.
- Threads could wait for slow I/O without making the main IDL thread block.
- User interface code could use threads to perform computation while the user interface remains responsive.

This alternative approach may seem easy or even elegant at first. However, most examples of how a user-visible threading API in IDL might be used have single-threaded alternatives that are not very difficult, or can be addressed better by using threads under the surface.

Additional problems introduced by user-visible threading include:

1. There are many fundamental organizational questions that arise with no obvious answers. For instance, if a user types an interrupt (^C), does it go to all the threads, or just one of them? If only one, which one? Does the .CONTINUE executive command cause all threads to resume, or just the one you sent it to? Is there a separate input window for each thread, or do they share a common input window? How do you keep them straight? The existing organization is easier and simpler.
2. IDL is intended to be simple to learn and to use. IDL users are often scientists, not computer scientists, who find IDL easier and better geared to their needs than more traditional languages (like Fortran, C/C++, Java, etc). Threading at the user level brings a high degree of complexity and introduces an entirely new type of bug (statistical bugs due to the order in which threads run) that is very difficult to understand or fix.

Implementing an internal thread pool instead of exposing threads at the IDL user level allows IDL users to benefit from the application of multi-threading to time-intensive tasks such as numerical computation without encountering undue complexity. It also avoids the resource-intensive task of re-architecting a large, complex and long-lived single-threaded program, allowing ITT-VIS to continue to concentrate on making other enhancements to IDL that benefit a broad set of users.

IDL users should be aware that ITT-VIS offers a Global Services Group (GSG) that can be hired to help optimize user-written code or to parallelize specific algorithms beyond those that use the thread pool.

Performance of the Thread Pool

Your Mileage May Vary

This chapter presents the results of IDL thread pool benchmark tests on several platforms. The measurements were made using an IDL program called TIME_THREAD, located in the lib directory of the IDL distribution. TIME_THREAD performs two separate simple floating point computations: B=A+A and B=SQRT(A). By varying the number of elements in A and the number of threads used in the computation, we can see the effect of data size and the number of CPUs on performance. As noted in the discussion, threading performance depends on many factors, including minute-to-minute operating system behavior that results in statistical variability between runs. Therefore, these benchmarks apply only to the systems shown here. The clear conclusion to be drawn from these benchmarks is that threading performance can be surprising, and there is no substitute for careful testing. See the appendix for benchmarks of additional computations.

Amdahl's Law

The discussion in this section is a simplified version of a highly recommended discussion of multi-processor programming from SGI's Web site.

Amdahl's law explains in a formal manner something that seems like simple common sense: A given program consists of parts that have the potential to be overlapped in time with each other (the parallel part P) and parts that cannot be overlapped with any other part (the serial part S). The parallel part is usually expressed as a number between 0.0 and 1.0. Amdahl's Law states that for a program with a parallel part P, the speedup that one can expect by applying N CPUs to the job can be expressed as:

$$Speedup(n) = \frac{1}{((p/n)+(1-p))}$$

As stated in the SGI document referenced above:

> *There are always parts of a program that you cannot make parallel. These are the parts that must run serially. For example, consider the DO-loop. Some amount of code is devoted to setting up the loop, allocating the work between CPUs. Then comes the parallel part, with all CPUs running concurrently. At the end of the loop is more housekeeping that must be done serially; for example, if n does not divide MAX evenly, one CPU must execute the few iterations that are left over. The serial parts of the program cannot be speeded up by concurrency.*

It follows that the application of additional CPUs to such a program can only speed up the parallel part. Hence, it makes little or no sense to apply multithreading to a program that is dominated by its serial part. Understanding Amdahl's law is critical to making accurate judgments about the benefit that is possible from the application of threading to a given problem. Many algorithms have a very small P, and threading cannot help such algorithms.

One thing that Amdahl's Law does not help us do is to predict how much of a given program is parallelizable (i.e. to determine the value of P). There are several reasons why this is not practical:

- Most useful programs are highly complex, and the serial and parallelizable parts are not easily distinguishable.
- The compiler used, and the machine code thus generated, can have an effect on parallelizability that is not evident by reading code.
- The SMP capabilities of the operating system have an effect on the potential P, but the details of this are not visible to the application programmer and cannot be inferred by the system call interface.
- The design and construction of the underlying hardware is a very important factor in determining the potential value of P, but this information is difficult to apply.

So Amdahl's Law does not give us the ability to determine what value of P a program is theoretically capable of. Instead of a theoretical basis for such reasoning, we are forced to rely on experience and anecdotal evidence to decide if we are doing a good job or not.

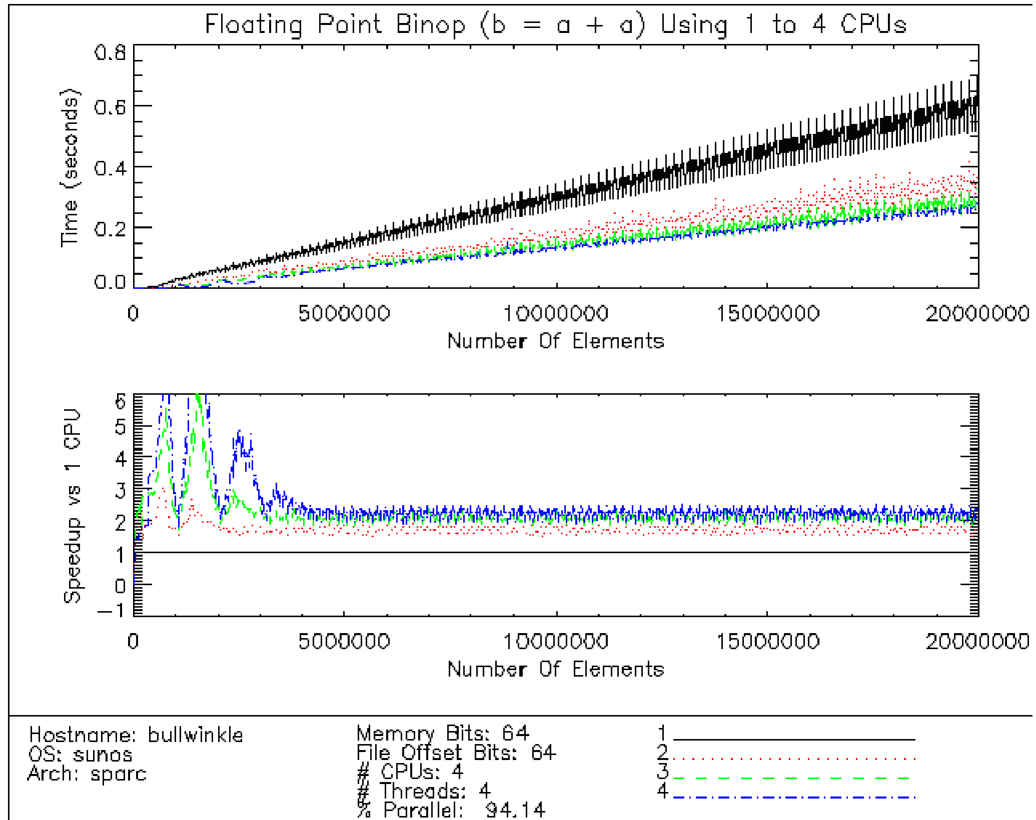
Although Amdahl's law does not allow us to determine the theoretical value of P, it does give us the necessary framework to derive the actual value of P exhibited by a program. If you run the same program twice, using n CPUs for the first run, and m for the second, the following variation of Amdahl's Law gives the actual value of P for that program:

$$p = \frac{Speedup(n) - Speedup(m)}{(1-1/n) * Speedup(n) - (1-1/m) * Speedup(m)}$$

The TIME_THREAD procedure uses this formula to calculate the value of P shown in the plots contained in this chapter.

Interpreting TIME_THREAD Plots

The following plot, which took a couple of hours to produce, shows a very strenuous TIME_THREAD run:



bullwinkle: SunOS 5.8 (Solaris 8, SPARC Ultra 80), 4 X 450MHz SPARC CPUs, 4GB memory

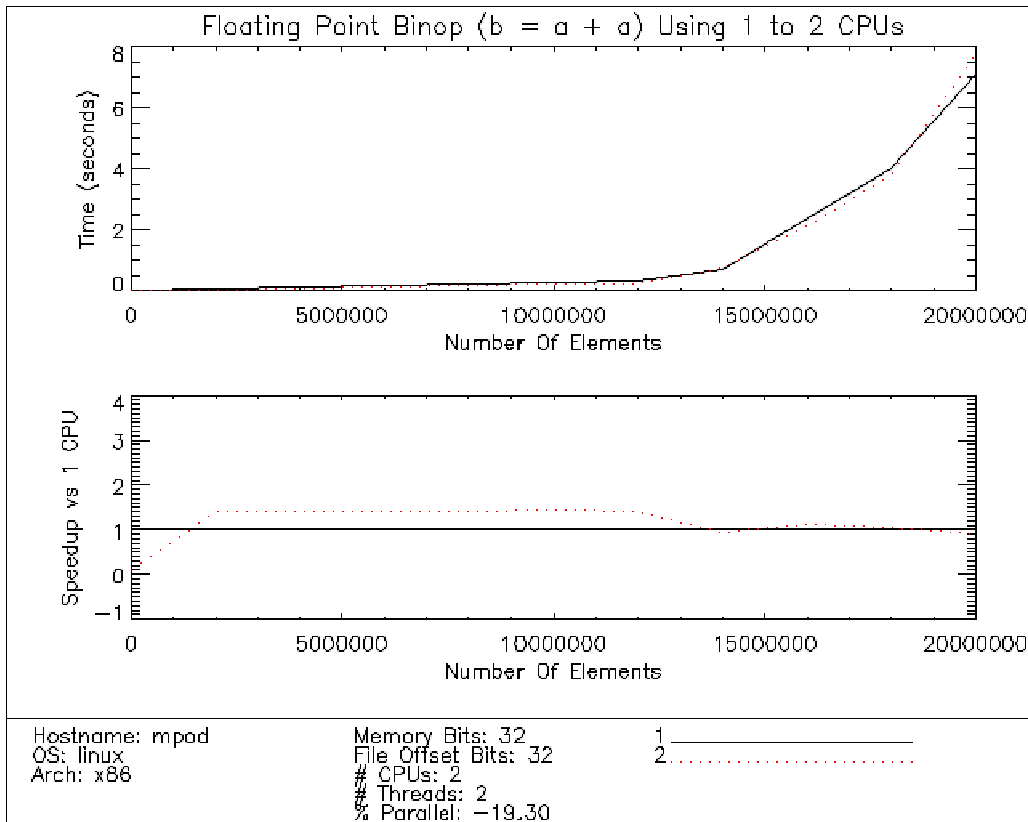
The computation was performed at vector lengths ranging from 4 to 20 million elements (16 to 80 million bytes). Between each run, the vector was lengthened by 25000 elements. At each length, the computation was run on all the possible number of physical CPUs (from 1 to 4 CPUs in this case). Finally, each run on a given number of CPUs was run 5 times, and the median time used for the result in order to account for statistical variance.

Due to the extreme amount of data being represented, this is a very dense plot. It would be unintelligible without the use of color to distinguish the various numbers of CPUs. (If you are reading this online using a Web browser, then the use of color helps you to see the separate threads. If you are reading a monochrome printed version, then the different shades of gray may provide some hint.)

The top plot shows actual elapsed wall clock time for each run. The bottom plot shows the speedup that using more than one CPU achieves relative to the performance of a single CPU doing the same task. (Hence the speedup for 1 CPU is a flat line with value 1). The calculation of the parallelism of the computation was performed as described in the description of Amdahl's Law, above. It represents the parallelism of the entire system, comprising the IDL program, the underlying operating system, all other programs running on the same system, and the underlying hardware. TIME_THREAD uses the final data point to compute this value.

Significant fluctuations in runtime appear between runs, even though the number of data elements being processed is almost the same. You see statistical variation in such measurements even with a single threaded program. They occur because the multitasking operating system underneath schedules processes (and thus, their threads) to run at various times that are independent of what IDL is doing at any given time. This causes individual runtimes to vary, though the average performance will be very predictable. This effect is magnified when a program uses multiple threads, because the operating system must schedule all of the threads. Another way to say this is that the performance of threaded programs is statistical in nature.

The following TIME_THREAD run uses far fewer data points, which makes it easier to see general trends:



mpod: Red Hat Linux 6.2 (Dell Precision 420), 2 X 600MHz Intel X86 CPUs, 128MB memory

There are 2 significant points illustrated by this run:

1. The use of multiple threads instead of a single thread implies a fixed amount of program overhead. Put simply, the program needs to spend time dividing up the work, starting the thread pool, and finally waiting for the threads to finish. The underlying thread API and operating system add the overhead of locking used to protect data from simultaneous access, and the overhead of scheduling multiple threads to run. A single threaded program does not incur these expenses. If the amount of data to be processed is large enough, the speedup from threading will exceed the fixed cost of doing it, and the overall performance will go up. If, however, the amount of data is too small, threading will cost you more than the speedup, and your overall performance will drop. The plot above shows clearly that until we reach about 1.2 million elements, we would have been better off not using threads even though we have the ability to do so, and that we don't reach full potential from threading until about 2 million elements. This is the reason that IDL performs smaller computations using a single thread, and does not use the thread pool until the size of the problem reaches !CPU.TPOOL_MIN_ELTS elements. The proper threshold to use depends very much on factors beyond IDL's control or ability to determine (machine hardware, load, the actual computation being carried out). For this reason, IDL provides a reasonable default for !CPU.TPOOL_MIN_ELTS, and provides the CPU procedure to allow the end user to change it to a more appropriate level for their system. In order to see the true effect of threading, TIME_THREAD sets this value to zero so that all possible computations are threaded.

2. The second interesting aspect of this plot is the way elapsed time begins to climb at about 12 million elements, and the fact that the speedup from using 2 CPUs simultaneously deteriorates to the single CPU level. This represents the onset of paging from the virtual memory system. TIME_THREAD uses the last data point to compute the level of parallelism, which accounts for the nonsensical parallelism reported. For the purposes of the measurements below, the computations were cut off before the system ran out of memory to avoid this effect. The lesson for the end user is that they might want to switch to single threaded operation if they wish to perform a computation that is too large for their hardware to handle in memory.

Example System Results

This section contains the results of running TIME_THREAD on ITT-VIS's MP systems. Results are presented in alphabetical order, based on hostname. For each system, two TIME_THREAD plots are given, one for B=A+A, and one for B=SQRT(A).

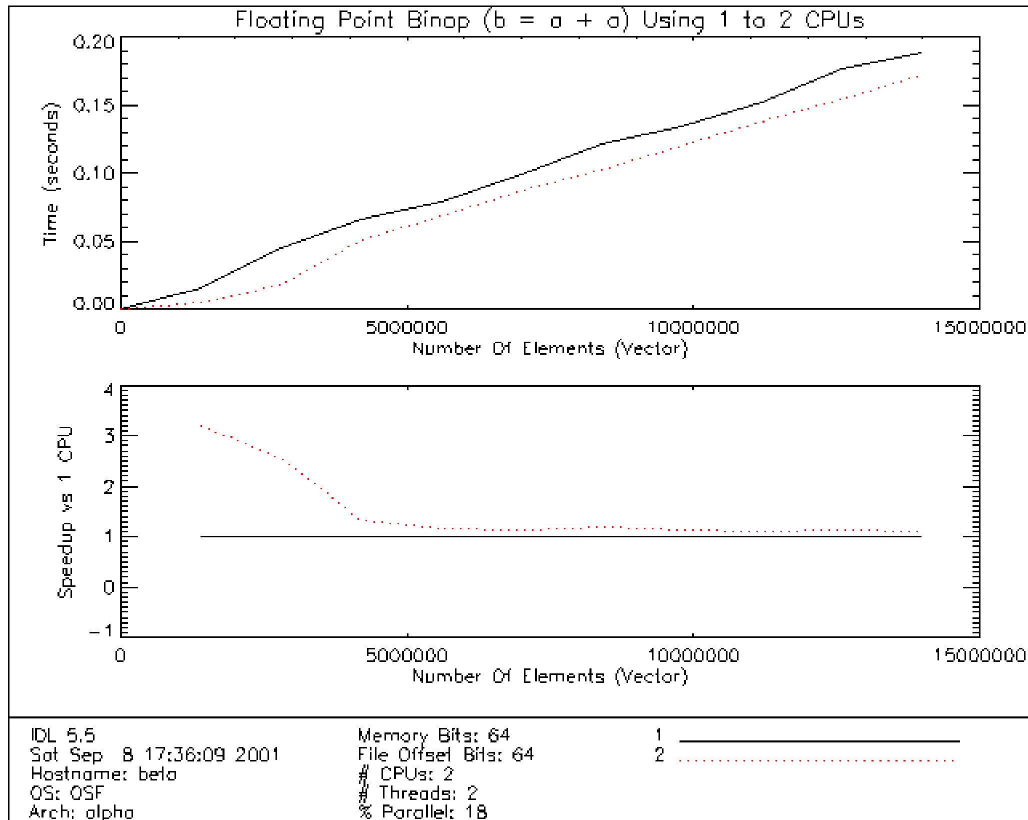
For the purposes of evaluating the thread pool, the parallelism score is the most interesting fact. However, one should pay attention to more than this single metric. Although high parallelism is desirable, the metric of overall importance to an IDL user is total elapsed time.

