Does Your Computation Belong on a Supercomputer?

by

Clark Thomborson*

May 1991

Technical Report 91-04

University of Minnesota
Duluth

Computer Science

Does Your Computation Belong on a Supercomputer?

by

Clark Thomborson*

May 1991

Technical Report 91-04

Department of Computer Science
University of Minnesota
Duluth, Minnesota 55812
U.S.A.

*Supported in part by the National Science Foundation, through its Design, Tools ans Test Program under grant number MIP 87006139, and by the Minnesota Supercomputer Institute.

Abstract

Since a vector supercomputer costs about 1000 times as much as a workstation, you might expect it to run your program about 1000 times faster. This is highly unlikely: an "average" floating-point-intensive computation runs about 20 times faster on a Cray Y-MP than on a DECstation 3100, although speedups as high as 100 and as low as 5 are not uncommon. Speedups for other workstations are comparable, ranging between 3 and 25 for an IBM RS6000/320, and between 12 and 200 for a Sparcstation 1.

The purpose of this article is to help its readers decide whether their computations belong on a supercomputer instead of a workstation. We pose five questions in a self-evaluation format, touching on such concepts as the time-value of your results, the vectorizability of your code, the total time required for all your program runs, the sensitivity of your code to nonstandard floating-point arithmetic, and the amount of memory required.

We answer our own questions for the case of our experimental system for routing wires on a semicustom VLSI circuit, concluding that our application belongs on a workstation. We solve wire-routing problems with large and very sparse linear programs, so our application bears a superficial resemblance to the computational workload of Professor Robert Bixby. However, our self-evaluation format explains why Professor Bixby's workload, consisting of *very* large linear programs, belongs on a vector supercomputer.

When working with a program that is at, or reaching, the acceptable limits of runtime on a workstation, the following question arises naturally: Will the performance of my application increase when run on a supercomputer? In this article, we pose five additional questions to help you answer the original question for yourself. If you answer 'no' to more than one of our questions, it is probably not worthwhile for you to attempt porting your code to a vector supercomputer.

We devote one section of this article to each of our five questions:

1. Would you pay 30 times as much to solve your problem 30 times faster?
2. Can your code be easily vectorized?
3. Will your program be used enough to warrant hand-optimization?
4. Can you afford to retune your software's floating-point arithmetic?
5. Is your application a little too large for a workstation?

We developed our list of questions to save others from wasting time and energy, as we did, trying to port code from a workstation to a supercomputer.

From 1986 to 1989, we believed we had a application that was well-suited for running on a Cray supercomputer. We were running a series of experiments on a promising new approach for deciding how to route the microscopic wires on a semicustom integrated circuit known as a gate array. Our approach is to generate a fairly large linear program (832 constraints) for a small (12x15) gate array. The solution of the linear program indicates how wires might be placed into the Manhattan-like channels between the gates in the array [14].

Until February 1990, the cpu-intensive kernel of our wire-routing system was Roy E. Marsten's XMP package for solving linear programs. This package of FORTRAN routines is running on many different computers, ranging from PCs to supercomputers [10, page 3]. Recently, for the reasons outlined below, we replaced the XMP linear program solver with a new package called CPLEX [3].

Since our Sun-3 workstation took 100 seconds to solve our 832-constraint problem using the XMP library, we feared that our system would not be feasible for large gate arrays. To give you a sense of scale, a modern gate array such as the 1 μm CMOS TGC118 has 18,620 sites in a 133×140 array [16]. Gate arrays with as many as half a million sites are now being produced by some vendors.

Routing a 133×140 array with our method would result in a linear program approximately 100 times larger than our 12×15 example. Since linear programming by the simplex method (as in XMP and CPLEX) usually runs in $O(n^3)$ time, Sun-3 runtime on a 133×140 array would be roughly $(100)^3$

times as long as in our 12×15 example — perhaps 10^8 seconds, or about two years.

Since the Cray-2 typically runs at tens of MFLOPS, and our Sun-3 with a 68881 co-processor typically runs at tens of KFLOPS, we naively hoped to see our runtime decrease by a factor of 1000 after porting our program to a Cray-2. With the expected 1000-fold speedup, we could route a full TGC118 array in a day of Cray-2 runtime.

