Reviews

DSSLIB(tm)—A Library of Parallelized and Optimized Linear Algebra Subroutines for SPARC Computers. Available from Dakota Scientific Software, Inc., 2241 Cedar Drive, Rapid City, SD 57702-3245. e-mail: sales@scisoft.com.

DETAILED SUMMARY

Cost: $995 for a single CPU or $4,995 one-time charge for a 10-CPU license; includes 1 year of support and 1 year of software upgrades; license can be upgraded for $1,000 per additional 10 CPUs. Optional support beyond the first year is available for 15% of the cost of the license.

Hardware and software requirements: SPARC and SPARC-compatible computers; SunOS 4.1, Solaris 1.0, or Solaris 2.x operating system; works with FORTRAN 1.x, 2.x, or 3.x; iMPact optional.

Licensing: Single-CPU licenses are node locked; 10-CPU licenses or larger are floating licenses.

Capabilities:

1. Automatically parallelizes large computations over available CPUs to significantly increase speed.
2. 100% compatible with LAPACK, LINPACK, FFTPACK, VFFTPACK, and Basic Linear Algebra Subprograms (BLAS) levels 1, 2, and 3 so that many programs can parallelize with no source code change or recompilation.
3. Two modes of parallelization for optimal parallel performance on either dedicated or shared machines.
4. Optimized for SPARC-compatible CPUs to speed up computations that do not parallelize.
5. 64-bit compatibility package simplifies the process of using SPARCs as the development platform for applications intended for a supercomputer.

Environmental considerations: Requires approximately 9 megabytes of disk space.

Performance: Runs LAPACK, BLAS and LINPACK up to four times faster than the netlib version when running on a single-CPU workstation. Performance is considerably higher than when using the automatic parallelism to run a computation on more than 1 CPU.

REVIEW TEXT

DSSLIB is a library of parallelized and optimized linear algebra subroutines based on LAPACK 2.0, LINPACK, FFTPACK, VFFTPACK, and BLAS levels 1, 2, and 3. The significant benefits of DSSLIB are its high speed and its ease of use. The major drawback to DSSLIB is that it is only helpful on applications that are floating-point intensive and it will not improve other types of applications.

LAPACK, LINPACK, FFTPACK, VFFTPACK, and the Basic Linear Algebra Subprograms (BLAS) are public domain linear algebra libraries used in thousands of scientific software packages, both public domain and proprietary. Popular software that can be accelerated with DSSLIB include IMSL/Math and IMSL/Stat from Visual Numerics, NAG from Numerical Analysis Group, and IDL from Research Systems. The company is developing interfaces to some third-party Fortran 90 compilers to allow Fortran 90 vector and matrix operations use the fast subroutines in DSSLIB. Information about using the public domain versions of these libraries can be retrieved by sending the following mail message to netlib@ornl.gov: *send index*.

Reviewed September 1994

Of course, the primary consideration in choosing parallel software is speed. DSSLIB delivers speed in two ways: optimization and parallelization. The result is good performance even on problems that are too small to effectively parallelize. For example, on a SPARCstation 10 the LINPACK 1000 × 1000 benchmark runs 2.5 times faster on a single CPU and well over 6 times faster on 3 CPUs. On a SPARCstation 5, DSSLIB runs the LINPACK benchmark 60% faster than libSci from CraySoft. Although DSSLIB contains significant improvements in most subroutines, little or no improvement is apparent in subroutines dealing with symmetric, Hermitian, or triangular matrices stored in packed-storage mode.

The subroutines in DSSLIB are optimized for the SuperSPARC, microSPARC, hyperSPARC, and SPARC CPUs. According to the company, DSSLIB aggressively uses characteristics of the hardware architecture that allow multiple instructions to proceed simultaneously. It is obvious that writing code so that it can do multiple concurrent operations will greatly improve speed. It is even more obvious that most scientists do not want to think about low-level hardware and compiler details while writing their code. We have found that the level of optimization in DSSLIB allows us to get very good performance without concern for low-level hardware detail.

In addition to optimization, the DSSLIB subroutines are parallelized. When a program calls one of the parallel subroutines then DSSLIB determines how many CPUs to use, how to partition the data, and divides the work among the available CPUs. This process of parallelizing a computation is automatic and no change is required either in the code or in the way that a program is run. DSSLIB has two modes of parallelism, one that gives peak performance on a dedicated machine and the other that is best when there are multiple jobs running concurrently. DSSLIB, Sun's iMPact, and CraySoft's libSci all show excellent performance on dedicated parallel machines. However the Sun and CraySoft parallelism is very resource-intensive and they both suffer great performance degradation in shared environments. We have found that DSSLIB maintains high performance in both environments.

Just as one expects to get high-speed from parallel software, one also expects that using parallel software is difficult and error prone. With DSSLIB, we found that everything from the installation to actual use in our production environment is reasonably easy. The ease of use comes from three factors: compatibility with netlib, documentation, and a 64-bit development option. Each of these factors is briefly described below.

