When are supercomputers worth the bother?

Jeffrey B. Mulligan

NASA Ames Research Center

Abstract: The availability of the UNIX operating system on large supercomputers has greatly simplified the importation of research software from workstations and minicomputers. Supercomputers generally offer much larger memory and storage areas than are available on smaller machines, which may make their use mandatory for some large applications. Realization of the full performance offered by vectorizing supercomputers such as the Cray C90 sometimes requires modification of existing code. The use of high-level interpreters for the development of applications is recommended to minimize the amount of compiled code that must be optimized.

For those with access, supercomputers offer high computing power and large capacity. These seductive attributes are sometimes offset by high up-front costs associated with adapting software. In this paper we will examine some of the tradeoffs in determining the benefit to be derived from using a supercomputer.

Recent years have seen a dramatic increase in the amount of computing power offered in desktop computers; it is hopeful (dare I say likely?) that future increases will continue at the

same phenomenal rate. The concept of what is a supercomputer is therefore somewhat slippery, when today's workstation would have been considered a supercomputer 10 years ago, and today's supercomputer may fit on a single chip ten years hence. For the purposes of the present discussion, the term "supercomputer" shall refer to a machine which is so expensive that it is not an option for an individual, or even a group of researchers, to purchase one for their exclusive use. Such machines therefore are found only in central computing centers which serve many users. As one would hope, their high cost does bring with it certain unique features which may make their use imperative in certain applications. For other applications, these machines may offer improved performance on a job which *could* be done on a smaller machine. And for other applications, the performance gain may be small --- which, if any work at all is required to make the program run on the new machine ("porting"), may not be worth the trouble.

This paper will focus on the Cray C90 here at Ames Research Center. The primary feature offered by this machine is "vectorization" hardware, by which a series of numerical operations is executed in parallel. To support large physical simulations, a correspondingly large amount of memory and disk storage is provided (see table 1). This system has the feature that, in general, code written for it looks similar to and will run on smaller machines; the compiler automatically determines when the machine should vectorize. In order to optimize performance, however, the programmer will want to read the report of the optimizations done by the compiler to insure that all desired vectorizations are indeed occurring. Certain codes may have to be rewritten to make this happen.

This situation is to be contrasted with another type of supercomputer, the massively parallel machine exemplified by the Connection Machine, which consists of a large number

(thousands) of relatively simple processors. The architecture of this machine is inherently unsuited to traditional linear programming languages, which has necessitated the introduction of language extensions to exploit the special features of these machines. The resulting programs can no longer be run as-is on a traditional machine, although emulators are available which allow development and debugging to be done on workstations.

Memory Capacities	
Machine	RAM
SUN Sparc2	64Mb
SGI Indigo2	64Mb
Cray C90	1600Mb

Table 1: Comparison of RAM complement of example workstations & Cray supercomputer. Numbers represent the actual memory installed in the test systems, which is not necessarily the maximum capacity of a fully-populated system.

Because supercomputer time is usually at a premium, interoperability (i.e., the ability to test programs on personal systems) is an important consideration. A major step in this direction was the adoption by Cray Research of UNIX (tm) as the basis of their UNICOS operating system, providing an environment similar to that of most workstations. An important aspect of a common operating system is the fact that users do not have to learn and remember two sets of commands for file manipulation and other housekeeping functions. This also makes porting applications from workstations up to the Cray simpler in most cases than porting down to DOS or Macintosh environments. Application programs that are easily ported allow development and debugging of a small-scale version to be done on a local

workstation, followed by a full-scale run on the supercomputer.

In keeping with this philosophy, and to minimize the amount of code that must be ported, it is often useful to concentrate coding efforts on the development of fairly general-purpose interpretive programs. Current commercially available examples include systems such at Matlab and Mathematica, but the idea can be extended to locally developed codes. The fundamental idea is to have a single copy of the often-used routines, and use a text interpreter to control the sequencing. Specific applications are then coded in the scripting language instead of the low-level programming language. In addition to producing a cleaner, more readable program which is easily understood and modified, these scripted applications are transparently portable *once the interpreter itself has been ported.* This philosophy has driven the development here at Ames of an interactive image-processing system known as QuIP (QUick Image Processing), which consists of a number of modules coded in C, including a text interpreter which controls the other functions.