In retrospect, our expectation of a 1000-fold speedup was naive. On our code, we observed only a 20-fold speedup: a Cray-2 took 5 seconds to solve our 832-constraint benchmark, as opposed to 100 seconds on a Sun-3.

After modernizing both software and hardware early in 1990, we still see no reason to run our application on a supercomputer. Our new Sun-4 runs about 2.5 times faster than our old Sun-3, on the XMP code. We gain an additional factor of 15 by replacing the XMP code with the CPLEX linear program solver. With CPLEX, our Sun-4 workstation solves our 832-constraint benchmark in 2.5 seconds, a 40-fold improvement on our Sun-3 runtime with XMP. Using the $O(n^3)$ runtime extrapolation described above, we might now be able to route the wires on a 133×140 gate array in two or three weeks of Sun-4 runtime. Gate arrays with more than 10,000 or 20,000 sites could be handled by decomposition into smaller subproblems, along the lines suggested by Hu and Shing [8]. Using current workstation technology, then, along with some "algorithmic tuning," our wire-routing system now shows promise of economic feasibility.

It would be interesting for us to use the CPLEX code to solve our 832-constraint problem on a Cray, in order to complete the comparison. Unfortunately, our $500 CPLEX academic-use license gives us access only to Sun-4 object code. Still, we can make some speedup estimates on the basis of CPLEX promotional literature[3]: their code runs only 10 to 14 times faster on a Cray Y-MP than on a Sun-4, for five linear programming problems similar to our 832-constraint problem. Larger gate arrays would, of course, lead to much larger linear programs, probably increasing the speedup ratio somewhat. For the reasons outlined in Section 2, however, we do not expect to see a speedup exceeding 30, even on our largest problems.

In summary, none of our experiments or projections show more than a 30-fold speed advantage in moving our programs to a Cray. This improvement does not justify either the cost of the port or the relative expense of access to the Cray. Although we get our Cray cycles "for free" (via a straightforward grant-writing process), we are interested in proving the economic feasibility of our wire-routing method. Thus we view the 30-fold increase in actual cost of rapidly solving a wire-routing problem on a Cray, versus solving it 30 times more slowly on a workstation, as a deciding factor in our comparison.

In contrast, Professor Bixby of Rice University is quite satisfied to use a

Cray instead of a Sun-4 workstation to solve his linear programs. Economic feasibility is not of particular concern to him, but the 30 to 40-fold speedup he typically observes is enough to make a dramatic difference in the quantity and quality of his research results [2].

The discussion above has centered on speedup and cost ratios. These are important, but by no means the sole factors in considering whether to port code to a supercomputer. For example, Professor Bixby cites the limited memory capacity of a Sun-4 as a compelling reason for doing most of his research on a Cray [2]. Below, we treat memory capacity and several other important issues in a five-question framework for deciding "to port or not to port?"

1. Would you pay 30 times as much to solve your problem 30 times faster?

We have never observed more than a 20-fold speedup on our test runs, when comparing our workstation to a Cray. Your code may be more amenable to speedup than ours. Still, we doubt you will see as much as a 30-fold speedup, without an expensive and time-consuming recoding. In the following section, we argue for a 30-fold limit on speedups; if you accept this value for now, we can make some broad-brush economic arguments.

A vector supercomputer costs between five and twenty-five million dollars, depending on the number of processors. A uniprocessor workstation costs between five and twenty-five thousand dollars. Let's compare a Sparcstation 1, at about five thousand dollars, with a single-processor Cray Y-MP, at about five million dollars. This 1000-fold difference in purchase cost can be translated into a 1000-fold difference in operating cost, if one assumes that yearly operating costs are 30% to 50% of the purchase cost. Both workstations and supercomputers are functionally obsolete within five years, so that yearly depreciation and finance charges alone are 20% to 30% of the purchase cost. Yearly budgets for operating personnel, software, and hardware maintenance each add another 5% to 10% of the hardware purchase cost to the yearly operating expense. Thus an hour of cpu time on a supercomputer costs about $500, and an hour of workstation time costs about 1000 times less, or about $10 per day.