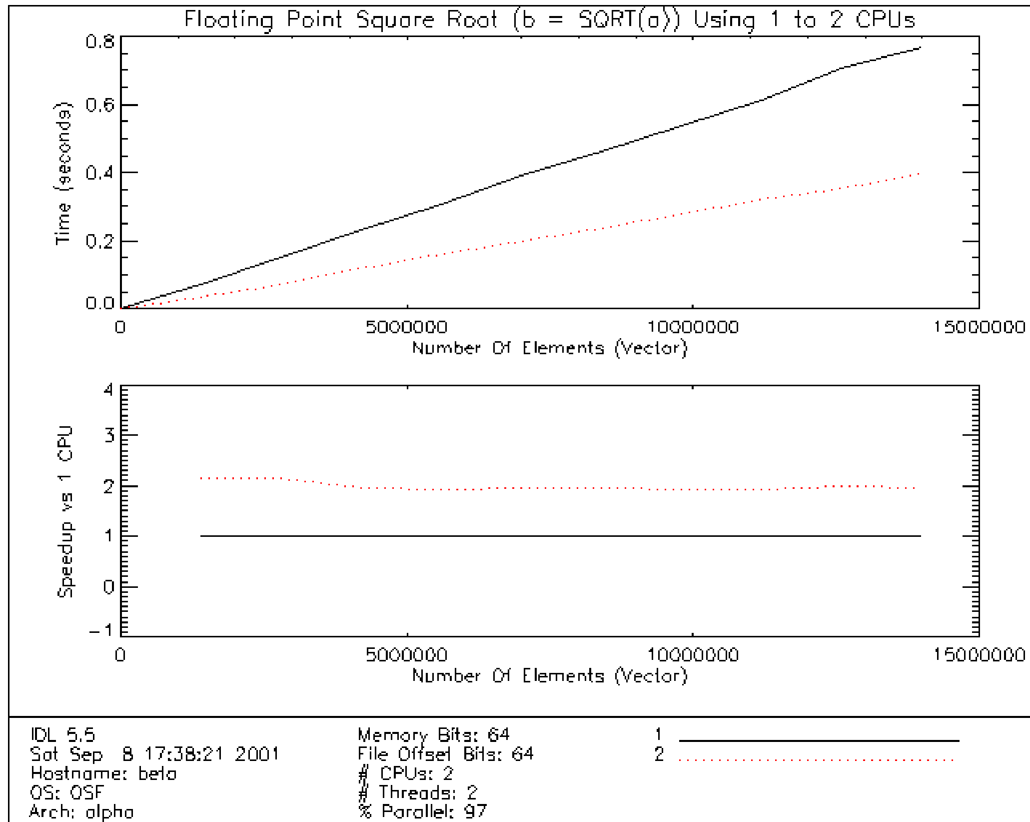
beta: Compaq Tru64 5.1 (DS20E), 2 X 667MHz Alpha EV6.7 (21264A) CPUs, 1GB memory

One would expect this system to do very well, based both on the quality of the hardware and the reputation of the operating system. The low level of parallelism exhibited by the computation of B=A+A was therefore surprising. On the surface, it appears that the system is displaying poor concurrency, and that the individual threads are not actually running in parallel. A week spent searching for the cause of this apparent problem showed that this not the case at all, and that Tru64 is doing a good job of scheduling threads.

By happy coincidence, the hardware running Alpha/Linux (epsilon) is identical to beta. The fact that epsilon achieves much higher parallelism served to deepen the mystery, until it became clear that beta is completing the computation in half the time. Experimentation with the Tru64 compiler shows that it generates such tight code for this particular computation that the job completes before the thread pool overhead is amortized. If you build an unoptimized IDL for Tru64, you get similar results to those posted by Alpha/Linux, underscoring the role of the compiler.

Changing the computation being measured to one that does not map as directly to the hardware (square root) demonstrates that Tru64 UNIX is capable of excellent parallelism if the computation merits it. Similarly, the extremely slow (though highly parallel) performance of Alpha/Linux on square root demonstrates the role of the system math library:



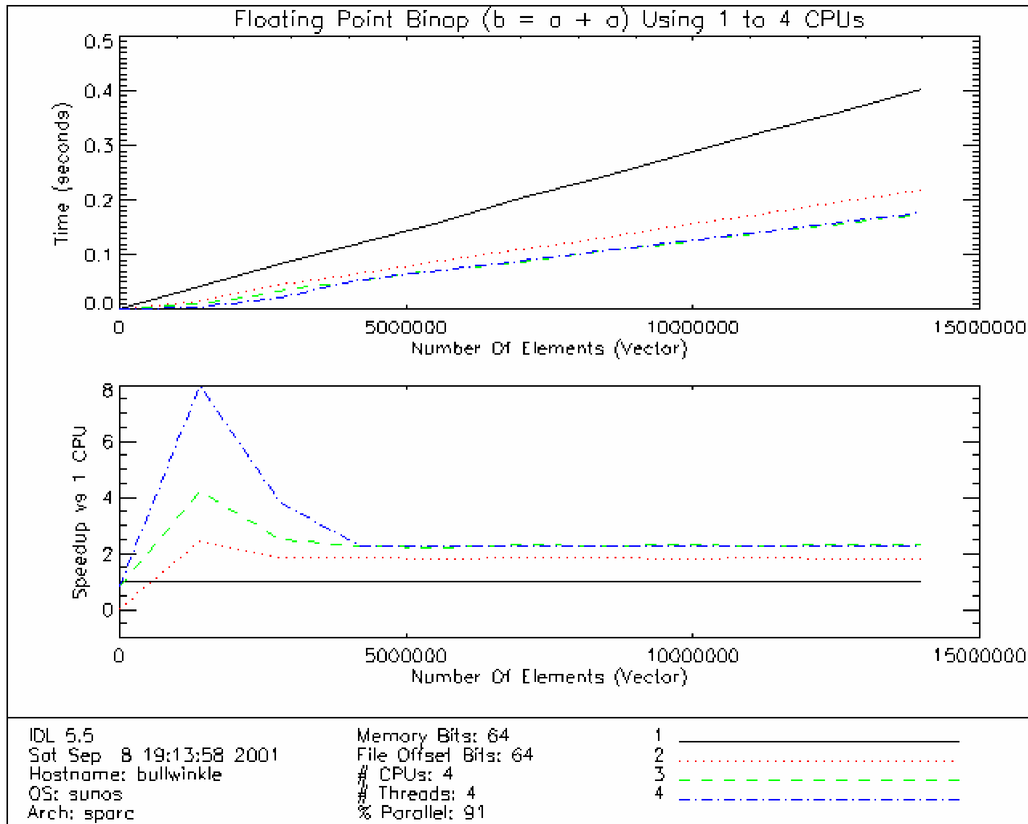


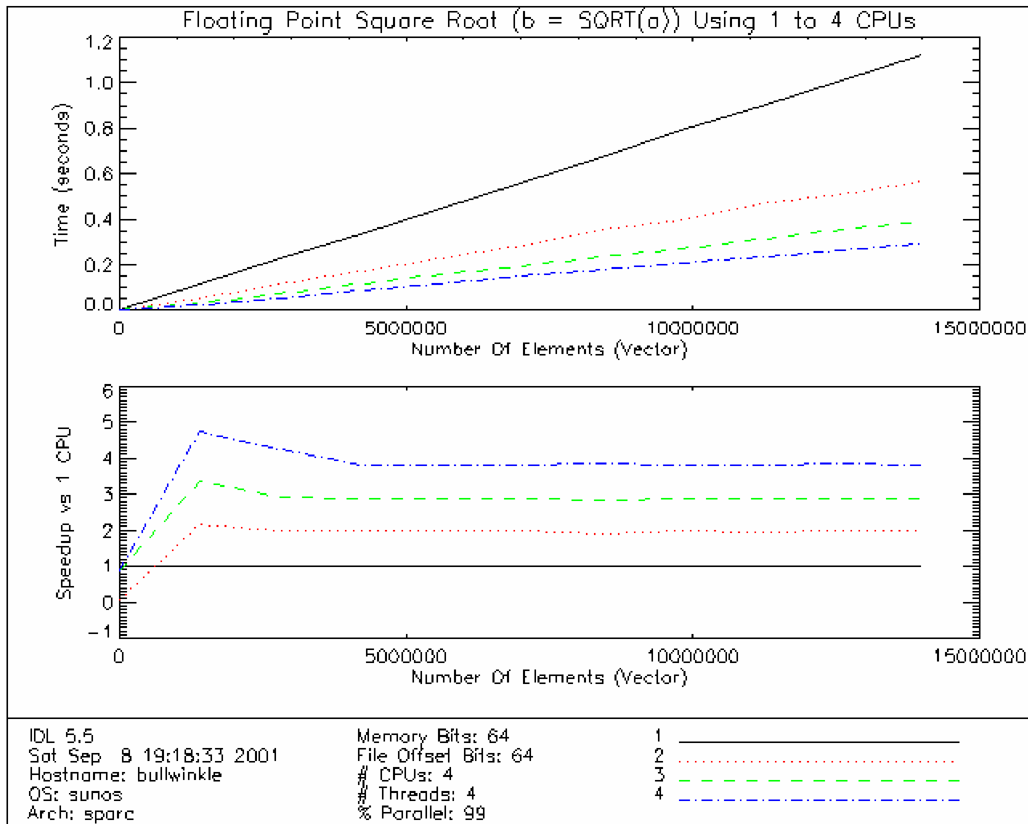
bullwinkle: SunOS 5.8 (Solaris 8, SPARC Ultra 80), 4 X 450MHz SPARC CPUs, 4GB memory

This system delivers a consistently high level of parallelism. It has 4 CPUs, unlike the other systems, which have only 2. You can see the natural diminishing rate of improvement in performance that comes with the addition of each CPU.

The higher than average speedup visible in the area of 1 million points is interesting. A likely explanation for this is memory cache effects.

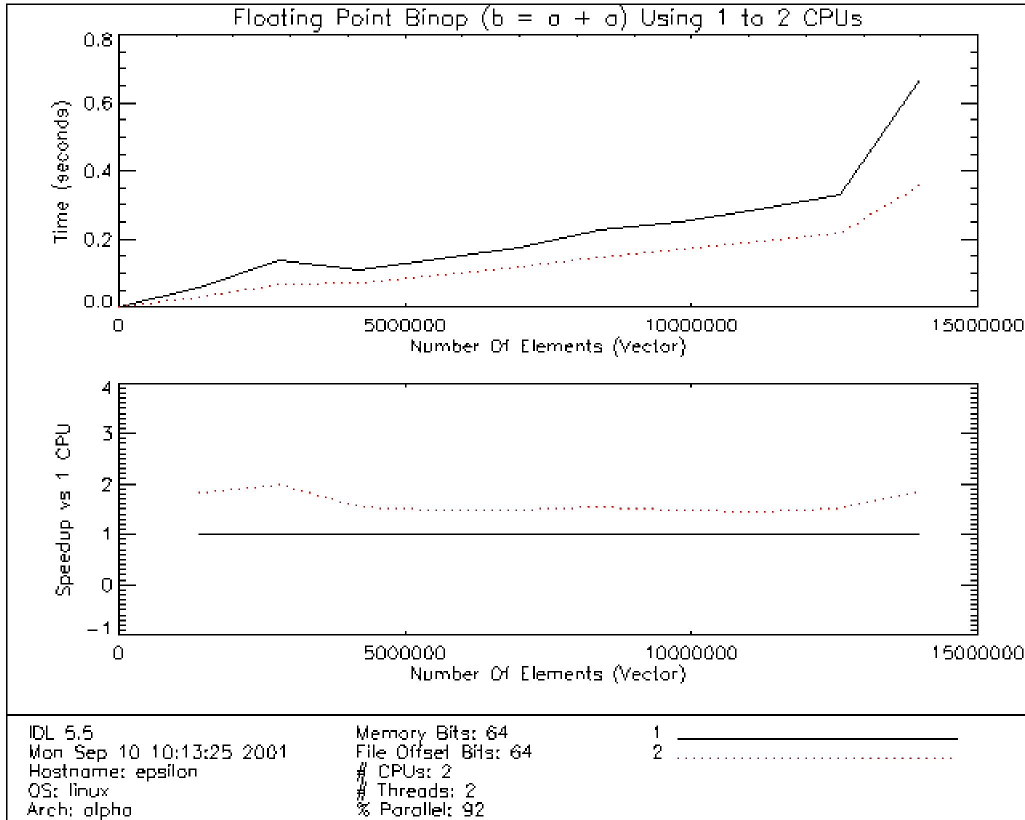
The SQRT function seems to scale exceptionally well on this system:

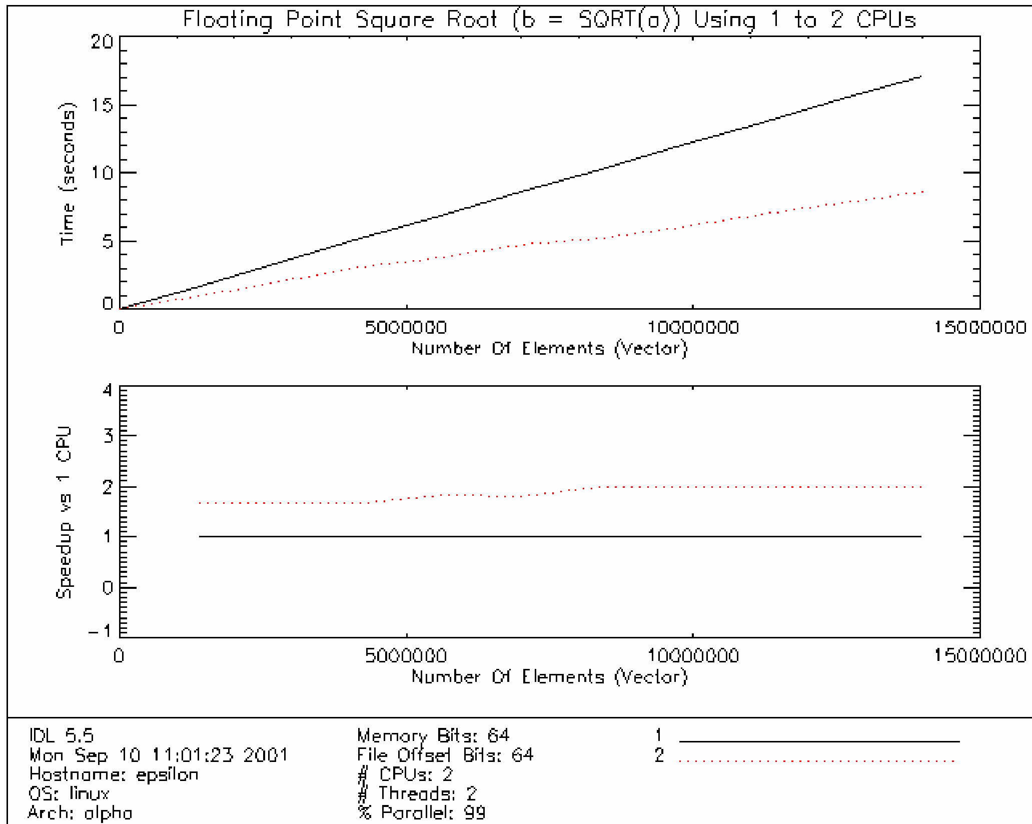

Floating Point Binop (b = a + a) Using 1 to 4 CPUs
Time (seconds)
Number Of Elements (Vector)
Speedup vs 1 CPU
Number Of Elements (Vector)
IDL 5.5
Sat Sep 8 19:13:58 2001
Hostname: bullwinkle
OS: sunos
Arch: sparc
Memory Bits: 64
File Offset Bits: 64
CPUs: 4
Threads: 4
% Parallel: 91


Floating Point Square Root (b = SQRT(a)) Using 1 to 4 CPUs
Time (seconds)
Number Of Elements (Vector)
Speedup vs 1 CPU
Number Of Elements (Vector)
IDL 5.5
Sat Sep 8 19:18:33 2001
Hostname: bullwinkle
OS: sunos
Arch: sparc
Memory Bits: 64
File Offset Bits: 64
CPUs: 4
Threads: 4
% Parallel: 99

epsilon: Red Hat Linux 6.2 (Compaq DS20E), 2 X 667MHz Alpha EV6.7 (21264A) CPUs, 1GB memory