Although text interpretation imposes additional overhead (compared to a compiled program), this overhead is often insignificant when the interpreted directives refer to the large, computationally intensive operations which are most likely to be run on a supercomputer in the first place. This is particularly true of image processing, where a simple statement requesting that two images be added together may result in a quarter of a million floating point operations.

The QuIP interpreter was used to produce the performance comparison given in table 2. A simple script was written which created 2 1K x 1K images, initialized them, and computed the pixelwise product 100 times. The time to process 1 million pixels (one image's worth) is obtained by dividing the running time by 100, and the reciprocal of this number is the

throughput in megaflops (floating point operations per second). (The numbers thus obtained are lower than the various manufacturers specifications because the inner loops of the QuIP routines are written to handle objects with non-contiguous data.)

Compute Throughput			
Machine	Elapsed time	CPU utilization	Throughput (Mflops)
SUN Sparc2	2:18	93%	0.7
SGI Indigo2	1:00	96%	1.6
Cray C90 (unvectorized)	0:24	88%	5
Cray C90 (vectorized)	0:03	37%	100

Table 2: Comparison of run times on a simple problem. CPU utilization is the percentage of the elapsed time during which the CPU was active on the test job; these numbers fall below 100% when the CPU has to wait for input/output activity (e.g. disk access) to complete, or because the CPU is shared with other users.

As can be seen from the last two lines of Table 2, while the unvectorized performance of the Cray C90 is nothing to sneeze at, it is in the same ballpark as the workstations (and indeed there are higher-end workstations on the market today that can duplicate this performance). The vectorized performance, on the other hand, is almost two orders of magnitude better than the workstations, and for many jobs may represent a worthwhile speedup. Unfortunately, this performance was not obtained when the current version of the software was recompiled on the Cray, and it may be instructive to see why not. The C code for the inner loop of the routine which does the multiplication was something like the following fragment:

```
multiply_vectors(p1,inc1,p2,inc2,p3,inc3,n)
float *p1, *p2, *p3;
int inc1, inc2, inc3, int n;
{
        while(n--){
                *p3 = *p1 * *p2;
                p1 += inc1;
                p2 += inc2;
                p3 += inc3;
        }
}
```

While this subroutine may be somewhat obscure to readers not familiar with pointers and C, the function is fairly simple: the addresses of three arrays of data are passed in the pointer variables p1, p2 and p3. The arrays are scanned, with the product of the entries pointed to by p1 and p2 being deposited in the location pointed to by p3. After each multiplication, the pointers are incremented by the arguments inc1, inc2 and inc3.

Why does this subroutine not vectorize as written? The key to successful vectorization is that the values of the variables in a given iteration must not depend on values computed in previous iterations. Now, in the above example, the value of pointer p1 is incremented by inc1, i.e. the new value is the value from the last iteration plus inc1. In this case, because the increment is not changed, it is possible to determine ahead of time what the values of the pointers will be in every iteration, but unfortunately the compiler is not smart enough to recognize this on its own! The following code fragment does exactly the same thing, but is vectorized by the compiler:

```
multiply_vectors(p1,inc1,p2,inc2,p3,inc3,n)
float *p1, *p2, *p3;
int inc1, inc2, inc3, int n;
{
        int i, i1=0, i2=0, i3=0;

        for(i=0;i<n;i++){
                p3[i3] = p1[i1] * p2[i2];
```

```
                i1 += inc1;
                i2 += inc2;
                i3 += inc3;
            }
        }
```

By the above reasoning, it might be thought that this code would not vectorize, since the values of i1, i2 and i3 are computed iteratively, like the values of the pointers in the first example. The compiler, however, happily vectorizes this version, evidently treating pointer variables differently from indices!? The confusing nature of this example illustrates the degree to which programming supercomputers is still something of a black art. Because of this, it is desirable to use subroutines from standard mathematical libraries (that have already been optimized and tested by someone else) whenever possible.

In summary, the large capacity of supercomputers make them possibly the only choice for simulating very large systems. For medium-sized problems, faster execution speed may not be adequate compensation for the extra programming effort required. The use of high-level interpretive systems is recommended both to make applications portable and minimize the amount of compiled code that must be maintained.