DSSLIB interfaces are 100% compatible with the standard netlib interfaces. (The standard LAPACK 2.0 has different workspace requirements than LAPACK 1.1 but DSSLIB has made algorithmic adjustments to be compatible with programs that use either LAPACK 1.1 or LAPACK 2.0.) This compatibility allowed us to parallelize our programs by relinking with DSSLIB. No source code changes were required, nor did we do anything differently in running our programs. In addition to our programs based on the standard libraries, we also parallelized an image processing program written in a proprietary interpreted language called IDL. The original IDL program did not use BLAS, so we did make source code changes. Most of the processing was in a 2-D discrete cosine transform subroutine, and we were able to parallelize that subroutine in under 3 hours.

The manual that comes with DSSLIB is very good. It is clear, complete, and well written. The documentation of each subroutine includes an example program with sample input and output. Many of the subroutines also come with references to related subroutines, tips on using the subroutines more effectively, and warnings about common programming errors. The on-line documentation consists of man pages for the subroutines. These are good, but not of the same quality as the written manual. There is no interactive capability similar to the Interactive Documentation Facility in Visual Numerics' IMSL products, so users will need the printed documentation for help on unfamiliar topics.

The 64-bit development option is for people who use workstations as a development platform for 64-bit supercomputers. The naming convention used by the standard libraries is that subroutines whose names begin with S or C process single precision real numbers and subroutines whose names begin with D or Z process double precision numbers. Single precision on a SPARC or most UNIX workstations is 32 bits, but single precision on a mainframe or supercomputer is 64 bits. This means that moving a 64-bit application from a Sun to a Cray requires the user to change the names of all of the subroutines. For example, the user must change calls from DGEMM (64-bit SPARC matrix multiply) to SGEMM (64-bit Cray matrix multiply). The 64-bit development option is a version of DSSLIB in which the S and C subroutines process 64-bit data. No name changes are required when the user moves to or from a 64-bit computer. This feature is an interesting one, and it is not available on the other libraries that we have, but it is not useful unless you work on mainframes or supercomputers.

The ease of use allows DSSLIB to fulfill its promise of "parallelizing your application within minutes," but this ease of use exacts a performance penalty on some applications. DSSLIB only parallelizes those operations that are performed by one of its predefined subroutines. It is not a general-purpose parallelization system like Express(tm) or Linda(tm). DSSLIB does an excellent job of parallelizing an application dominated by solving linear systems, eigenproblems, solutions to least-squares problems, and other linear algebra operations. DSSLIB will not help an application dominated by I/O, sorting, or non-mathematical computations. DSSLIB is also not helpful on applications dominated by computationally trivial operations, even if there are many of those operations. For example, we have an image processing program that spends most of its time manipulating 3 × 3 matrices of integers and the rest of its time with 4 × 4 matrices of reals. These operations are so cheap that we do them with our own in-line code rather than use DSSLIB.

In summary, we have found that DSSLIB gives us a fast and easy way to parallelize our numerically intensive applications, especially those based on LAPACK, BLAS, LINPACK, FFTPACK, or VFFTPACK. It is not a general-purpose parallel system, and it is useful only for numerically-intensive applications.

Jeremy Week
South Dakota School of Mines and Technology
501 E. St. Joseph Street
Rapid City, SD 57701-3995

e-mail:jcw6998@silver.sdsmt.edu
voice: 1(605)343-6496

A Comparative Study of Parallel Programming Languages: The Salishan Problems, by John Feo, Ed., North-Holland (Elsevier), Amsterdam, *1992, $120.00,* 386 pp.

As parallel computing moves out of research labs and into the supercomputing mainstream, the problem of programming parallel machines is receiving greater attention. The designers and manufacturers of a parallel machine are usually willing to spend hundreds of hours coding for it at the assembly level. Their users, on the other hand, rarely enjoy having to invest an order-of-magnitude more time to get reasonable performance out of their new machine than they would spend programming a conventional workstation or vector supercomputer.

Many benchmarks assess the numerical performance of novel architectures, but no similar tests inform potential users about programmability. Indeed, the very idea of measuring programmability is a suspicious one. There is tremendous variation in users' taste, aptitude, and background. Just as important, no matter how bizarre an architecture or programming system, there exists at least one application for which it is ideally suited. Thus, one finds the advocates of data-parallel languages concentrating on regularly-structured problems which are intrinsically load-balanced, while message-passing's proponents show us task-farm after task-farm, and devotees of the religious sects which have grown up around various functional, dataflow, and logic languages keep pointing out how much simpler their programs appear (to them, at least) than those of their competitors.

Feo's book represents a laudable attempt to establish some kind of baseline to compare the programming language usability on parallel computers. The editor, a member of the Computing Research Group at Lawrence Livermore National Laboratory (LLNL) presents four non-trivial problems. The four problems contain a variety of different types of parallelism, including dynamic task creation, producer/consumer synchronization, and array management (unlike matrix multiplication, numerical quadrature, or the eight-queens problem, which are often used to show language features). These problems are "solved" using eight different programming languages by

Reviewed June 1993

the participants at a 1988 Salishan workshop sponsored by LLNL.