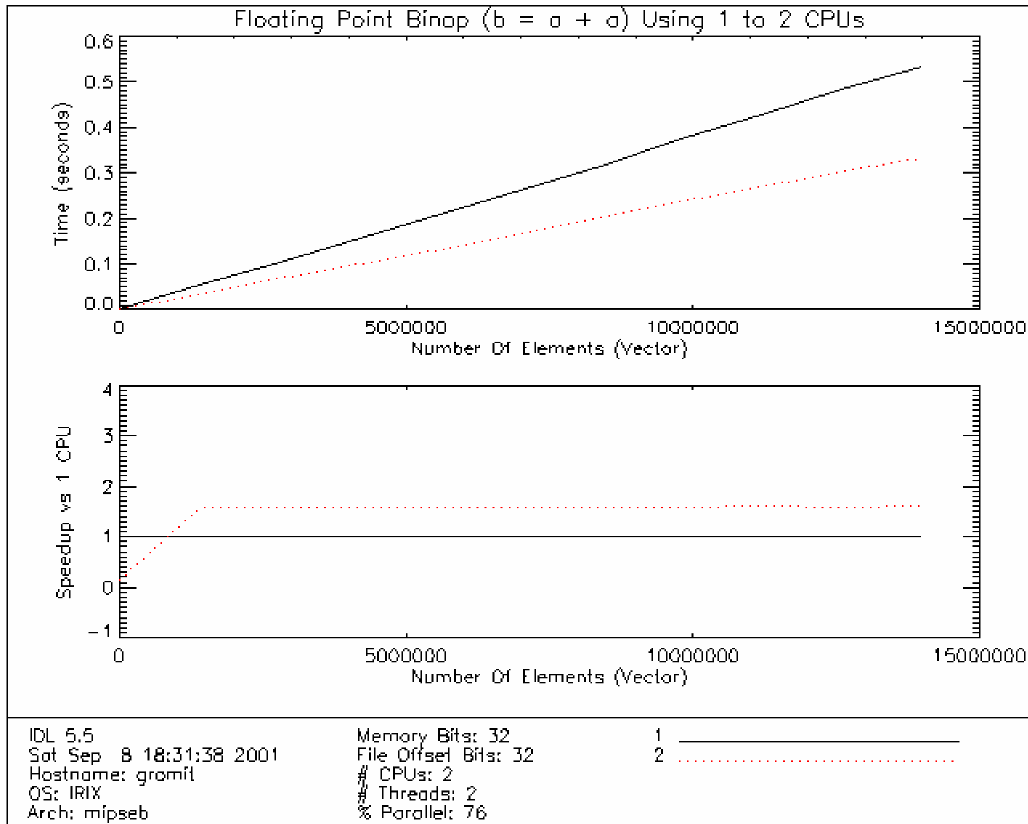
Although posting much better parallelism results for floating point addition than Tru64 UNIX on identical hardware, Alpha/Linux is much slower overall. This appears to be mainly due to better code generated by the Compaq C compiler. The performance of the Alpha/Linux square root implementation in the system math library obviously leaves room for improvement:

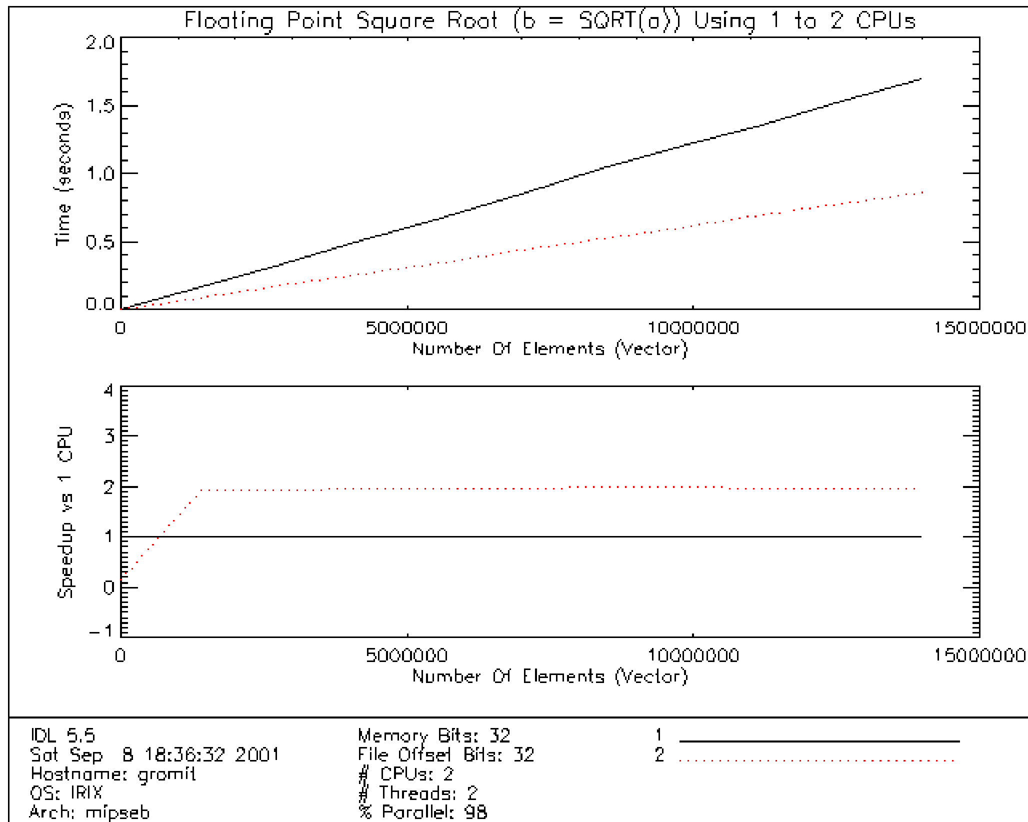




gromit: IRIX 6.5.10m (Octane), 2 X 300MHz R12000 CPUs, 512MB memory

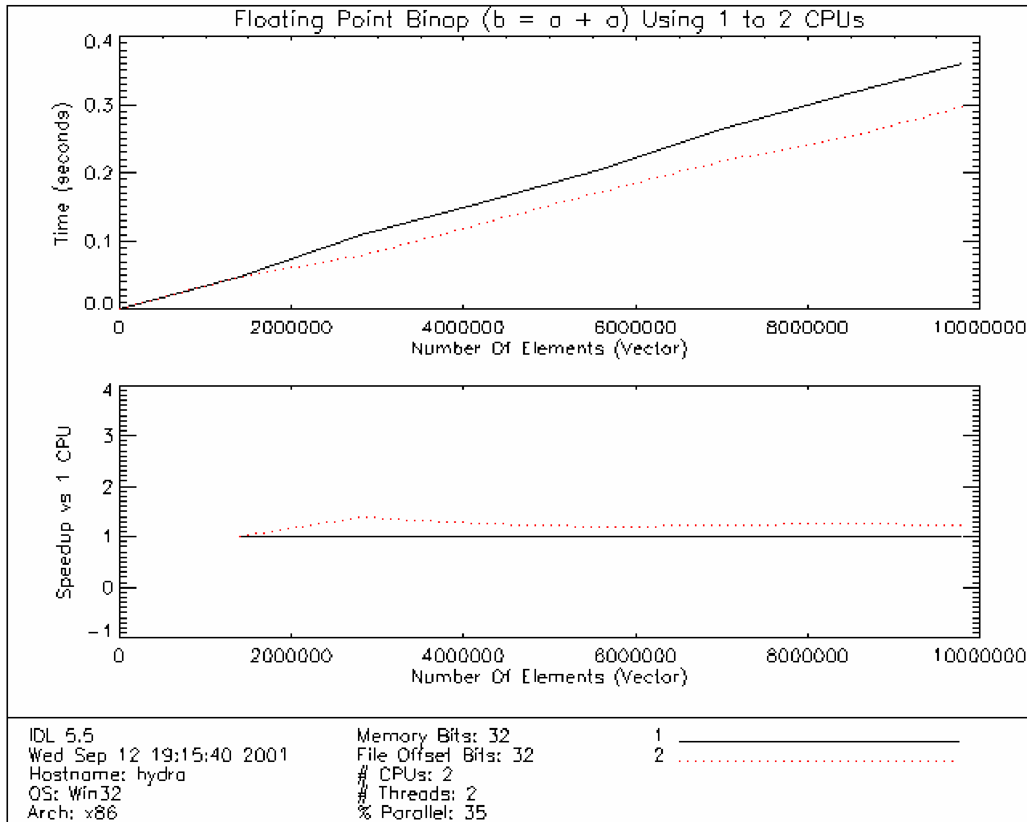
This system delivers solid parallelism for these two computations that puts it in the middle of the pack:

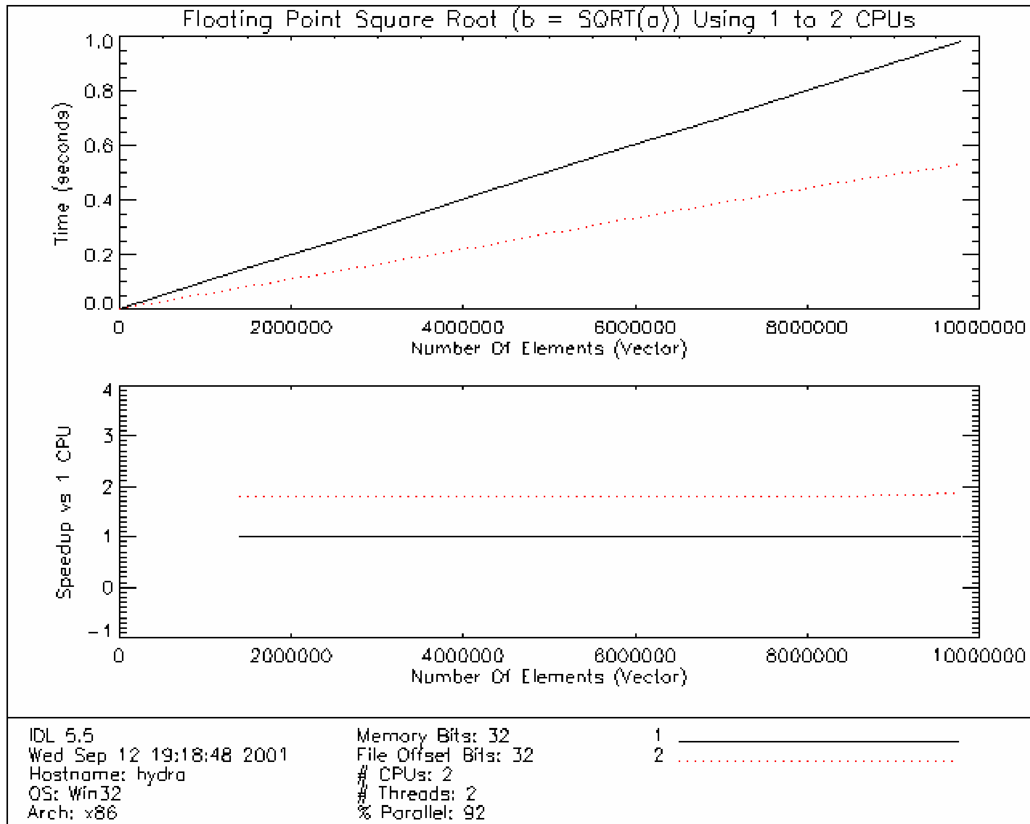

Floating Point Binop (b = a + a) Using 1 to 2 CPUs
Time (seconds)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0
5000000
10000000
15000000
Number Of Elements (Vector)
Speedup vs 1 CPU
-1
0
1
2
3
4
0
5000000
10000000
15000000
Number Of Elements (Vector)
IDL 5.5
Sat Sep 8 18:31:38 2001
Hostname: gromit
OS: IRIX
Arch: mipseb
Memory Bits: 32
File Offset Bits: 32
CPUs: 2
Threads: 2
% Parallel: 76
1
2


Floating Point Square Root (b = SQRT(a)) Using 1 to 2 CPUs
Time (seconds)
0.0
0.5
1.0
1.5
2.0
0
5000000
10000000
15000000
Number Of Elements (Vector)
Speedup vs 1 CPU
-1
0
1
2
3
4
0
5000000
10000000
15000000
Number Of Elements (Vector)
IDL 5.5
Sat Sep 8 18:36:32 2001
Hostname: gromit
OS: IRIX
Arch: mipseb
Memory Bits: 32
File Offset Bits: 32
CPUs: 2
Threads: 2
% Parallel: 98
1
2

hydra: Windows NT 4 (Dell Precision 420), 2 X 600MHz X86 CPUs, 128MB memory

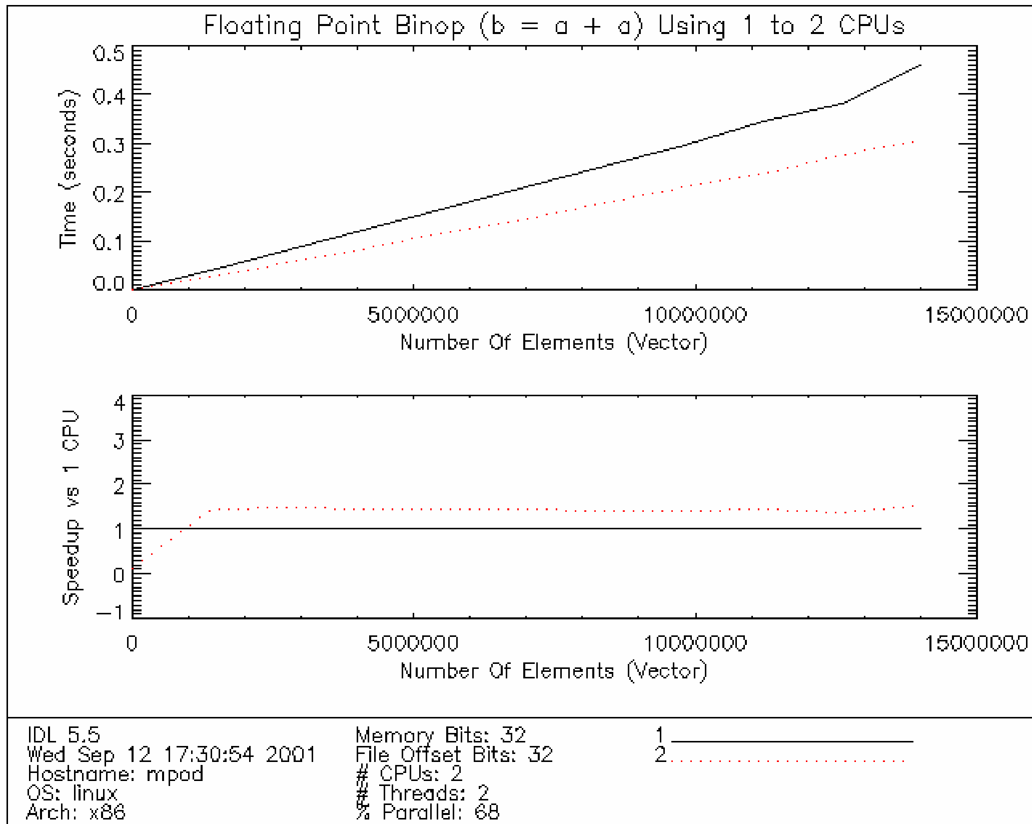
ITT-VIS owns two identical dual processor Dell systems, one running Windows NT and the other (mpod, below) running Red Hat Linux 6.2. The parallelism achieved by hydra, running NT, is not as high for these computations as many of the other systems, but is quite reasonable given the price of mid-range PC hardware:



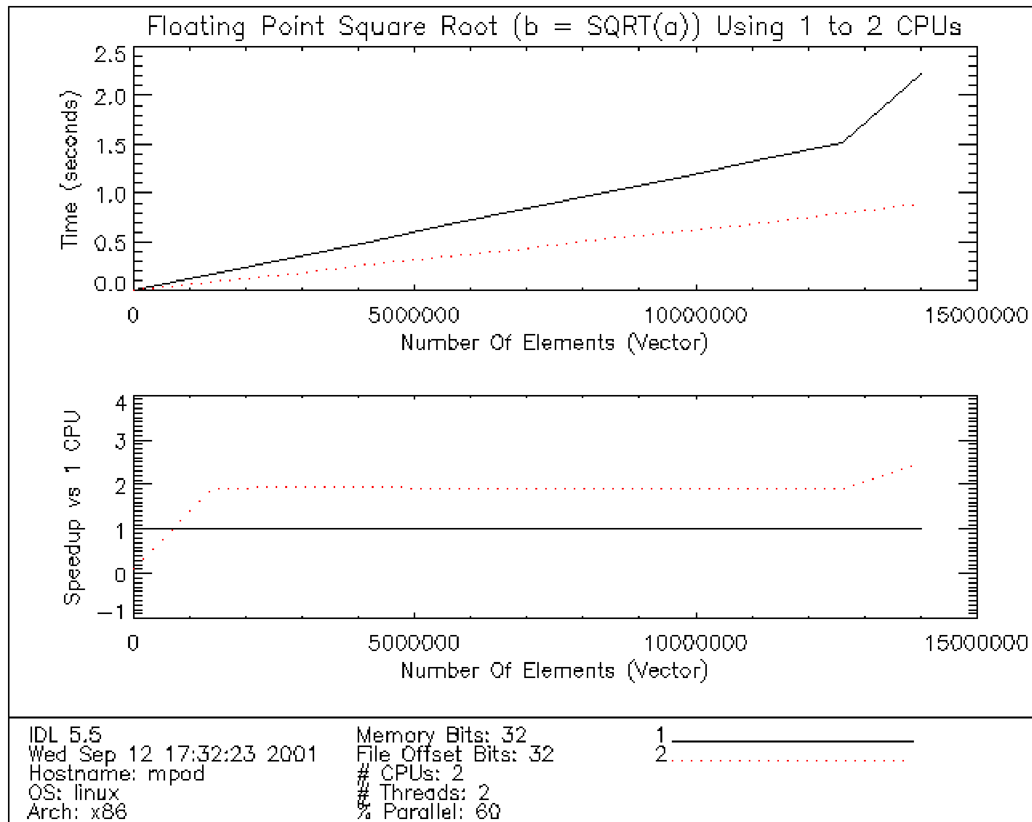


mpod: Red Hat Linux 7.1 (Dell Precision 420), 2 X 600MHz Intel X86 CPUs, 128MB memory

Although mpod shares identical hardware with hydra, the threading results for these computations are very different. The different IDL binaries (Win32 and Linux) and different operating systems (Red Hat Linux 7.1 and Windows NT 4) between the two systems may affect the results:



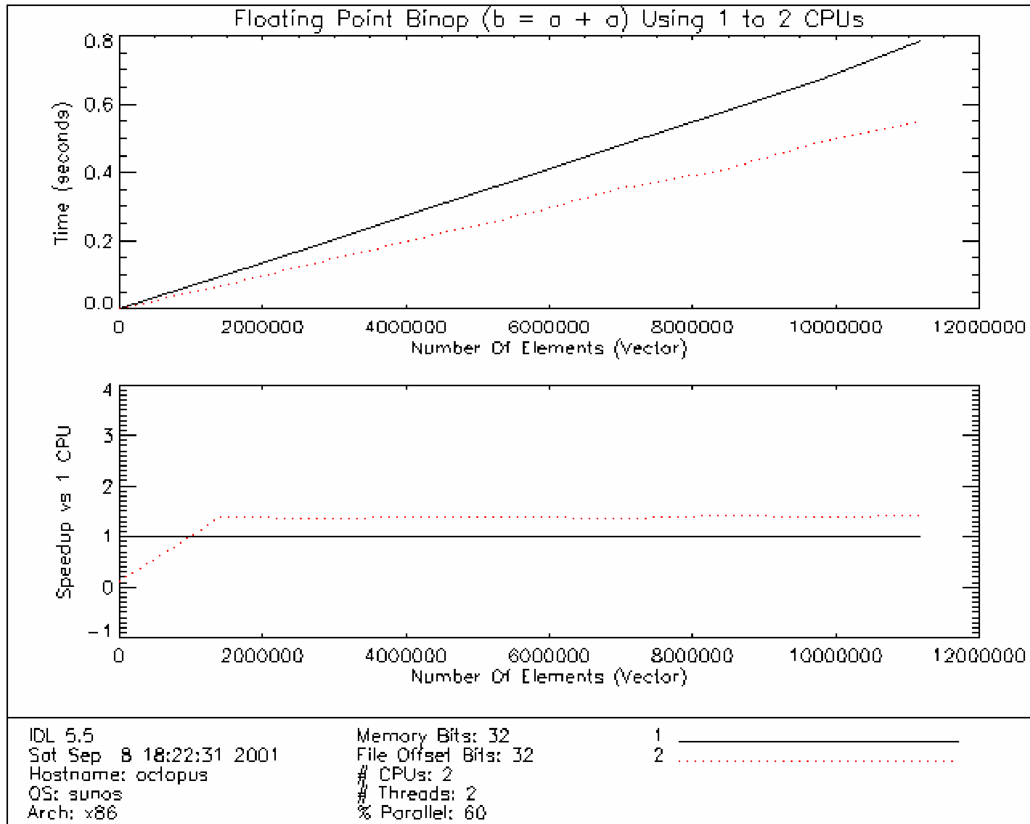
Floating Point Binop (b = a + a) Using 1 to 2 CPUs
Time (seconds)
0.0
0.1
0.2
0.3
0.4
0.5
0
5000000
10000000
15000000
Number Of Elements (Vector)
Speedup vs 1 CPU
−1
0
1
2
3
4
0
5000000
10000000
15000000
Number Of Elements (Vector)
IDL 5.5
Wed Sep 12 17:30:54 2001
Hostname: mpod
OS: linux
Arch: x86
Memory Bits: 32
File Offset Bits: 32
CPUs: 2
Threads: 2
% Parallel: 68
1
2

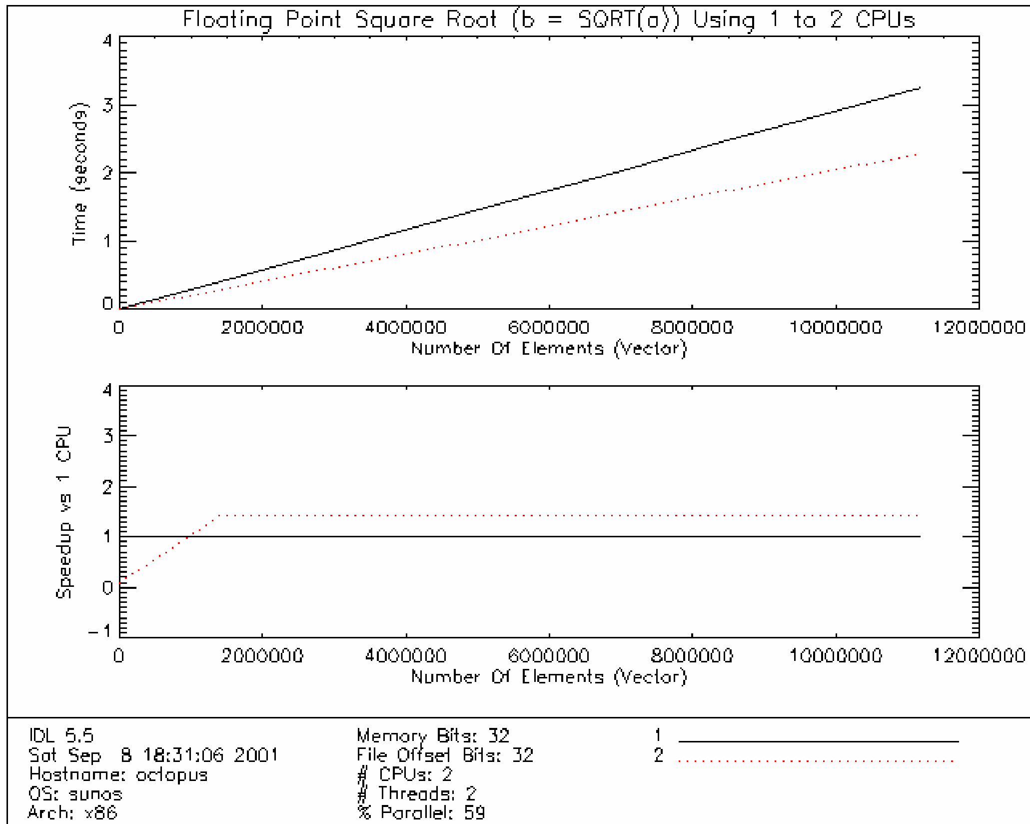


Floating Point Square Root (b = SQRT(a)) Using 1 to 2 CPUs
Time (seconds)
0.0
0.5
1.0
1.5
2.0
2.5
0
5000000
10000000
15000000
Number Of Elements (Vector)
Speedup vs 1 CPU
−1
0
1
2
3
4
0
5000000
10000000
15000000
Number Of Elements (Vector)
IDL 5.5
Wed Sep 12 17:32:23 2001
Hostname: mpod
OS: linux
Arch: x86
Memory Bits: 32
File Offset Bits: 32
CPUs: 2
Threads: 2
% Parallel: 60
1
2

octopus: SunOS 5.7 (Solaris 7, HP-Kayak), 2 X 266MHz Intel X86 CPUs, 128MB memory

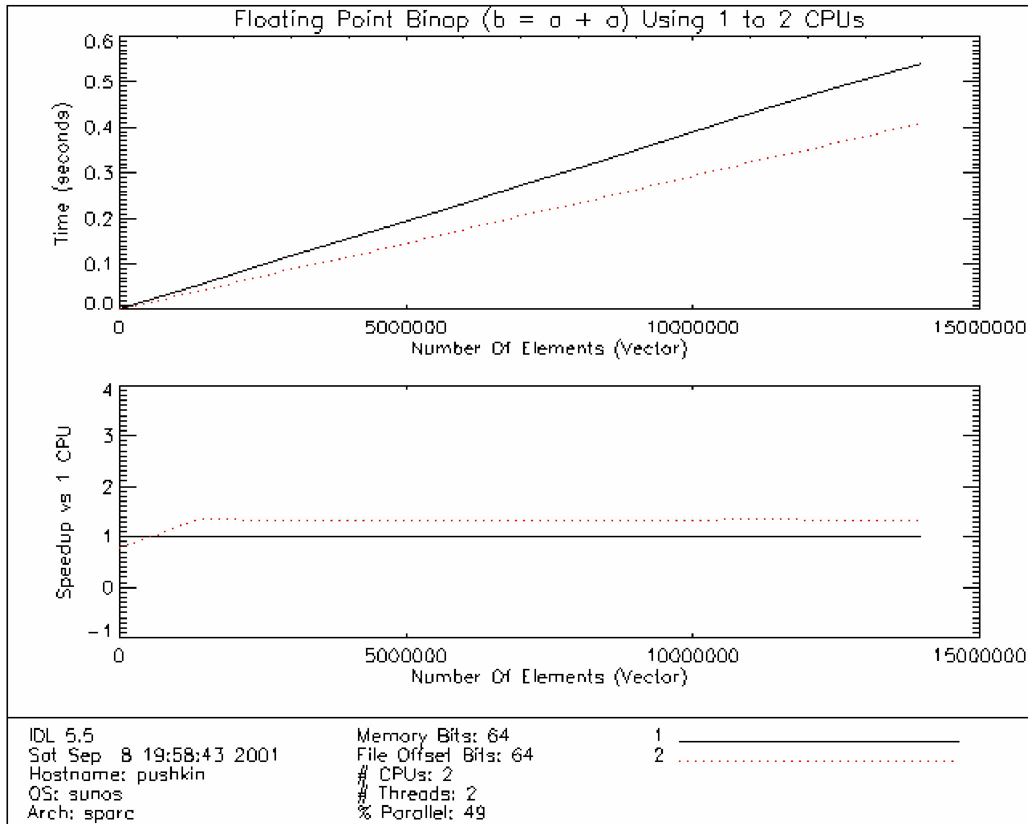
Octopus runs the Solaris operating system, but since it is Solaris for Intel hardware, the operating system and IDL binary differ from the SPARC versions. The parallelism achieved is higher than that posted by pushkin (SPARC Ultra 2), but significantly lower than that achieved by bullwinkle (Ultra 80). This would seem to indicate that high quality PC hardware can achieve results similar to more expensive workstation class hardware. The Kayak is very expensive by PC hardware standards, and bears a great deal of similarity to HP workstations in construction and quality of components:

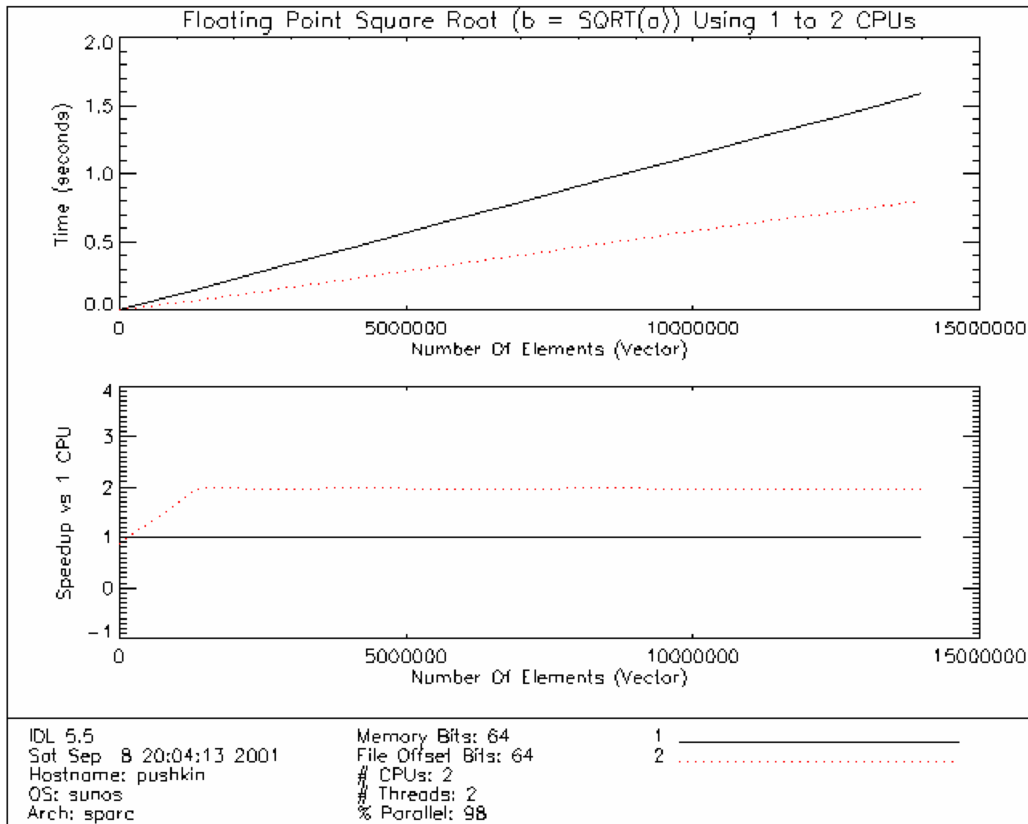




pushkin: SunOS 5.8 (Solaris 8, SPARC Ultra 2), 2 X 300MHz SPARC CPUs, 256MB memory

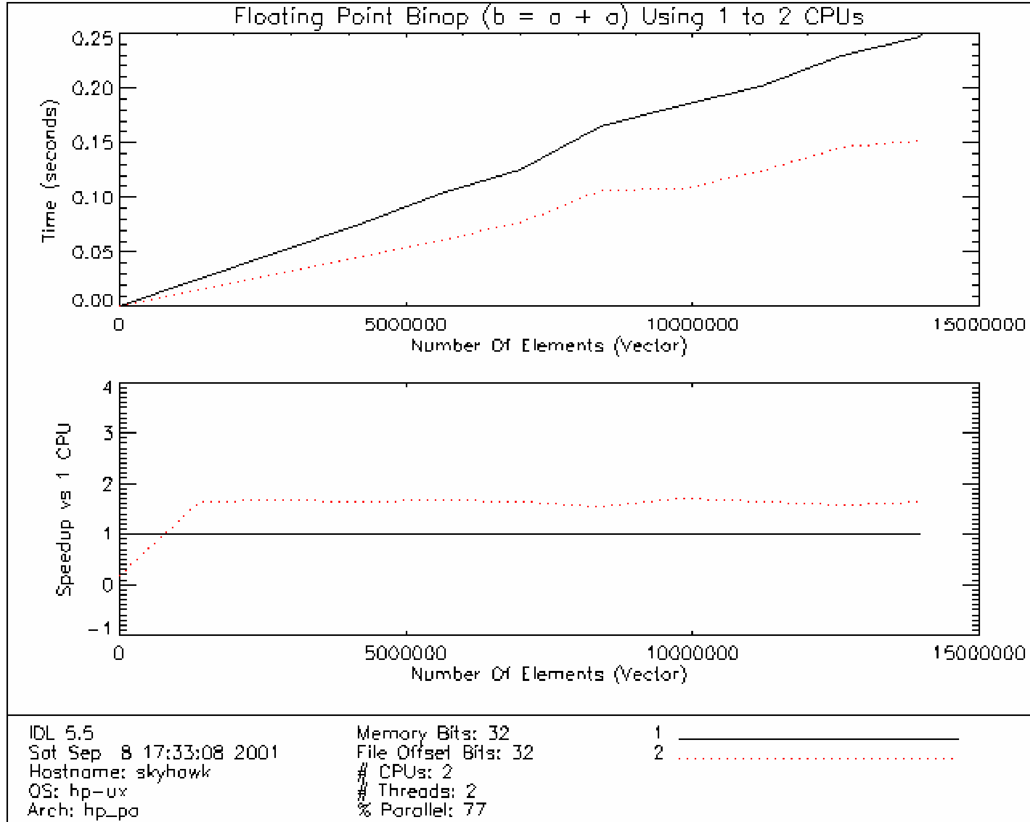
Pushkin runs the same version of Solaris as bullwinkle (Ultra 80), and has the same hardware architecture as that machine. This test was run using the same IDL binary as the bullwinkle test above. The only significant difference is the hardware itself. It follows that the dramatic difference in parallelism for addition is due to the difference in hardware:

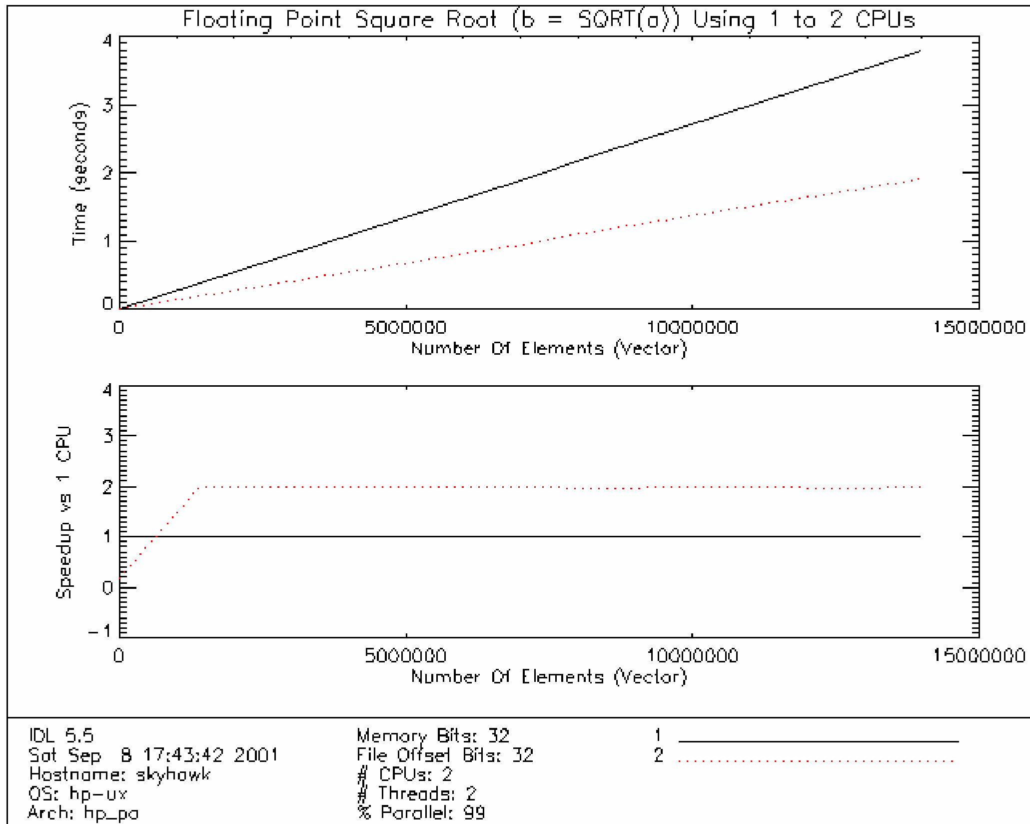

Floating Point Binop (b = a + a) Using 1 to 2 CPUs
Time (seconds)
Number Of Elements (Vector)
Speedup vs 1 CPU
IDL 5.5
Sat Sep 8 19:58:43 2001
Hostname: pushkin
OS: sunos
Arch: sparc
Memory Bits: 64
File Offset Bits: 64
CPUs: 2
Threads: 2
% Parallel: 49


Floating Point Square Root (b = SQRT(a)) Using 1 to 2 CPUs
Time (seconds)
Number Of Elements (Vector)
Speedup vs 1 CPU
IDL 5.5
Sat Sep 8 20:04:13 2001
Hostname: pushkin
OS: sunos
Arch: sparc
Memory Bits: 64
File Offset Bits: 64
CPUs: 2
Threads: 2
% Parallel: 98

skyhawk: HP-UX 11.0 (Visualize J5600), 2 X 550MHz PS RISC CPUs, 4GB memory

The level of parallelism for this system falls in the middle of the overall range:





Platform Comparisons

The following surface plots make it easier to make cross system comparisons by showing all timing and speedup data for the addition and square root computations for 4.2 million data points. The number of points was selected to be large enough that thread overhead might be amortized, and small enough that none of the systems being compared would encounter virtual memory paging.