If you obtain a 30-fold speedup by using a supercomputer instead of a workstation, then you will pay $1000 \div 30 = 33.3$ times as much to solve each problem. In some applications, this is perfectly acceptable. Consider the problem of weather prediction. If you developed a code that enables a Sun workstation to predict tomorrow's weather in just 24 hours of runtime, your code would have no commercial value. However, if your code ran in just 1 hour on a Cray, you could sell your weather predictions for a tidy profit,

even after paying $500 per prediction for your Cray runtime.

Our advice, then, is for you to examine your the time-value of results in your application, to see if you are able to accept a 30-fold speedup at a 30-fold increase in cost.

In our case, we are not able to justify the increased cost of a Cray computation. We believe that our end-user population would not be willing to spend more than about $1000 for a solution to a wire-routing problem, nor would they be willing to wait more than a week for the results. One week ($70) of Sun-4 time will solve about twice as many problems as one thousand dollars (two hours) of Cray cpu time, even assuming an optimistic 30-fold speedup for our application.

Professor Bixby, in contrast, is often content to spend 150 seconds of Cray runtime, instead of 4500 seconds of Sun-4 runtime, to solve a difficult problem. When solving a large problem with integrality constraints, for example, he will solve a series of associated linear programming (LP) problems. The Cray turnaround is short enough that he can monitor the solution process: if the current batch of LPs is not being solved rapidly, he can abort it, adjust some algorithmic parameters, and start another batch. This adjustment can result in significant savings in total Cray runtime, compared with how long the Cray would have taken to run the same series of LPs without adjustment. On a slower machine, he says, "what usually happens is that nothing at all gets done in terms of experimenting with the LPs" [2]. In other words, the native 30- to 40-fold speedup of the Cray on large-scale LPs can be magnified significantly by on-line algorithmic adjustments to a batched series of problems. The same adjustments are, of course, available to workstation users, but it is much more disruptive to make once-per-hour adjustments to workstation codes throughout a working day, than it is to spend a concentrated block of time making similar adjustments to supercomputer codes.

2. Can your code be easily vectorized?

By following the advice in this section, you should be able to determine if you would obtain more than a 30-fold speedup if you moved your workstation code to a supercomputer.

In a nutshell, the difficulty with using a supercomputer efficiently is that code written for use on scalar machines, such as a workstation or a mainframe, will not take full advantage of the specialized functional units of a supercomputer. To gain the potential 100- to 1000-fold speed advantage of a vector supercomputer, programs must be vectorized [9, page 81].*

*Parallelization is another technique for acceleration, somewhat beyond the scope of this paper. If a program is efficiently parallelizable, one can run it on k processors in about

Vectorization can be accomplished in two ways.

The simplest approach is to compile your unmodified source code on a supercomputer's vectorizing compiler. The second, and much more expensive, method of vectorization is to employ a skilled programmer to rewrite portions of the source code. We discuss the use of vectorizing compilers in this section, deferring code restructuring to the next section.

Despite major advances in compiler technology, any of the following programming constructs hamper the vectorization of loops: recursion, subroutine calls, I/O statements, assigned GOTO's, certain nested if statements, GOTO's that exit the loop, backward transfers within the loop, and references to non-vectorized external functions [9, page 90]. Such constructs are common but relatively innocuous, outside of inner loops. When they appear in an inner loop, however, that loop will not be vectorizable.

Code that is not vectorizable will not run much faster on a supercomputer than on a workstation. For example, Professor John McCalpin of U. Delaware has developed a benchmark which is typical of numerical models in meteorology/oceanography/geophysical fluid dynamics. Running it on dozens of machines, he finds that a Cray Y/MP executes his code about three times faster than does an IBM RS/6000 Model 530, about seven times faster than a DECstation 3100, and about 20 times faster than a Sun 4/260 [11]. The code he wrote is completely vectorizable; however, it calls a library routine (for solving an elliptic equation on a 101x41 finite difference grid) that is not vectorizable. Since this library routine accounts for about 30% of the total cpu operations, we would only see a 3-to-1 speedup on a hypothetical supercomputer that executes vector code instantaneously but executes scalar code no faster than does a workstation.