The first problem, known as Hamming's Problem, takes a set of primes {a, b, c, ...}, and a limit N, and to output in increasing order, without duplicates, all integers with exactly those prime factors which are less than N. The Paraffins Problem is similar—given an integer $N > 0$, output the chemical structure of all paraffin molecules which have up to N carbon atoms. (A paraffin molecule contains only single carbon–carbon and carbon–hydrogen bonds, and no loops.) The third problem simulates a doctor's office, where a set of patients become ill at random intervals and queue up to be served by one of several doctors. The final, and only numerical, problem solves a system of linear equations $Ax = b$, where A is a skyline matrix, i.e., a matrix whose nonzero elements are contained with a known envelope. While the matrix can use a conventional solver, the intent is that solutions take advantage of the location of zeros in the matrix.

The languages are divided into categories:

1. imperative languages with data-parallel extensions (C*),
2. imperative languages with control-parallel operations (Ada and Occam), and
3. functional or dataflow languages.

Of the latter, Scheme and Sisal are the most widely used, with Haskell, Id, and Program Composition Notation (PCN) representing more modern or extreme alternatives.

Each chapter overviews a language and its implementation, and then discusses the four problems. Most of the contributors present the entire source code for their solutions, which run from a few 10's of lines to several pages per program. As expected, the imperative programs are usually longer than their higher-level brethren, while most of the functional and dataflow solutions presented are broadly similar to one another. Most of the discussion is clear and concise; more programming language comparisons would have helped, but the material presented is a good overview of what people are doing and thinking in parallel computing.

While this book is generally very good, it does have two shortcomings. The first problem is the lack of discussion about how long it took the contributors to develop their programs. An elegant solution achieved after months of hard thinking is probably a poorer measure of usability than a workable solution produced in a day or a week; some measure of programming effort is required and would serve the same purpose (and have the same pitfalls) as megaFLOPS figures for LINPACK, NAS, and other benchmarks.

The second, and more important, problem with The Salishan Problems is its price. $120 for 386 pages is well out of the reach of graduate students and most lecturers, and indeed of many college libraries. One only hopes that Elsevier will produce a paperback edition in the near future so that this valuable work can become more widely known.

Gregory V. Wilson
Computer Systems Research Institute
University of Toronto
6 King's College Road
Toronto, Ontario
Canada M5S 1A4

email:gvw@cs.toronto.edu
Phone: 1(416)978-1241
Fax:1(416)978-1676

Redundant Disk Arrays: Reliable Parallel Secondary Storage, by Garth A. Gibson, MIT Press, *Cambridge*, 1992, *$35.00*, 250 pp.

The simplest way to sum up this book is to say that anyone who is doing research in computer systems should sit down and read it. Even if I/O systems and ways of modelling reliability are not one's primary interests (and Gibson's writing makes them seem very interesting), this book is a beautiful example of how one ought to conduct and analyze research, and indeed of how to choose important directions for research.

The book's central thesis is by now well known. Just as volume production of microprocessors has

Reviewed June 1993

made them more cost-effective than the multichip or multiboard CPUs which typically inhabit mainframes, so the volume production of small disk systems for the microcomputer and workstation markets has led to them providing more storage per dollar, volume, or watt than their larger counterparts. As a result, a system containing a dozen small disks may provide the capacity of a single large one at a significantly lower cost. Such an array might be expected to have a higher overall failure rate because of its larger number of components, but this can be ameliorated by storing data redundantly, using the same coding techniques used to detect and correct single-word faults in most solid-state memories.

Gibson argues that such redundant arrays of inexpensive disks (RAIDs) will inevitably replace large single-disk systems. He backs up this argument with statistics drawn from the behavior of commercially-available disk systems, with a variety of performance models, and with the experience of the RAID group at UC Berkeley. The book's first two chapters introduce his thesis, and review the current and likely future state of I/O systems. Chapter 3 then presents the RAID concept, while Chapters 4 and 5 characterize disk lifetimes, and use these characterizations to support the reliability models which are crucial to the overall argument. The final chapter summarizes his conclusion that RAID systems could exceed the throughput of conventional disks by factors of 6 to 8, while being more reliable, and no more expensive.

It is easy to see why the ACM chose this book as a Distinguished Dissertation in 1991. A decade from now, the work it presents will probably be seen as having been as influential in the 1990s as the development of RISC technology and multiprocessor architectures were in the 1980s.

Gregory V. Wilson
Computer Systems Research Institute
University of Toronto
6 King's College Road
Toronto, Ontario
Canada M5S 1A4

email:gvw@cs.toronto.edu
Phone: 1(416)978-1241
Fax: 1(416) 978-1676