In these plots, the data is sorted so that the taller bars will fall behind the shorter ones in order to avoid data from one system obscuring the results from another.

In the elapsed time plots, the slower systems will be in the back with the faster ones in front.

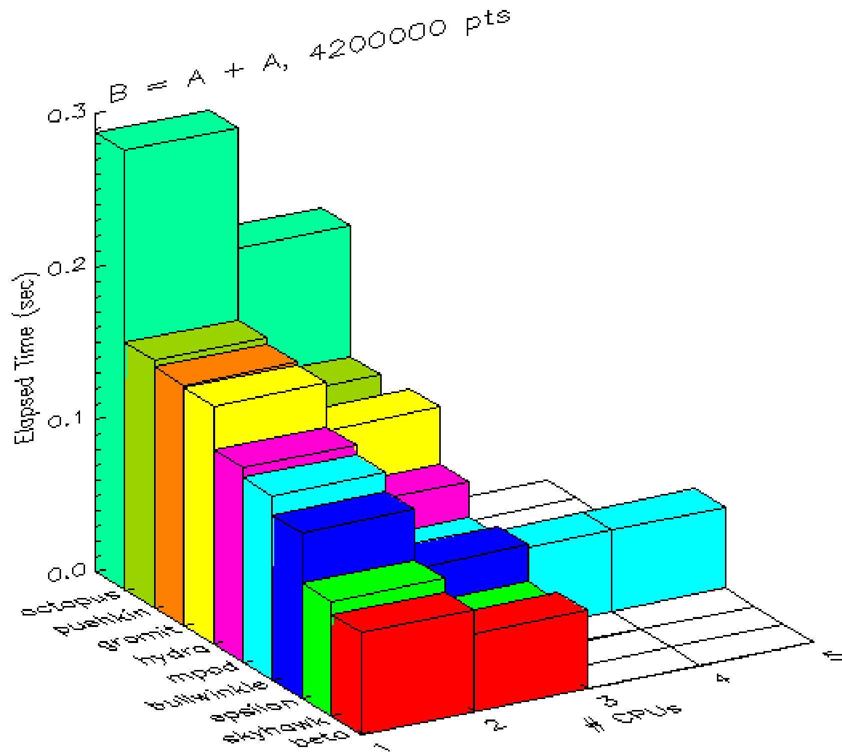
In the speedup plots, the systems that demonstrate higher levels of thread pool utilization will be in the back, while those with lower levels will be in front.

In most cases, a comparison of raw elapsed times between these systems is meaningless. These systems vary widely in age and hardware architecture, making direct comparison difficult. For example, octopus (Solaris X86) is very slow compared to most of the other systems. It is an old dual Intel Pentium system with 266Mhz CPUs. One could not reasonably expect it to outperform any of the other systems, but the results posted are quite respectable for that system.

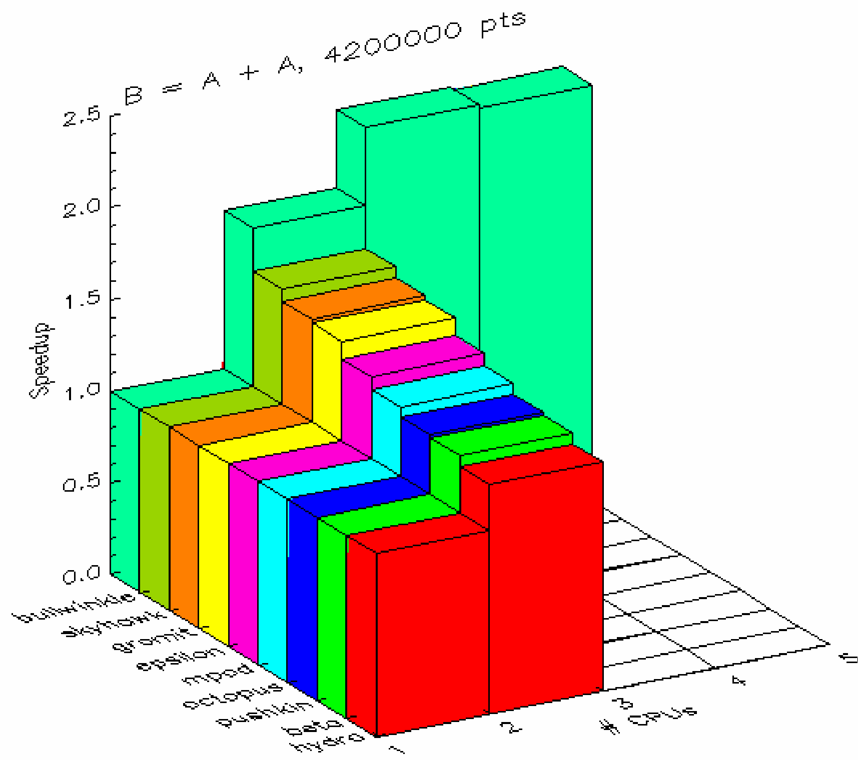
One notable case where such comparisons can be reasonable is when the underlying hardware is identical, but running different operating systems. In these measurements, hydra (Windows NT) and mpod (Red Hat Intel 7.1) are identical machines, as are beta (Compaq Tru64 UNIX) and epsilon (Red Hat Alpha 6.2).

Elapsed Time: A=B+B, 4200000 Points

Interestingly, the fastest system (beta, Compaq Tru64 UNIX) demonstrates poor parallelism for floating point addition. This issue is discussed in more detail above:

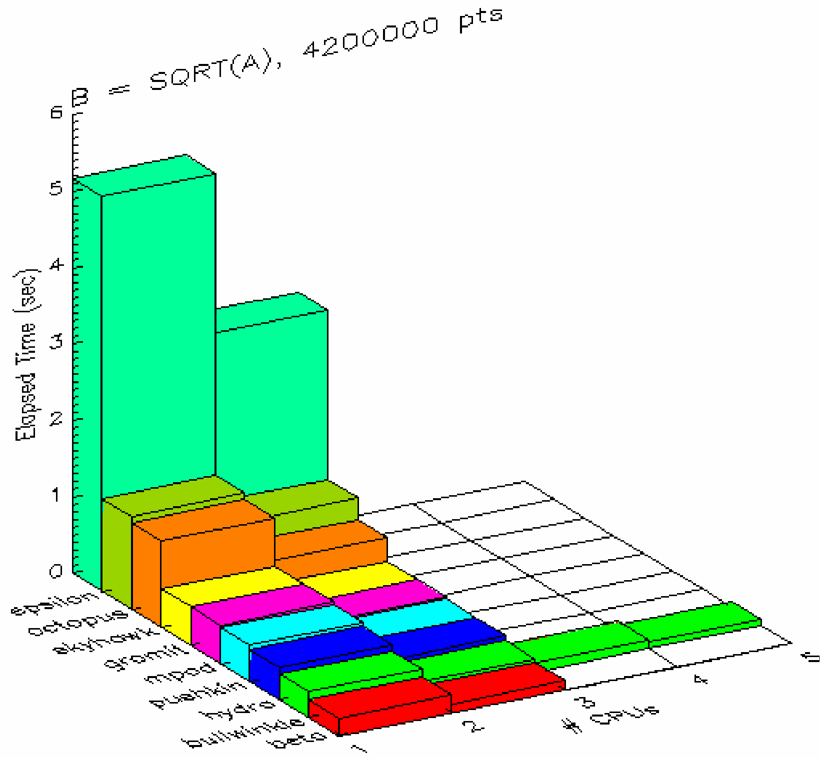


Speedup: A=B+B, 4200000 Points:

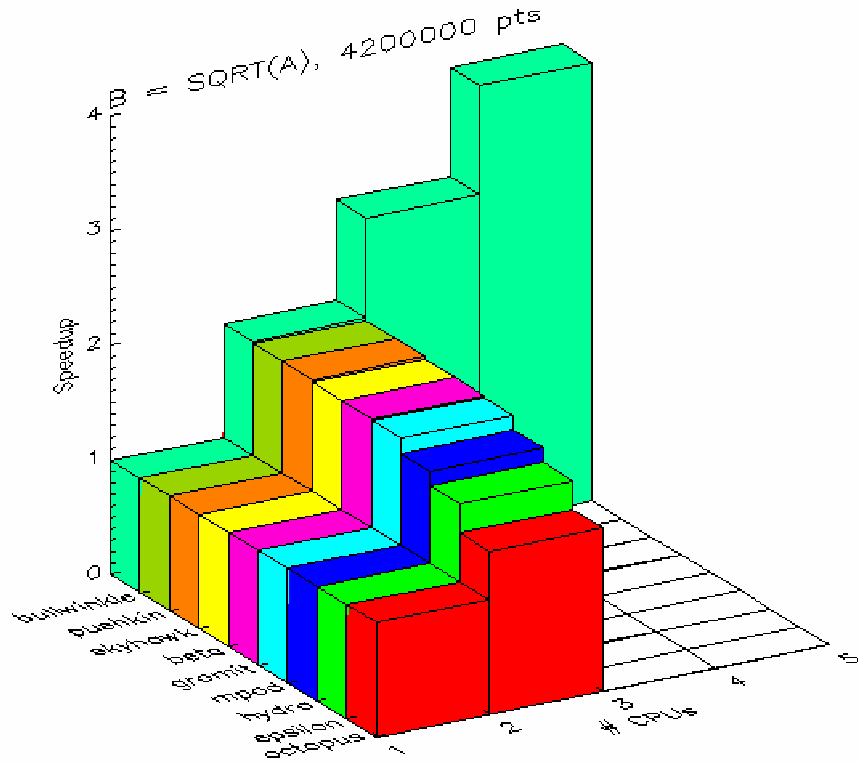


Elapsed Time: A=SQRT(B), 4200000 Points

In this case, the fastest and slowest machines actually have identical hardware. There is apparently something wrong with the implementation of the square root function under Alpha/Linux:



Speedup: A=SQRT(B), 4200000 Points:



Conclusions

Summary

On a multi-CPU machine primarily being used to run IDL for a single user, all but one of the CPUs spend most of their time sleeping. Use of the IDL thread pool under such circumstances usually speeds up the overall computation, and the benefit of the speedup increases with the size of the problem being solved, until the problem becomes too large for the hardware it is running on.

If the problem size exceeds the size of physical memory, all of the threads will simultaneously begin to page fault, requiring the virtual memory system to page in the desired data from disk. The speed of disk to memory transfer is orders of magnitude slower than that of memory to CPU, so the CPUs will stall and the benefit of multiple threads disappears. Performance in this case may actually be worse than for a single processor, because the multiple threads will compete with each other for limited memory to disk bandwidth, whereas a single thread would own all of the bandwidth.

Thread performance depends on the size and physical capabilities of the underlying hardware, the compiler used to generate the code, and the operating system, as well as on how many simultaneous threads are trying to run on the system (which is a factor of how many users are on the system, and the programs they are running). The variety and scope of such issues means that it is impossible for IDL to correctly determine the best number of threads and the minimum number of elements required to consider threading in many, if not most, situations. This underscores the need for manual user control over these parameters, as provided by the IDL thread pool via the CPU procedure and the thread pool keywords to individual function calls.

Overall thread performance depends on the quality of the algorithms and implementation of the program, the compiler used, the threading implementation of the operating system, and the underlying hardware. To make useful comparisons, it is necessary to hold three of these factors steady while altering the fourth. Such an exercise is resource intensive and not realistic given the fact that many of these parameters are implicitly tied together and cannot be separated. However, some of the included system configurations come close, and provide some interesting insights.

The same IDL binary, running on the same version of Solaris, on two different SPARC platforms yield widely differing levels of parallelism. Clearly the hardware used is a very important part of the performance story. A related data point is the performance of the 2-CPU Intel PC running Solaris X86, the parallelism of which lies between the two SPARC systems. Although interesting, it is difficult to draw firm conclusions about this because it is likely that Sun does not put as much effort into their X86 C compiler as they do for the SPARC version.

ITT-VIS's two alpha CPU systems consist of identical hardware. One system is running Compaq Tru64 UNIX (beta) while the other runs Red Hat Linux 6.2 (epsilon). For floating point addition, the Tru64 system beats the Red Hat system on computational throughput, but does not score highly on thread pool utilization. On square root, both systems score highly on thread pool utilization, but Tru64 beats it by an even larger margin than before. Alpha/Linux performs this operation especially slowly, leading to an interesting situation in which the same hardware is both the fastest and the slowest system in the overall timing. In this case, the Tru64 compiler is the biggest difference, and possibly the implementation of the Linux square root algorithm.

Advice (So What Should I Buy?)

The measurements shown in this document only give an accurate picture of relative performance of these specific ITT-VIS systems, running these specific computations. There are so many variables involved in thread pool performance (hardware, compiler, operating system, algorithm) that it is dangerous to infer too much from any specific set of measurements, including those presented here.

Any of the systems described in this document can be used effectively with the IDL thread pool. However, the best system to buy depends entirely on the specific job for which you intend to use it. This is especially tricky for a general program like IDL, because most people use IDL for a variety of things. On any given system, the thread pool will help with some things, but not others. Among the many lessons to consider, note that the system with the poorest thread pool utilization for floating point arithmetic was also the fastest system overall for that task, and that these facts are clearly related. It would be a mistake to eliminate a system from consideration because it was too fast to use the thread pool well on a given problem. Remember that computational throughput is the goal, and not necessarily thread pool utilization. Thread pool utilization is but a single variable in a very complex situation.

The best piece of advice is to run your own program on a potential system, if at all possible, and use that information to make your choice. For numeric computation, it appears that you can pick your operating system based on concerns other than thread performance, although the compiler used to build IDL is important, and compilers tend to be tied tightly to operating systems. Whatever OS you run, you will have some hardware choices to make, and performance differences between different hardware can be significant.

Raw performance is often not the most important criterion for a general-purpose machine, and it pays to focus on affordability, usability, and maintainability. You might be happier on a slower system that runs an operating system that you find more usable (however you choose to define that term). You might be better off running a system that is common in your area of specialization. Your system administrator might prefer you to run a system that is common in your environment, and you might be well advised to keep such a person happy.

If performance is your highest goal, then you will have to do some work to pick the best system. If the measurements in this document prove anything, it is that the specific computation being executed can make a large difference in performance. A system can excel at one operation, but be average or worse on others. The combination of hardware, compiler, operating system, and system libraries introduce so many variables that it is dangerous to generalize too much from measurements of any specific operation. So, if performance is your most important criterion:

- Determine which operations are the most important to you.
- Construct IDL test programs that emphasize those operations.
- Run the tests and measure their performance.
- Buy the best system that your budget will handle.

Finally, it is important to understand that while threading can provide significant speedups, there are cases when it fails to speed things up, and in such cases, one would have been far better off not to have threaded the computation at all. The user has to be involved in making this determination. The IDL CPU procedure and the thread pool keywords to individual function calls give you the necessary control over this decision.

To learn more about IDL's functionality please visit www.ittvis.com/idl/ or contact your ITT-VIS sales representative at (303)-786-9900 <info@ittvis.com>.