As a matter of fact, modern workstations can execute scalar code about as fast as does a supercomputer, which is to say at 10 to 25 million instructions per second. Vector supercomputers are limited to this speed, whenever the operands and the machine instructions cannot be prefetched. On non-vectorizable code, the limiting factor is the 40 nanoseconds or so it takes to fetch an arbitrary operand from fast semiconductor memory (SRAM).

One way to decide if your code is a candidate for vectorization is to profile its execution time on your workstation. If your workstation spends less than 80% of its time executing the inner loops of your code, you will not see a large speedup if you move this code to a supercomputer. This prediction is based on classic work by G. Amdahl[1]; we applied a simplified

---

$1/k$-th the time, at about the same cost. Thus a cost- and time-sensitive user might try to modify their workstation code to run on several workstations in parallel, although this is generally more difficult than acheiving parallelism on a supercomputer. Supercomputer parallelism is generally attempted only after efficient vectorization is obtained, returning us to the main question of this paper: Can your computation make efficient use of a single processor on a supercomputer?

version of Amdahl's Law a couple of paragraphs ago to put a 3-to-1 speedup limit on McCalpin's benchmark.

The code outside your inner loops is unlikely to be accelerated much, due to its use of non-vectorizable constructs. If your non-inner loop code were, in fact, accelerated by an (optimistic) factor of six, and if your inner loops were executed instantaneously, your total speedup can be no more than $6/(20\%) = 30$, if 20% of your workstation runtime is due to non-inner loop computations.

Amdahl's law is a good "first cut" at predicting speedups, as evidenced by Professor Mohan Ramamurthy's data on six geophysical fluid dynamics models. One model is 95% scalar, yielding only a 3-, 5-, and 12-fold speedup of a Cray Y/MP over an IBM RS6000/320, a DECstation 3100, and a Sparcstation 1. Another is highly vectorizable, yielding speedups of 25-, 100-, and 200-fold on the same machines. The geometric mean of the speedups observed on Ramamurthy's six models is 10-, 20-, and 50-fold [15].

You can estimate the percentage of cpu operations occuring in the inner loop of your code, by running a statement-counting utility known as a "profiler." If your inner loop consists of a few floating-point operations, simple array indexing, and no data-conditional branches, it will undoubtedly be vectorizable. In this case, you may apply the simplified version of Amdahl's law, as explained above.

More refined estimates are sometimes possible, by informed examination of your source code. The idea is to find speedup data for a loop that is similar to your inner loop, in a book[9] or some other published source[5, 12, 6]. The iteration count of the loop is a critical parameter[7]. We present some illustrative data below.

On a Cray Y-MP in single processor mode, the loops A and B run at about 5 MFLOPS when the iteration count n is 10, at about 50 MFLOPS when $n = 100$, and at about 200 MFLOPS when $n = 1000$. Loops A and B are therefore amenable to vectorization when $n > 100$. In contrast, Loop C's indirect addressing limits it to 33 MFLOPS, even for large n.

```
Loop A:        do 1 i=1,n
           1   s=s+a(i)*b(i)
Loop B:        do 2 i=2,n
               do 2 j=1,i-1
           2   a(i)=a(i)+b(i,j)*a(i-j)
Loop C:        do 3 i=1,n
           3   a(ia(i)) = b(ia(i)) + c(ia(i))
```

By way of contrast, all three loops run at 1 to 3 MFLOPS on a Sun-4 workstation, for any $n > 2$.

Summarizing our discussion of speedup analysis, we assert that unless you have simple arithmetic statements in your inner loops, unless your vector lengths are longer than 100, unless you have no indirect addressing in your inner loops, and unless your inner loops account for at least 80% of your program execution time, you will not observe a speedup greater than 30 when moving from a workstation to a supercomputer.

You may be uninterested in learning enough about vectorizing compilers to estimate the speedup possible on your code. As a simpler alternative, we suggest you compile your code on a supercomputer, then run it on a representative set of test data. Most supercomputer centers are willing to let a potential client make a test run of this form, with little bureaucratic overhead.

We made test runs on several different workstations and supercomputers, timing how long each takes to solve our 832-constraint linear program with the XMP code. Here are our results. A Cray X-MP acheived

- a 30-fold speedup over a Sun-3,
- a 12-fold speedup over a Sun-4,
- a 6-fold speedup over a DECstation 3100, and
- a 1.5-fold speedup over a Cray-2.

