Applied and
Computational
Mathematics
Division

NISTIR 89-4135

Center for Computing and Applied Mathematics

Supercomputers Need Super Arithmetic

D. W. Lozier and P. R. Turner

October 1989

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology
Gaithersburg, MD 20899

SUPERCOMPUTERS NEED SUPER ARITHMETIC

D.W.Lozier and P.R.Turner

Applied and Computational Mathematics Division
National Institute of Standards and Technology
Gaithersburg
MD 20899

lozier@enh.nist.gov

(301) 975-2706

Mathematics Department
US Naval Academy
Annapolis
MD 21402

pturner@usna.mil

(301) 267-3703

ABSTRACT

The title of the paper will be justified by the consideration of the parallel computation of vector norms and inner products in floating-point and a proposed new form of computer arithmetic, the symmetric level-index system.

KEYWORDS Parallel computation, vector norms, symmetric level-index arithmetic, floating-point arithmetic, error analysis, vectorized algorithms.

1. Introduction.

In this paper we discuss the computation of vector p-norms and scalar products and the implementation of algorithms for them on vector and parallel computers. For the most part, we concentrate on the vectorization of algorithms for these operations. We begin with a brief overview of three approaches to the p-norm calculation including an extended version of Blue's algorithm for the euclidean norm of a vector [2]. These particular vector norms are attracting increased computational interest caused in part by their role in the radial basis function approach to bivariate approximation; see [19] for example. The analysis of finite element techniques also makes use of these norms.

In Section 2, we concentrate on the relative merits of the various approaches for a serial machine. Section 3 extends the discussion to vector and (briefly) to other parallel architectures. We see there that considerations of the ease of vectorization lead to very different conclusions as to which is the algorithm of choice. For other parallel architectures the decision may well be different again.

The choice and performance of the algorithm for the computation of the p-norm is not just architecture-dependent but also depends on the arithmetic which is to be used and on the detailed implementation of that arithmetic. In order to fix a framework for the discussion, we adopt the Brown [3] model of floating-point arithmetic. In the computational examples we use the IEEE single precision standard [IEEE].

The IEEE standard facilitates comparison with the symmetric level-index, or *sli*, system. This arithmetic, which is an extension of the original level-index scheme of Clenshaw and Olver [6], [7], is outlined in Section 4. For greater detail on this arithmetic and its possible implementation see [6], [7], [9] and the introductory survey [8]. As long as it remains the case that sli arithmetic must be implemented in software, it is difficult to implement a genuine simulation of the double-length symmetric level-index scheme. Therefore, we consider the single-length versions of both arithmetics for the purposes of comparison. (One such software implementation for experimentation on PC's is available

[21] while a Fortran implementation with its own precompiler is under development by Lozier.)

In Section 5, we discuss floating-point error analysis and the analysis of extensions of the floating-point system as well as error analysis for sli arithmetic. The focus of the discussion is the computation of extended sums. This operation is central to any of the more complicated operations such as evaluation of scalar products or vector norms.

In Section 6, we consider some of the sli vector algorithms for the tasks of evaluation of the p-norm and scalar products and see that these would be immediately parallelizable as soon as a symmetric level-index arithmetic processor is available. Section 7, on computational experience with the algorithms under discussion, demonstrates the advantages claimed. The findings are summarized in the final section of the paper.

2. Algorithms for Vector Norms

The vector space of complex n-tuples $\mathbf{x} = (x_1, x_2, ..., x_n)$ is a Banach space under the norm

$$\| \mathbf{x} \|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}, \tag{2.1}$$

where $p \geq 1$ is a fixed real number. This norm is called the p-norm. In the special case where $p = 2$, the norm derives from the usual inner product

$$(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{n} x_i \bar{y}_i, \tag{2.2}$$

so that $\| \mathbf{x} \|_2 = (\mathbf{x}, \mathbf{x})^{1/2}$. With this 2-norm, the vector space becomes a Hilbert space. The special importance of Hilbert spaces in applications of mathematics helps account for the existence of robust algorithms for the 2-norm such as Blue's algorithm [2] and for the existence of library software in such collections as NAG, IMSL, SLATEC and many others. Two other special cases - each easier to compute than the 2-norm are

$$\| x \|_1 = \sum_{i=1}^{n} |x_i|, \tag{2.3}$$

and

$$\| x \|_\infty = \lim_{n\to\infty} \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p} = \max_i |x_i|. \tag{2.4}$$

Like the 2-norm, the 1-norm and the ∞-norm are commonly found in software libraries.