Appendices

Appendix A: IDL Routines that use the Thread Pool

Multithreading does not offer the possibility of increased execution speed for all IDL routines. The operators and routines currently using the thread pool in IDL are listed below, grouped by functional category.

Binary and Unary Operators:

- –
- ––
- +
- ++
- NOT
- AND
- /
- *
- EQ
- NE
- GE
- LE
- GT
- LT
- >
- <
- OR
- XOR
- ^
- MOD
- #
- ##

Note: If an operator uses the thread pool, any compound assignment operator based on that operator (+=, *=, etc.) also uses the thread pool.

Mathematical Routines:

- ABS
- ERRORF
- MATRIX_MULTIPLY
- ACOS
- EXP
- PRODUCT
- ALOG
- EXPINT

- ROUND
- ALOG10
- FINITE
- SIN
- ASIN
- FLOOR
- SINH
- ATAN
- GAMMA
- SQRT
- CEIL
- GAUSSINT
- TAN
- CONJ
- IMAGINARY
- TANH
- COS
- ISHFT
- VOIGT
- COSH
- LNGAMMA

Image Processing Routines:

- BYTSCL
- INTERPOLATE
- CONVOL
- POLY_2D
- FFT
- TVSCL

Array Creation Routines:

- BINDGEN
- LINDGEN
- BYTARR
- L64INDGEN
- CINDGEN
- MAKE_ARRAY
- DCINDGEN
- REPLICATE
- DCOMPLEXARR
- UINDGEN
- DINDGEN
- ULINDGEN
- FINDGEN
- UL64INDGEN
- INDGEN

Non-string Data Type Conversion Routines:

- BYTE
- LONG

- COMPLEX
- LONG64
- DCOMPLEX
- UINT
- DOUBLE
- ULONG
- FIX
- ULONG64
- FLOAT

Array Manipulation Routines:

- MAX
- TOTAL
- MIN
- WHERE
- REPLICATE_INPLACE

Programming and IDL Control Routines:

- BYTEORDER
- LOGICAL_OR
- LOGICAL_AND
- LOGICAL_TRUE

Appendix B: Additional Benchmark Results

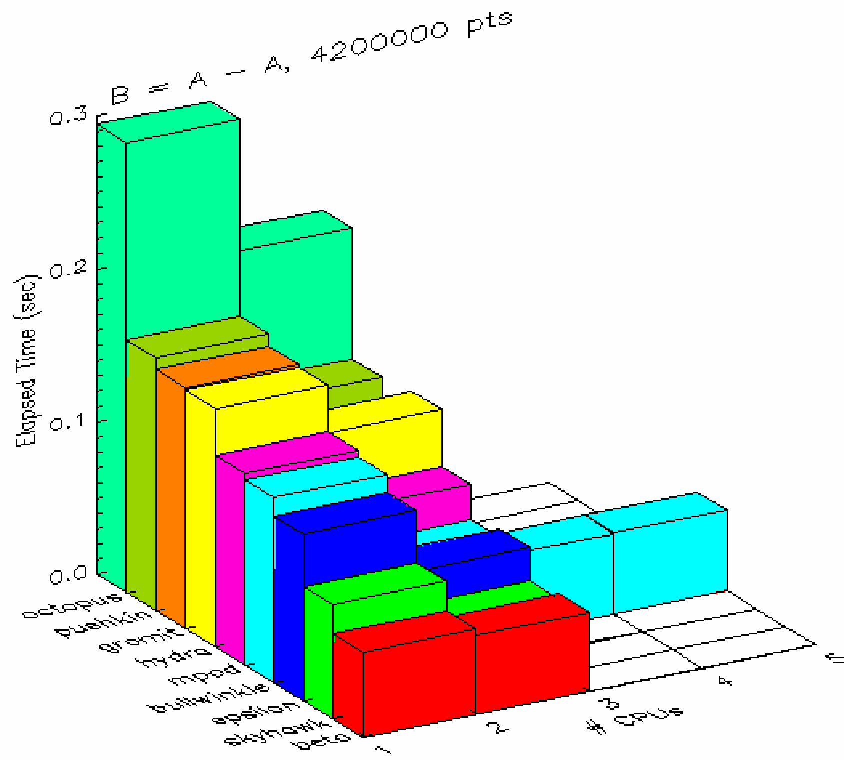
This appendix contains additional threading performance tests for various computations. For details on the specifications of systems, match the hostname label on the plot with the hostname descriptions in Chapter 3. For 2-dimensional computations, the input arrays are square matrices with dimensions equal to the square root of the number of elements (the closest integer). The maximum number of elements for each test varies depending on available system memory. For system comparison plots, the results are reported for arrays of 4.2 million elements.

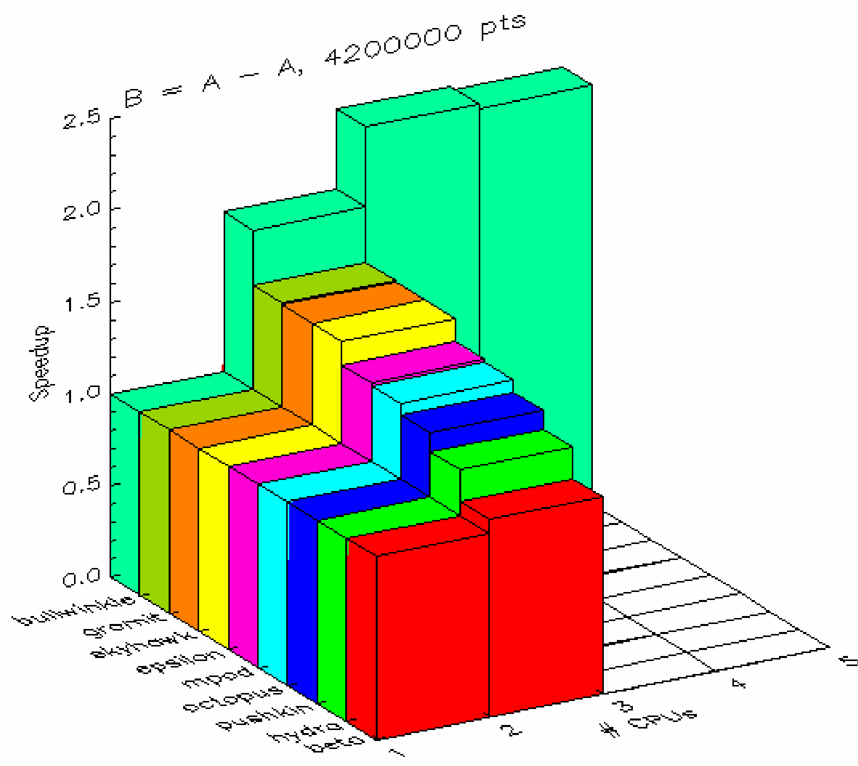
Note that for 1D and 2D FFT, the number of elements is constrained to powers of four to eliminate internal algorithm differences. For the system comparisons of FFT, the number of elements is 1,048,576. Additionally, the minimum number of elements (TPOOL_MIN_ELTS) is set to 1000 for all FFT runs.

The kernel for the 1D convolution is simply the three-element vector [1,2,1].

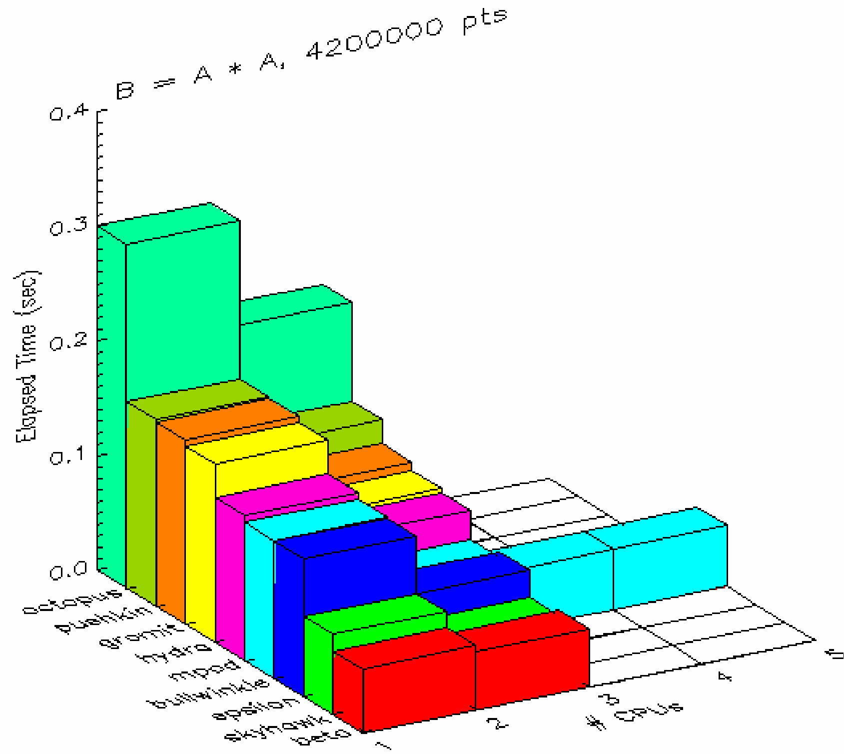
For 2D interpolation, PX and PY are identical and are equal to FINDGEN(SQRT(N))+0.5, where N is the number of elements.

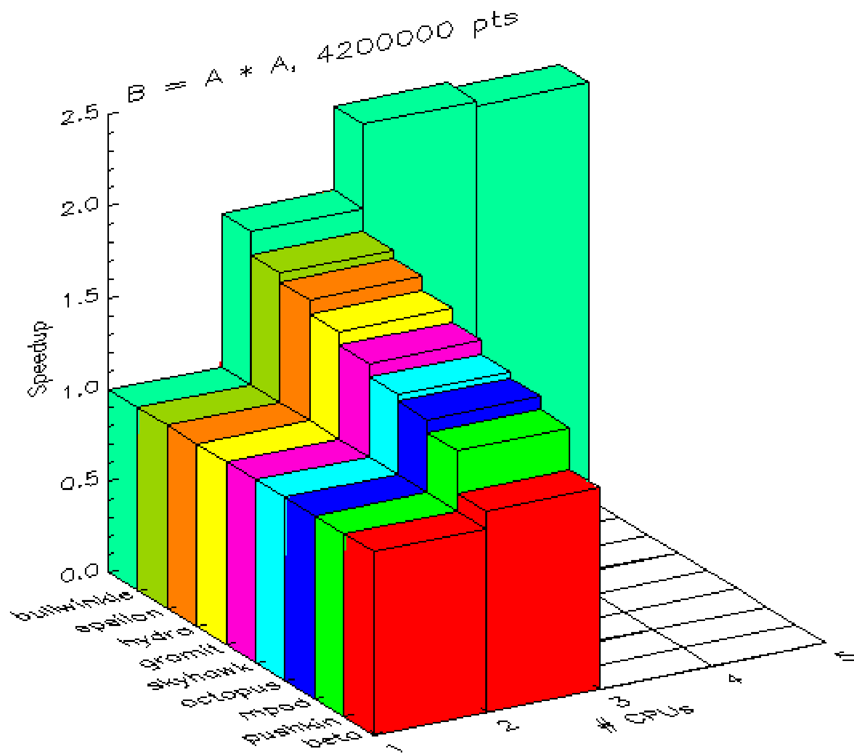
Floating Point Subtraction (B=A–A):



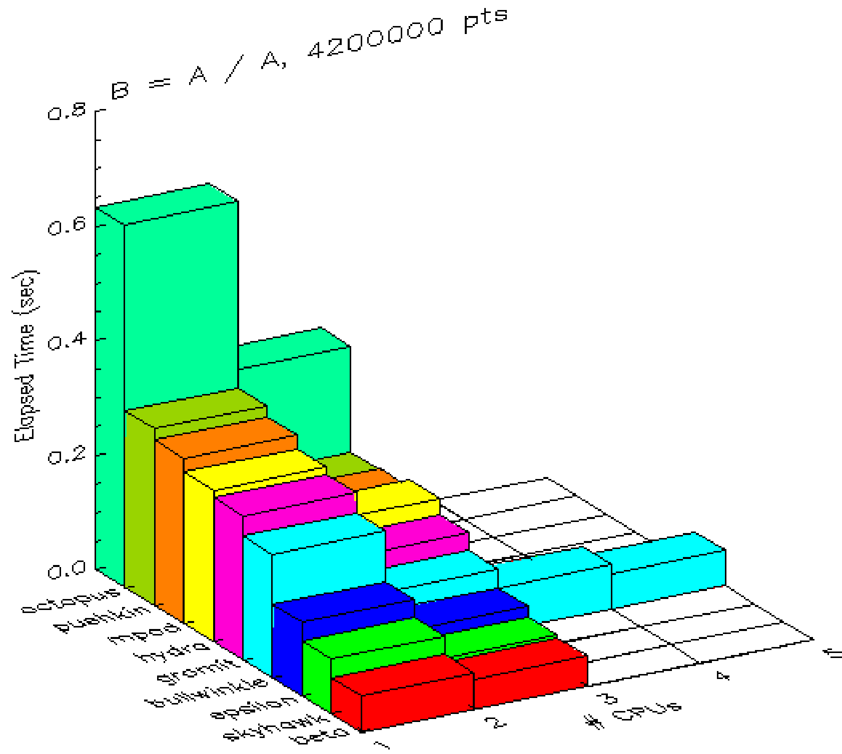


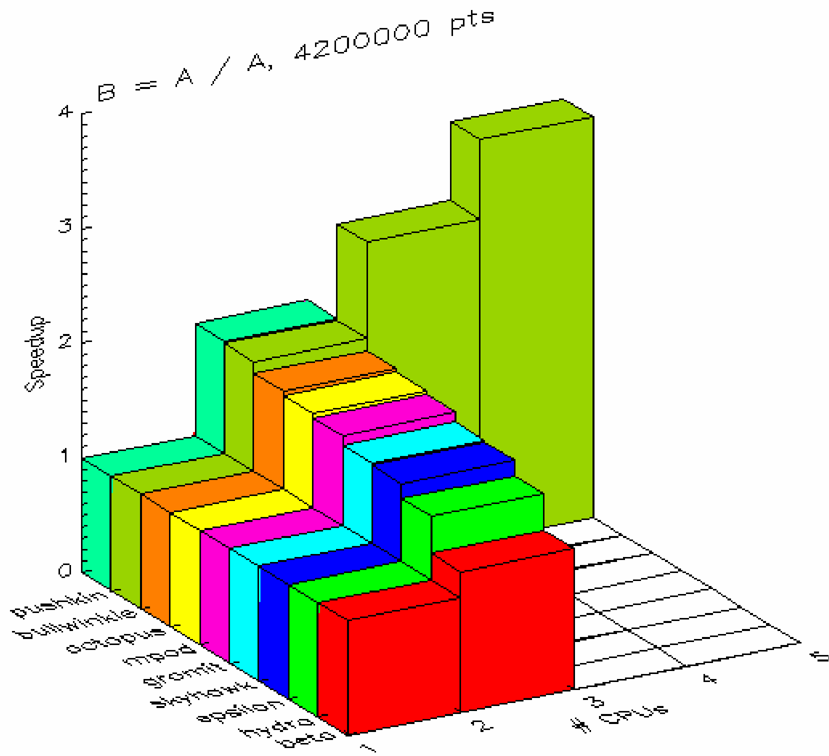
Floating Point Multiplication (B=A*A):