The actual runtimes were

- 101 seconds on our Sun-3/160,
- 42.1 seconds on our Sun-4/110,
- 19.9 seconds on a DECstation 3100,
- 5.1 seconds on a Cray-2, and
- 3.4 seconds on a Cray X-MP.[†]

Leaving the obsolescent Sun-3 out of the comparison, the largest speedup observed when moving from a workstation to a Cray is 12. Thus, the $10-per-day workstations will solve the same set of problems about 80 times

[†]All workstations had floating point chips and at least 8 megabytes of memory. The Sun-3/160 is rated at 2 MIPS; it was running Sun UNIX 4.2 release 3.5. The Sun-4/110 is rated at 7 MIPS; it was running SunOS Unix 4.0. The DECstation 3100 is rated at 14 MIPS; it was running an early version of its operating system, ULTRIX-32 ver 2.0 rev 7.0. The Cray-2 is rated at 1800 MFLOPS (peak); it has 4 processors and 4 gigabytes of primary memory; it was running UNICOS 4.0 at the time of our experiments. The Cray X-MP has 4 processors, 128 Mbytes of primary memory, and 1 gigabyte of semiconductor secondary memory; it was running UNICOS 5.0. We used the CFT77 compiler on the Crays, and the FORTRAN–77 compiler supplied by each workstation's manufacturer.

more cheaply than a $500-per-hour supercomputer, albeit 12 times more slowly.

We made more than a dozen runs on the Cray-2, trying all combinations of the compiler flags that affect vectorization. (We also tried a pre-release version of the parallelizing front-end to the compiler, but it produced an error message on our code.) None had a significant effect on the runtime of the XMP source code, for our test data. This implies that the code is not vectorizable. Upon further investigation, we found that the inner loop of the XMP code contained dozens of lines of source code, several data-conditional branches, and iterates, on average, just 1.8 times per entry. This is sufficient explanation for its non-vectorizability. It seems unlikely that any compiler would be able to parallelize this code, either.

As mentioned in the introduction, we recently switched to a better linear program solver, CPLEX. It is superior not only in runtime, but in the number of iterations and in the number of integers in its solution vector.

When we mounted CPLEX on the Sun-4, we found it could solve our 832-constraint problem in just 2.5 seconds. This is twice as fast as the Cray-2 running the inferior XMP linear program package! Unfortunately, a license to run CPLEX on a supercomputer is very expensive, so we are unable to directly measure its speedup, on our problems, on a supercomputer. However, according to its supplier's data, CPLEX runs only 10 to 14 times faster on a Cray Y-MP than it does on a Sun-4, on five "netlib" LPs (*viz.* scagr25, seba, scsd8, nesm, and ship121) consisting of 1,554 to 16,170 constraints with 5 to 20 nonzeros per row. We would expect to see a similar 10- to 14-fold speedup on our 832-constraint problem.

Due to the locality of wiring in integrated circuits, we expect our LPs to become sparser as the circuits get larger: perhaps 20,000 rows, 40,000 columns, and 200,000 non-zeros. We have uncovered little data on the speedups to be expected on such extremely sparse LPs, although we would expect our current 10- to 14-fold speedups to increase somewhat with the scale of the problems we solve.

Professor Bixby has kindly supplied us with timing data on a new, "far better vectorized" version of CPLEX, for ten large linear programs with 50 to 1200 nonzeros per row. The speedups ranged between 15 to 66 on a Cray Y-MP versus a Sun 4/390, when the same solution algorithm is used for both machines. (Using the default solution algorithm for each machine's version of CPLEX, the speedup range is 11 to 96, thereby illustrating how important algorithmic adjustment can be, in such problems.) Prof. Bixby says he typically sees speedup ratios in the range of 30 to 40.

We note, in passing, that workstation performance is increasing at a rapid rate, and is expected to do so for some time. The recently introduced DECstation 3100 and IBM RS6000 are several times faster than a Sun-4.