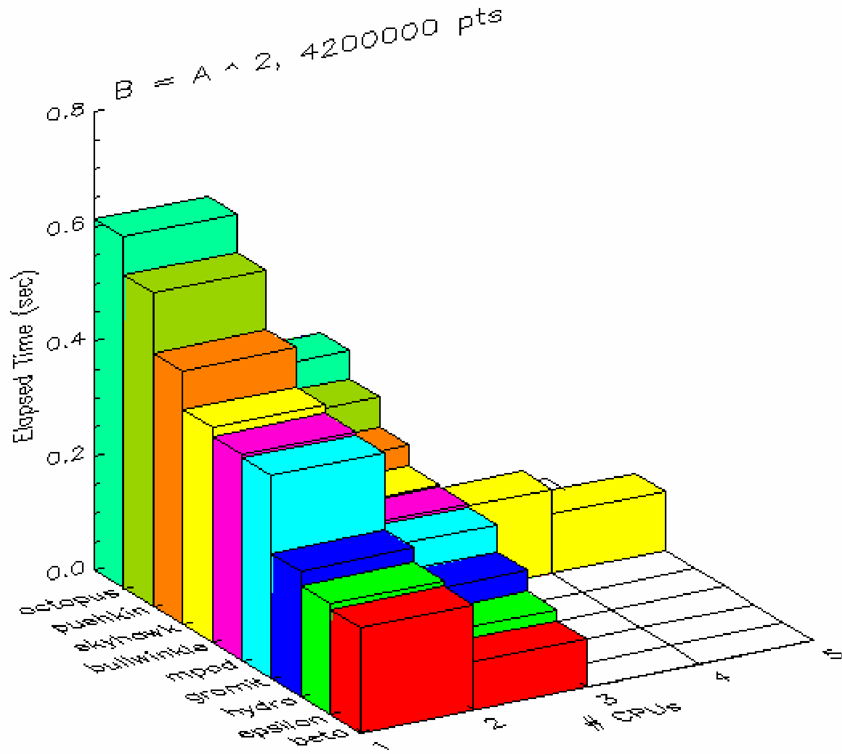


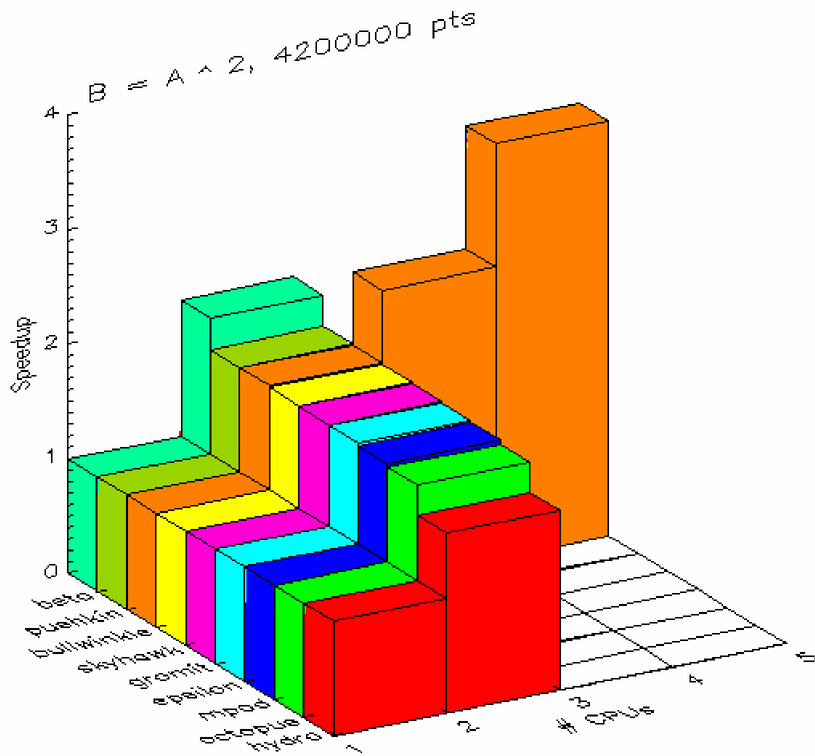
Floating Point Division (B=A/A):



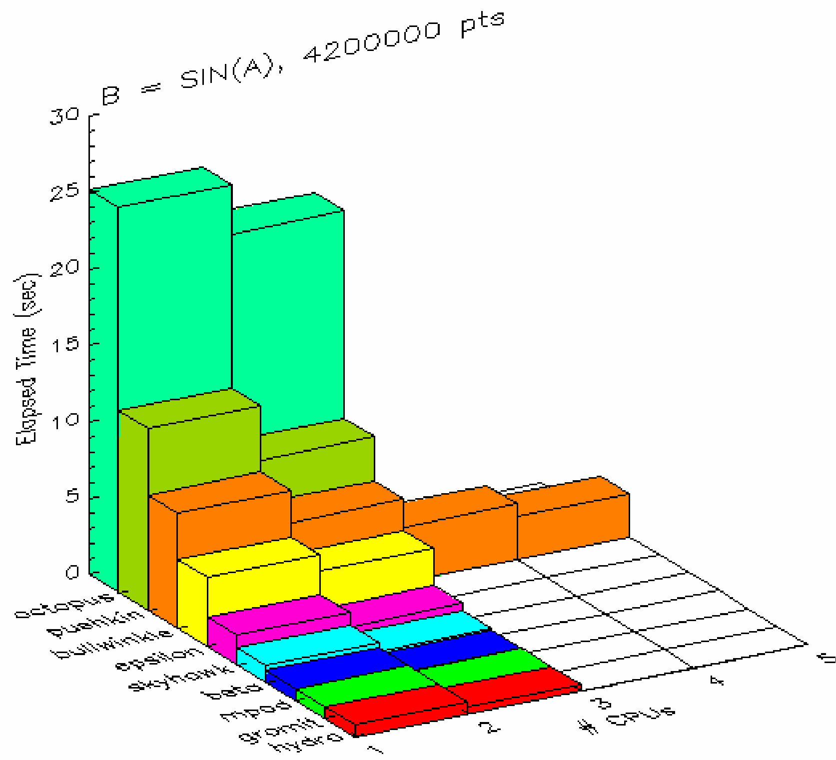


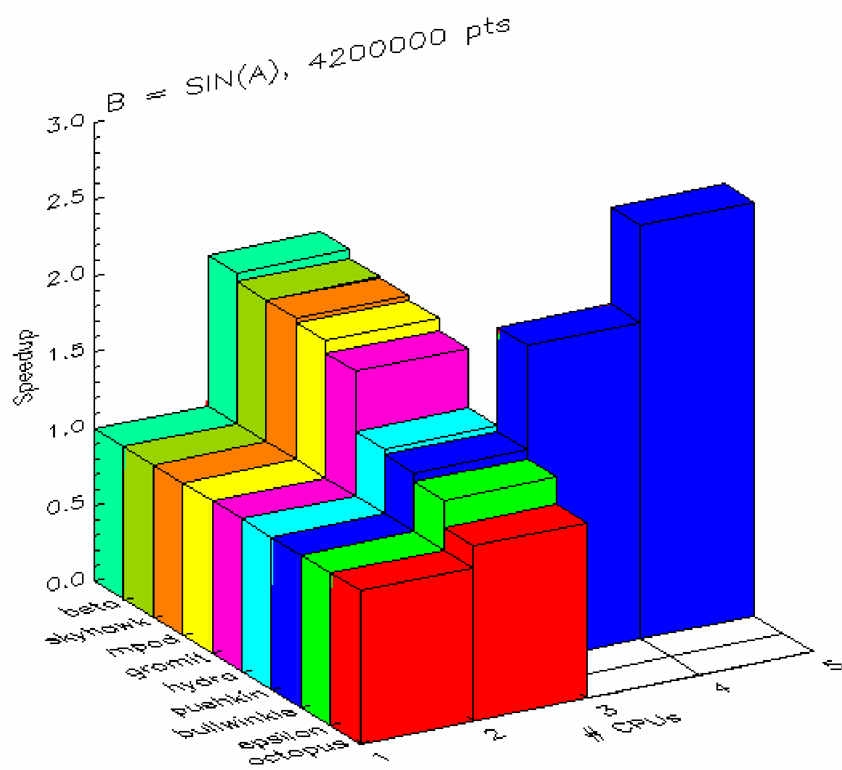
Floating Point Square (B=A^2):



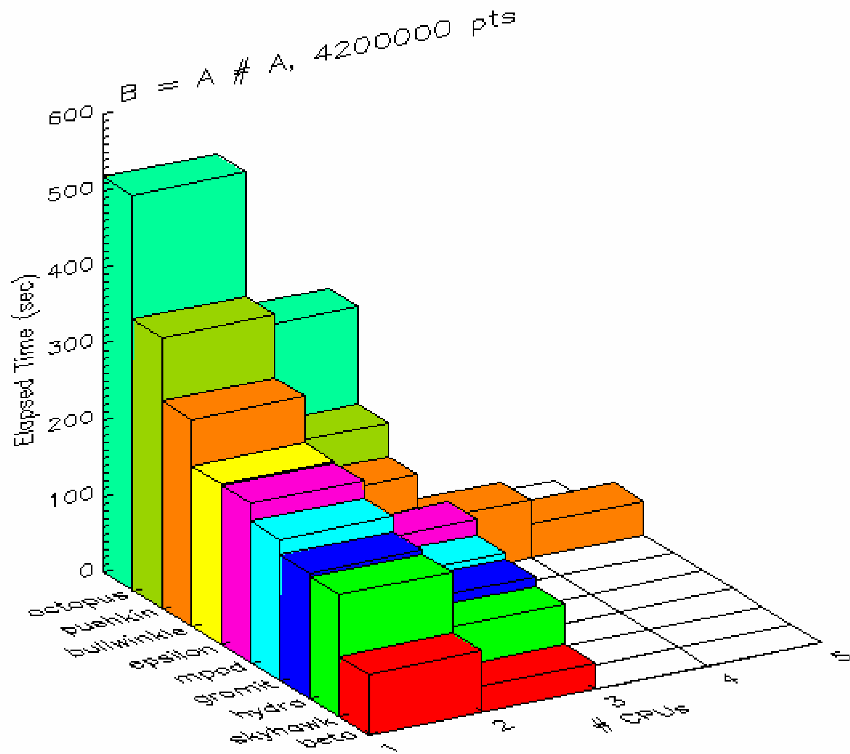


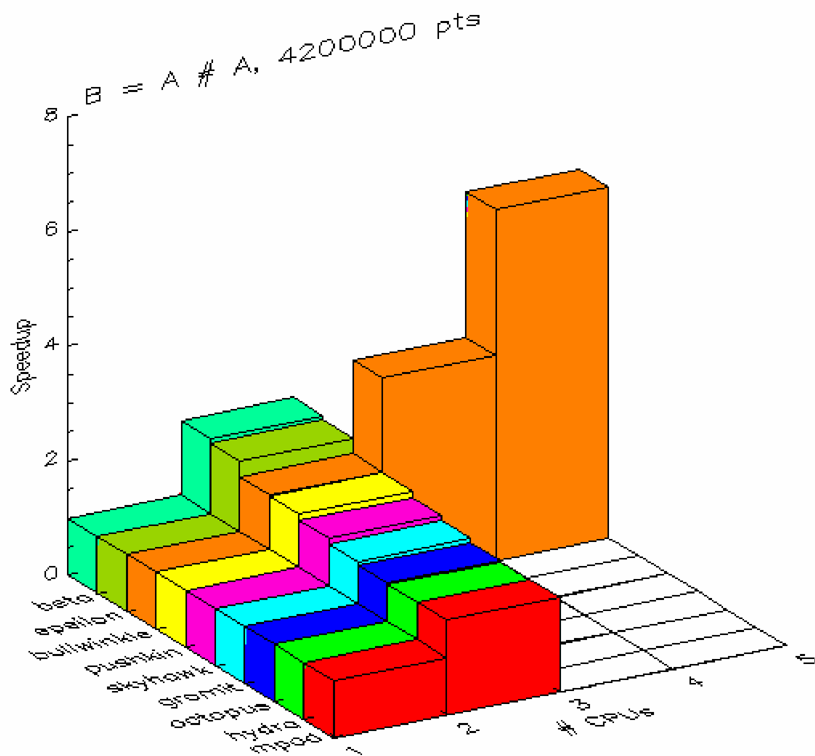
Floating Point Sine (B=SIN(A)):



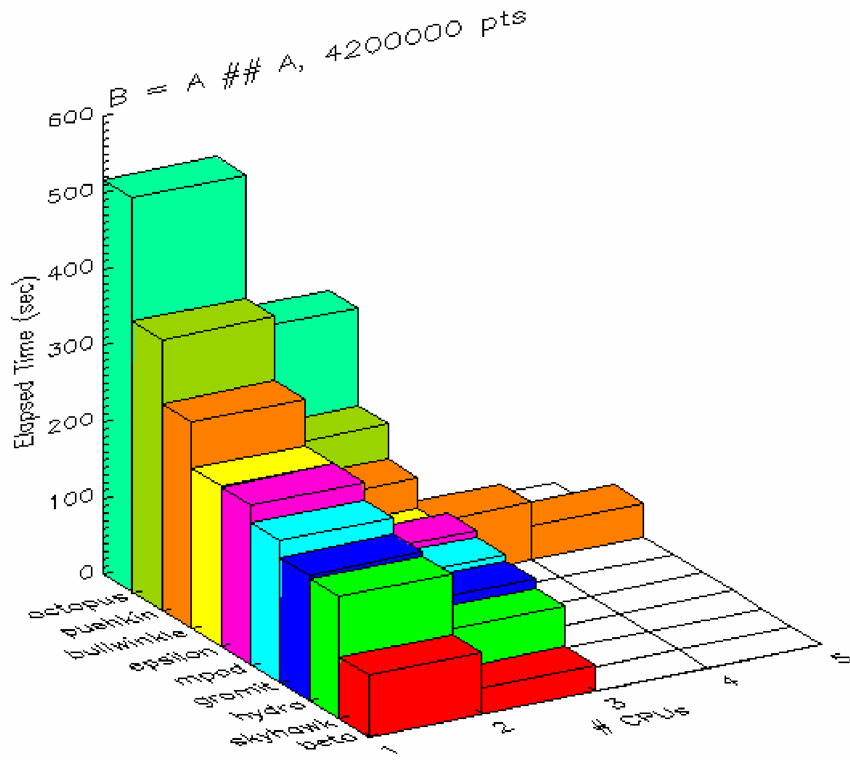


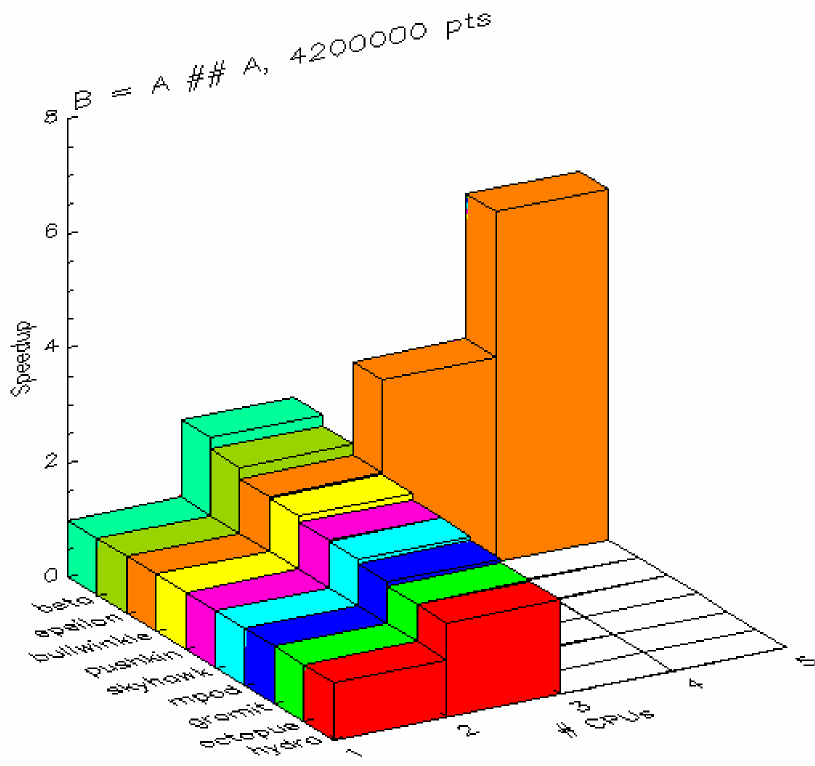
Floating Point Matrix Multiplication (B=A#A):



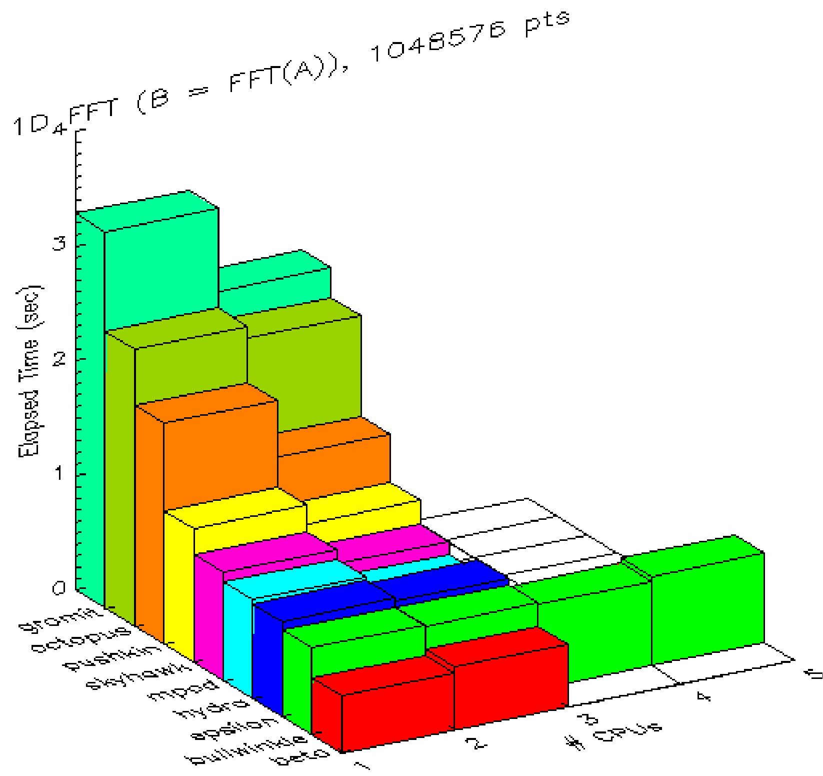


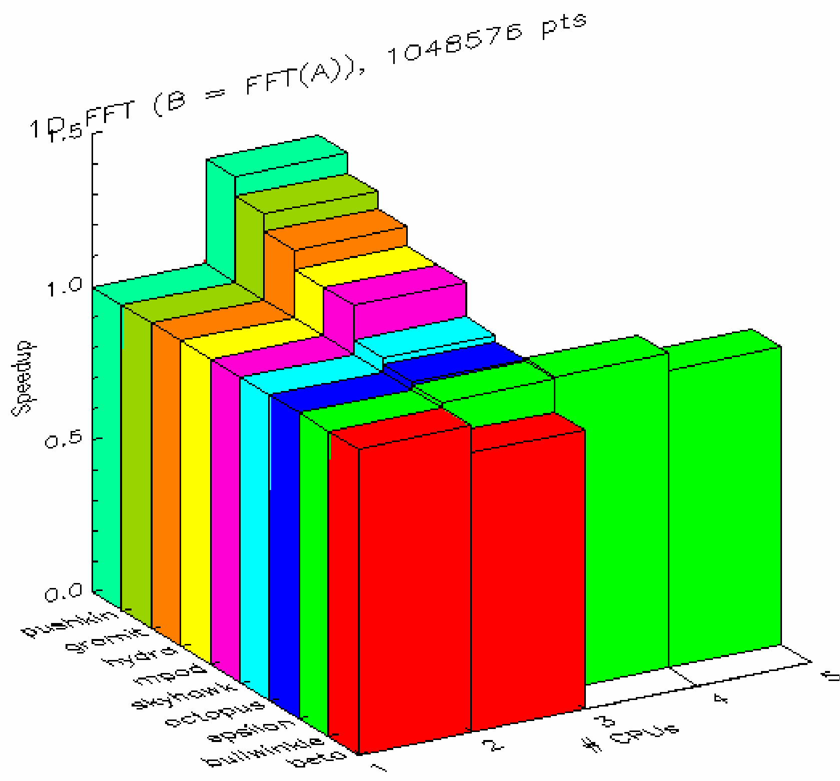
Floating Point Matrix Multiplication (B=A##A):