Also, it seems clear that the next generation of workstations will have vectorized floating point units. Already, one can buy a floating-point accelerator board for a Sun workstation, boosting its peak performance to 80 MFLOPS. These developments will greatly reduce the maximum speedups observed when moving from a workstation to a supercomputer. Paradoxically, they may also expand the use of supercomputers, since code optimized for a vectorized workstation will probably run efficiently on a supercomputer.

3. Will your program be used enough to warrant hand-optimization?

As noted in the preceding section, global compiler directives are not often sufficient to give much (if any) speedup. Hand-optimization is necessary for efficient use of the functional units of a supercomputer, in order to approach its full speed advantage over a scalar processor.

The time required to hand-optimize a program will depend upon the size of the utilized code [9, pages 6–7], the style in which it was written, and the programmer's familiarity with it. By style we mean, "Does the program contain good comments and descriptive variables?"

Of course, the expense of programmer-mediated optimization must be offset with the savings gained from the decrease in execution time. In the worst case, it will take months or years to hire and train programmers to rewrite, document, and test a few thousand lines of code. In the best case, one's supercomputer center consultants may be able to point out a simple modification to your inner loop, achieving a factor-of-ten speedup in a matter of days.

In our case, examination of the XMP source code showed its inner loop to be a prime example of "dusty deck" FORTRAN. No obvious fixes suggest themselves. The development of an efficient code for solving LPs on a supercomputer is a major development task, one that I have neither the expertise nor the interest in pursuing.

This brings us to our final bit of advice: before rewriting your own code, we strongly recommend you make a thorough search for similar code that has already been vectorized and/or parallelized. Should this search be successful, you stand to save a lot of time, money, and aggravation.

4. Can you afford to retune your software's floating-point arithmetic?

Our fourth question is pointed at the profound differences between, and general inferiority of, floating-point arithmetic on a Cray versus that on a

workstation. All modern workstations follow the IEEE 754 floating-point standard. Supercomputers do not.

In general, you should expect to encounter little difficulty when porting your code from one IEEE-standard workstation to another. If, however, your code is at all sensitive to floating-point roundoff errors, you should be prepared to "retune" and revalidate your code upon moving it to a supercomputer.

The Cray floating point storage format has 1 sign bit, 15 exponent bits, and 48 mantissa bits. Floating point numbers have the most significant bit in the leftmost mantissa bit, giving 48 bits of precision [4]. The IEEE-standard double-precision storage format has 1 sign bit, 11 exponent bits, and 52 mantissa bits. Here, the most significant bit is implied, giving the user 53 bits of precision [13, page 1-9], fully 5 bits more storage precision than is available on the Cray-2. Furthermore, the IEEE standard specifies a roundoff algorithm to be employed when storing a floating-point number; Crays do some form of truncation.

The Cray multiplication algorithm uses 56 bits of precision [4]. This is significantly less than the intermediate format of the IEEE standard, which has 64 bits of precision [13, page 1-9]. The 56-bit Cray result is not an intermediate format in the IEEE sense, since it is not available after the multiplication is complete. Thus, for example, inner-product accumulations are done with at most 48 bits of precision on the Cray, whereas the IEEE standard provides 64 bits.

In computations where precision is not crucial, workstation users may employ the single-precision IEEE-standard floating point storage format. In this format, a floating point number occupies only 32 bits of storage, but its range and precision are limited. On most modern workstations, computations on single-precision numbers proceed at about the same speed as double-precision numbers, unless bandwidth to and from workstation memory is a bottleneck.

We note, in passing, that very-high-precision floating point arithmetic is available on some workstations (such as the DECStation 3100) and the Cray supercomputers, for users willing to pay a substantial time penalty. It takes four to ten times as long to run a program with about 100 bits of precision, as opposed to the 48- to 64-bit precision of the standard floating point arithmetic.

In summary, although there is a slight time penalty for *storing* a floating point number in double precision on a workstation, floating point arithmetic on a workstation is always *performed* at a higher level of precision than on a Cray.

In addition to the precision advantage outlined above, the IEEE standard provides a straightforward method for handling exceptional operations, such

as dividing by 0 or taking the square root of a negative number. These exceptional operations yield results that, when printed, are easily understood. The alternative method of handling exceptions, seen on Crays (and on older mainframes) is an easily-lost or misinterpreted error message, or, possibly, an abnormal termination of your job. The latter is irritating, to say the least, when the exceptional operation would not have affected the major results of your run.

In our application, we observed that the XMP linear program package was unable to solve a 1500-constraint problem when it was compiled and run on a Cray-2. The identical XMP source code was able to solve this problem when compiled and run on the Sun-3, the Sun-4, and the DECstation 3100. We surmise that the loop termination conditions in the XMP package were tuned for IEEE-standard arithmetic, and would require some modification to work adequately on the Cray-2. If we were still interested in porting XMP to a Cray, we would contact its supplier for advice on modifying the code.

We conclude that the peculiarities and relative imprecision of supercomputer floating-point arithmetic could pose severe difficulties when porting code from a workstation to a supercomputer.

5. Is your application a little too large for a workstation?

Most workstations have ten or twenty megabytes of primary memory. A fully-configured state-of-the-art workstation such as a DECstation 3100 or an IBM RS6000 can have up to 256 megabytes of primary memory. If you need more main memory than this, you should consider moving to a Cray-2, which can have up to 16 gigabytes of primary memory. However, if your code makes random access to more than 16 gigabytes, you belong on a workstation or possibly a mainframe. The reason is that an inexpensive CPU can keep its disks busy much more cost-effectively than can a Cray.

The importance of matching application size to primary memory space can not be overstated. If your program frequently accesses a data structure that is larger than its main memory, its runtime will be limited by your computer's disk transfer time. A slowdown for this reason is called "disk thrashing." In the worst case, your program might make random accesses to single words in a data structure stored on its disk. If this occurs, your workstation could run 100,000 times more slowly than if its data structure fit into its primary memory: a random access to fast semiconductor RAM takes about 100 nanoseconds, while a random access to a fast disk takes about 10 milliseconds.

On a supercomputer, disk thrashing has more severe economic consequences than it does on a workstation. A $500-per-hour Cray Y-MP can do

only a few more disk accesses per second than can a $10-per-day workstation. Also, the cost of a disk access on a workstation can be mitigated by multiprogramming, a technique whereby another task (or another user) can use the workstation's CPU(s) while the first task is awaiting disk service. It takes only a millisecond or two to switch tasks. By contrast, a multiprogrammed Cray processor also takes a millisecond or two to switch tasks — but this millisecond of "wasted CPU time" costs 1000 times as much as on a workstation!

The Cray X-MP and Y-MP architectures include a large semiconductor "secondary memory" designed to allow rapid block-level accesses to a 1000-megabyte dataset. There is thus some hope of efficiently running large programs on these machines, even though they lack a large random-access memory.

In order to estimate your program's space requirements, we suggest asking your computation center's consulting staff for help. If you give them a running copy of your code and a sample dataset, they should be able to measure (or tell you how to measure) your program's use of the system's disk drives and virtual memory. A bit of analysis and examination of source code documentation will be in order as well, in order to extrapolate how your program would behave on a worst-case set of input data.

Alternatively, you might try to calculate the maximum total size of your program's data and code area. You can then use this as a worst-case estimate of your program's memory requirements. Most programs will not thrash if limited to a small fraction of their total memory area, since at any given time, they will access only a small part of their data and code. The actively-referenced part of a program's data and code space is called its "working set."

In our application, a linear program with 1000 constraints uses 0.5 megabytes of memory for data storage on the workstations, or 0.7 megabytes of memory on the Cray-2. The difference between the Cray-2 and the workstation is a result of the use of 4-byte integers on the workstations, as opposed to the 8-byte Cray-2 integers. On both machines, the space required for the program object code quickly becomes irrelevant as the problem size increases.

Since we are using linear programming code based on sparse matrices, our working set will grow with the number of non-zeros in the constraint matrix. In our problems, we expect the number of non-zeros per row to be essentially independent of the number of constraints. We thus extrapolate linearly, finding that 40 megabytes of working set should be enough to handle the 80,000-constraint linear program arising from using our method to route the wires in a modern gate array. Thus a memory-rich workstation should be able to solve a wire routing problem for even the largest current gate

arrays. Gate arrays are expected to become larger in the future, but so are workstation memories.