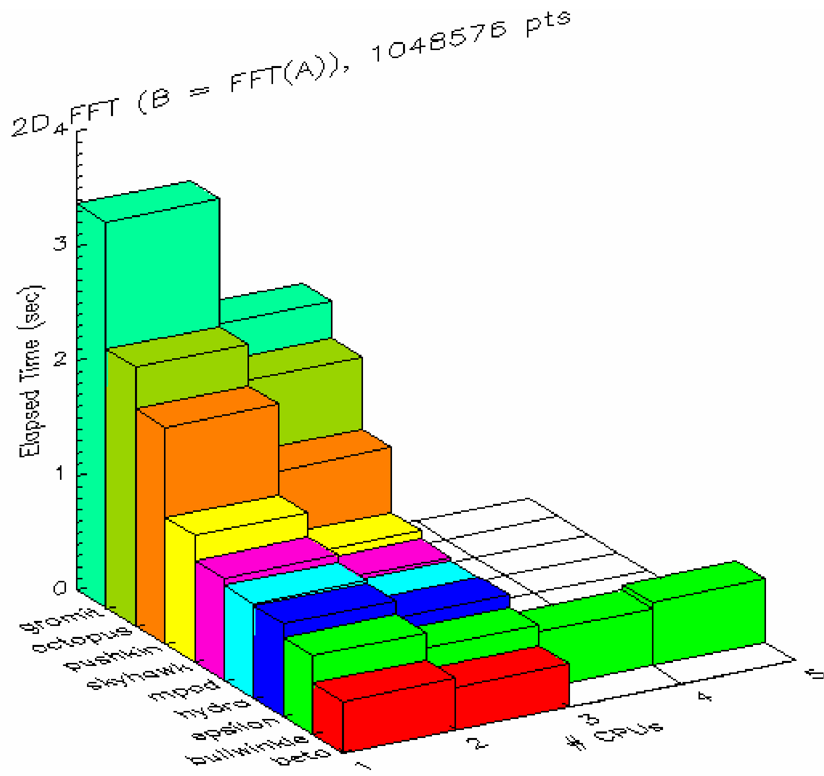


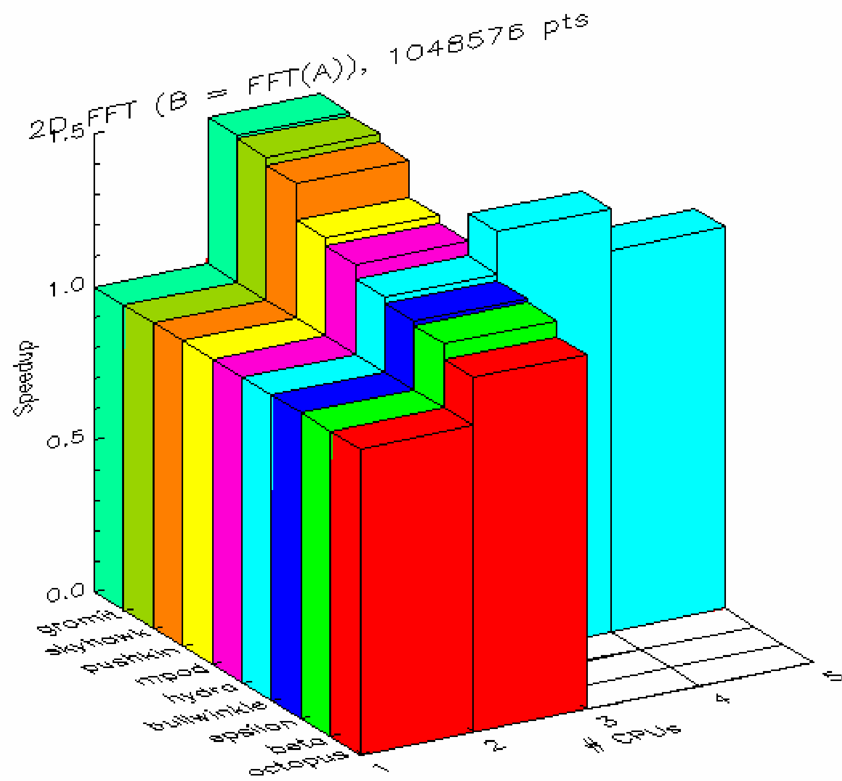
Floating Point 1-Dimensional FFT (B=FFT(A)):



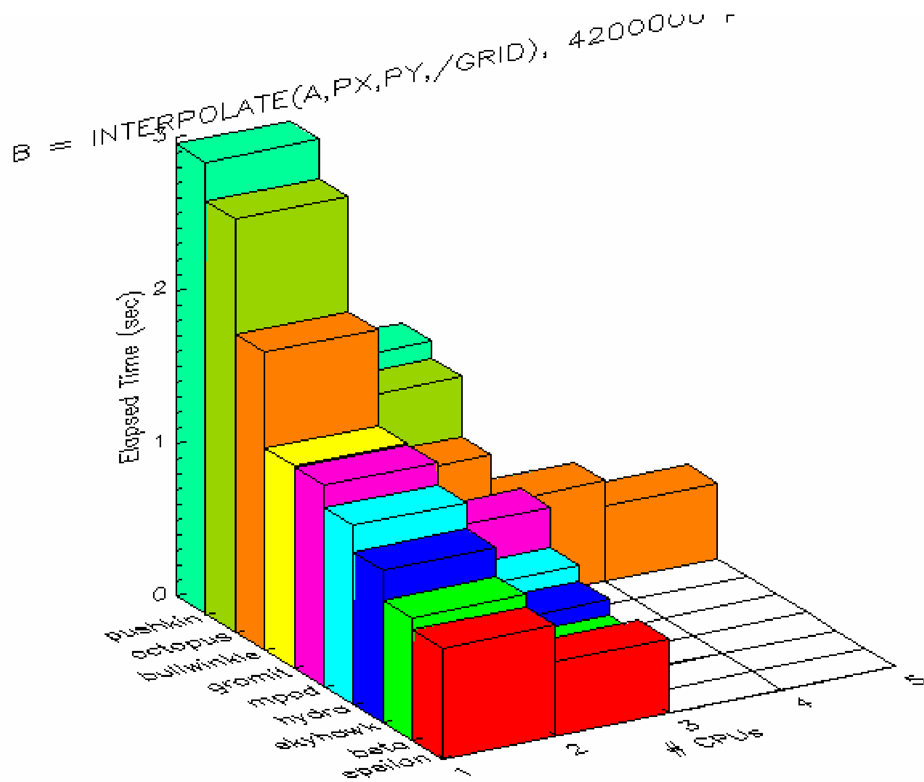


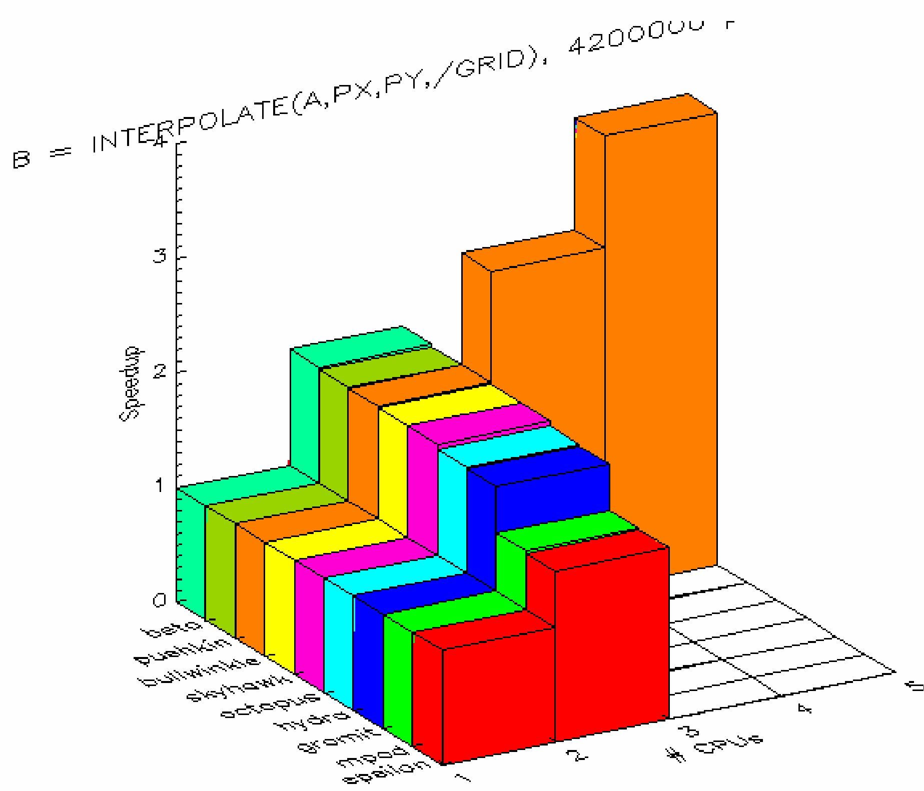
Floating Point 2-Dimensional FFT (B=FFT(A)):



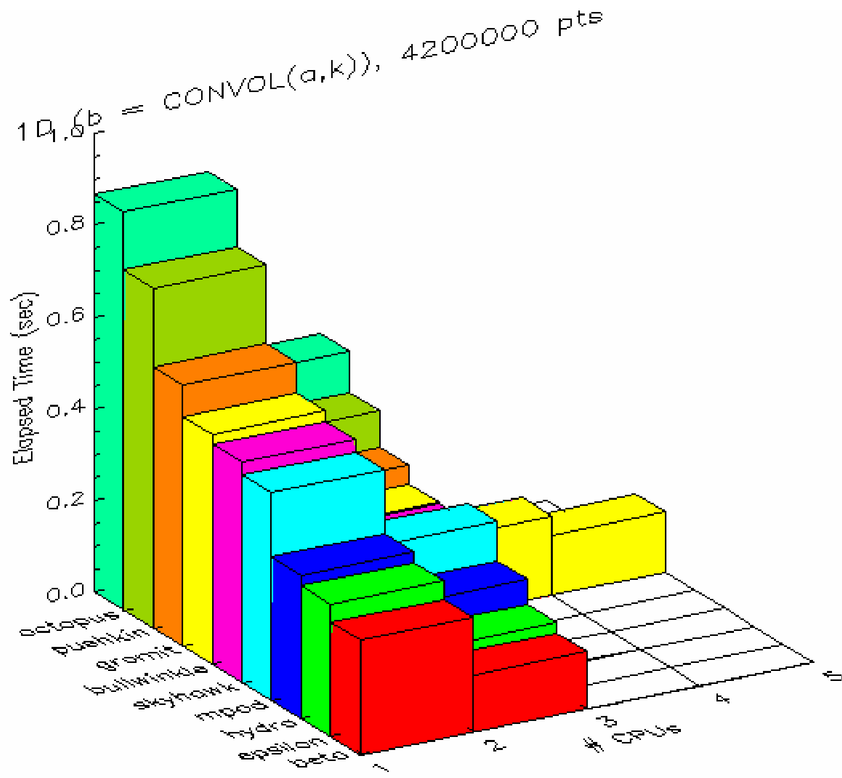


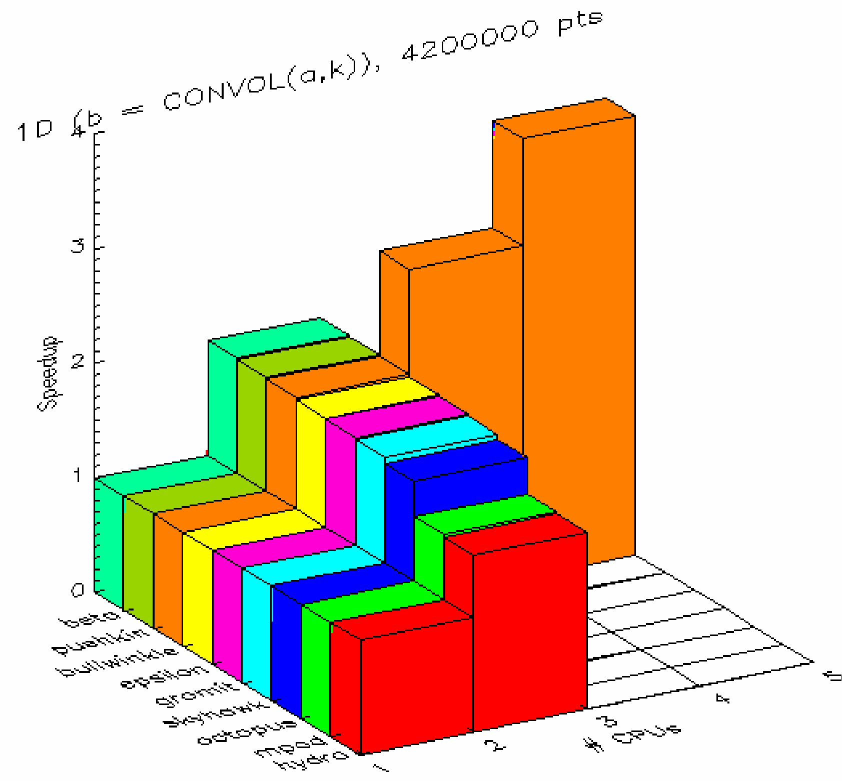
Floating Point 2-D Interpolation (B=INTERPOLATE(A,PX,PY,/GRID)):





Floating Point 1-Dimensional Convolution (B=CONVOL(A,K)):





Appendix C: Glossary

CPU (Central Processing Unit)

This is the core of the computer hardware that executes instructions. In a nutshell, multi-processing gives programs the illusion that more than one CPU is simultaneously available to it. With multi-processor hardware, this may actually be more than an illusion, and multiple processors do run simultaneously.

MP (Multi-Processor or Multi-Processing)

Multi-processor usually refers to hardware with multiple CPUs. Multi-Processing refers to either hardware or software depending on the context.

MT (Multi-Threaded)

This refers to software that makes use of multiple threads. If you run a MT program on a MP system, the potential exists for more than one thing to execute simultaneously. On non-MP hardware, the operating system gives the illusion that this is true by scheduling the threads to run on the single available processor.

Preemptive Multitasking

A multitasking operating system allows more than one program or thread to appear to run simultaneously by controlling the order and duration in which they execute. In a preemptive system, this process is managed by a combination of operating system and hardware support, and does not require the cooperation of the programs being run. This protects the system from poorly written programs, and ensures fair and responsive behavior. Serious operating systems are always both preemptive and multitasking. Once, smaller operating systems for desktop use were not, but this is rapidly changing.

Reentrant

Reentrant functions allow multiple concurrent invocations of themselves. To ensure that simultaneous function calls do not interfere with each other, the function must avoid writing to static data. Reentrancy is a property that is not specific to multi-threading. However, reentrancy is a prerequisite for being thread-safe, and it is not uncommon to see them being used interchangeably.

SMP (Symmetric Multi-Processing)

In an SMP system, the operating system is able to distribute its own operation across multiple CPUs simultaneously. Non-SMP systems can still support multi-processing, and many early MP systems were not SMP. However, the operating system rapidly becomes a bottleneck and prevents effective use of the hardware. Quality multi-processing usually requires an SMP operating system.

Thread-Safe

A thread-safe function is designed to allow multiple simultaneous calls from threads. Unlike reentrant functions, which achieve this through careful memory management, thread-safe functions handle multiple simultaneous calls through thread synchronization mechanisms.

Uniprocessor

A uniprocessor is a system with only a single CPU. Until recently, almost all computers were uniprocessors, but this is rapidly changing.

To learn more about IDL's functionality please visit www.ittvis.com/idl/ or contact your ITT-VIS sales representative at (303)-786-9900 <info@ittvis.com>.

© 2001-2007 ITT Visual Information Solutions
All rights reserved

The information contained in this document pertains to software products and services that are subject to the controls of the Export Administration Regulations (EAR). All products and generic services described have been classified as EAR99 under U.S. Export Control laws and regulations, and may be re-transferred to any destination other than those expressly prohibited by U.S. laws and regulations. The recipient is responsible for ensuring compliance to all applicable U.S. Export Control laws and regulations.