We conclude that our program's working set is not large enough to require a supercomputer.

Professor Bixby's answer to this section's question is "yes." Many of his most interesting problems require 70 to 100 megabytes of data space. The largest workstation at Bixby's immediate disposal has only 32 megabytes. This might be sufficient reason to keep him on the Cray for most of his computations. It is not always possible to predict memory usage before a computation begins, and it is very irritating to see your program abort after several hours (or several days!) with no usable output, due to lack of memory space.

We may conclude that vector supercomputers are appropriate for anyone who, like Professor Bixby, is exploring the contemporary limits of feasible floating-point computability.

6. Summary

We have posed five questions to help you decide whether or not to port your application to a vector supercomputer:

1. Would you pay 30 times as much to solve your problem 30 times faster?
2. Can your code be easily vectorized?
3. Will your program be used enough to warrant hand-optimization?
4. Can you afford to retune your software's floating-point arithmetic?
5. Is your application a little too large for a workstation?

If you answer "no" to the first question, but you think you could obtain a speedup greater than 30 with a vectorizing compiler alone (question two), or with hand-optimization followed by vectorized compilation (question three), then porting your code may well make economic sense. A negative answer to either question four or five, however, is a sure indication of trouble. Simplifying, we thus assert that if you answer any two of our questions "no," your application does not belong on a supercomputer.

All five questions are answered affirmatively for Professor Bixby's research into large-scale linear and integer programs, indicating that his work does indeed belong on a Cray. Our application, however, scored four "no" answers: in hindsight, it is strange that we ever attempted to port our wire-routing application to a supercomputer!

Acknowledgements

Most of the experiments described in this article were performed by Jim Fenno and Stephen Thomas of UMD. The wire-routing application was developed by Antony P-C Ng and Prabhakar Raghavan, while they were members of Thomborson's research group at UC Berkeley.

We would also like to thank Robert Bixby, Roy Marsten, John Gregory, and the staff of the Minnesota Supercomputer Institute for their assistance in technical questions arising in the course of this research.

Bibliography

[1] G. Amdahl. The validity of the single processor approach to acheiving large scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, 1967.

[2] Robert E. Bixby. private correspondence, September 1990.

[3] CPLEX Optimization, Inc., 7710-T Cherry Park, Suite 124, Houston Texas 77095. *CPLEX Performance Characteristics*, 1989.

[4] Cray Research, Inc. *Cray-2 Computer System Functional Description.* Manual HR-02000-0D, 1989.

[5] John T. Feo. An analysis of the compuational and parallel complexity of the livermore loops. *Parallel Computing*, 7:163–185, 1988.

[6] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1990. ISBN 1-55860-069-8.

[7] R. W. Hockney and C. R. Jesshope. *Parallel Computers 2.* Adam Hilger, 1988. ISBN 0-85274-812-4.

[8] T.C. Hu and M.T. Shing. A decomposition algorithm for circuit routing. *Mathematical Programming Study*, 24:87–103, 1985.

[9] John M. Levesque and Joel W. Williamson. *A Guidebook to* FORTRAN *on Supercomputers.* Academic Press, Inc., 1989. ISBN 0-12-444760-0.

[10] Roy E. Marsten. XMP *Technical Reference Manual.* XMP Software, Inc., July 1987.

[11] John D. McCalpin. Benchmark summary — FP-intensive, 2-d CFD model. *comp.benchmarks*, November 20, 1990.

[12] Frank McMahon. The Livermore fortran kernels: a computer test of numerical performance range. Technical Report UCRL–53745, Lawrence Livermore National Laboratory, 1986.

[13] Motorola, Inc. *MC68881/MC68882 Floating–Point Coprocessor User's Manual*, first edition, 1987.

[14] Antony P-C Ng, Prabhakar Raghavan, and Clark D. Thompson. Experimental results for a linear program global router. *Computers and Artificial Intelligence*, 6(3):229–242, 1987.

[15] Mohan K. Ramamurthy. Floating-point benchmark comparison. *comp.benchmarks*, November 12, 1990.

[16] Texas Instruments, Inc. *TGC100 Series 1μm CMOS Gate Arrays Data Manual SRGS007*, 1989. Manual SRGS007.