Little software is available for the more general p-norms. Yet general Banach spaces are used widely in applied mathematics. For example, finite element analysis is rooted in Banach spaces and not Hilbert spaces and one of the more promising techniques being developed for multivariate approximation using radial basis functions relies specifically on the computation of p-norms [19]. Perhaps the p-norm is available in finite element packages but these are not readily accessible except to finite element practitioners.

We are interested here in algorithms for computing p-norms on serial and vector computers. In each case, we want the algorithm to be matched to the architecture so as to speed the execution. Additionally - and perhaps even more importantly - we seek robust algorithms. Thus they should return either the correct answer or a clear indication of failure. The first, and conceptually simplest, algorithm is based directly on the definition.

ALGORITHM 1S

Set	$s_1 = \| x_1 \|^p$.	
Compute	$s_i = s_{i-1} + \| x_i \|^p$,	$i = 2, 3, ..., n.$
Set	$\| x \|_p = s_n^{1/p}$.	

It is generally accepted that floating-point systems of computer arithmetic are usefully characterized by four parameters. See, for example, [3], [4], [5] and [12]. These parameters are:

β: the radix or base (usually 2 or 16),

t: the number of β-digits in the significand,

e_{min}: the minimum exponent, and

e_{max}: the maximum exponent.

These parameters do not describe an arithmetic processor fully since, for example, the machine (roundoff) unit for arithmetic operations is not known until the rounding algorithm is specified. Brown's contribution [3] was to assign the parameters of a specific arithmetic processor not on the basis of their static number representation, but on the basis of a four-parameter arithmetic model which reflects an acceptable level of performance of the actual arithmetic. Thus any of t, e_{min} and e_{max} may be reduced to compensate for a processor that fails to meet a stipulated set of criteria.

It is not always easy to determine the parameters that best characterize the acceptable level of performance of a given processor. We shall assume, however, that this determination has been made, and that we have β, t, e_{min} and e_{max} available for use. The set of model numbers is the set of all *normalized* floating-point numbers

$$X = \pm 0.d_1 d_2 \ldots d_t \times \beta^e \tag{2.5}$$

with the integer e satisfying $e_{min} \le e \le e_{max}$ and $d_1 \neq 0$. Zero is included also. The model has a maximum, the *overflow threshold*,

$$\Lambda = (1 - \beta^{-t})\, \beta^{e_{max}} \tag{2.6}$$

and the set of positive model numbers has a minimum, the *underflow threshold*,

$$\lambda = \beta^{e_{min}-1}. \tag{2.7}$$

Obviously, Algorithm 1S is susceptible to both underflow and overflow in the calculation of s_i. If overflow occurs, the computation cannot proceed within the arithmetic model. Nevertheless, the final value of $\| \mathbf{x} \|_p$ (if it could be computed) may be well below the overflow threshold and therefore steps should be taken to avoid overflow so that a usable result is obtained. A simple scaling appears to do the trick.

ALGORITHM 2S

Set	$m = \max_i \lvert x_i \rvert, \quad s_1 = \lvert x_1/m \rvert^p$	
Compute	$s_i = s_{i-1} + \lvert x_1/m \rvert^p,$	$i = 2, 3, \ldots, n$
Set	$\lVert \mathbf{x} \rVert_p = m\, s_n^{1/p}.$	

A robust program must do somewhat more. Firstly, it must check to see whether $m = 0$, in which case it must set $\lVert \mathbf{x} \rVert_p = 0$ and exit. Secondly, it must check whether the final multiplication will cause overflow - a perhaps unlikely, but still possible event - in which case it must issue an appropriate error message and quit or, alternatively, link into an error-recovery system designed to allow the programmer an opportunity to circumvent the overflow. Thirdly, the program must take into account the possibility of underflow in the calculation of $\lvert x_i / m \rvert^p$.

We note that Algorithm 2S overcomes the vulnerability to overflow of the simple Algorithm 1S at the expense of a preliminary pass over the data vector to determine the maximal element. The third of our serial algorithms, devised by Blue [2] for the 2-norm case and extended here to the general p-norm, avoids this cost. To simplify the presentation, we denote by ω the largest model number satisfying

$$\omega \leq \min(\Lambda, \lambda^{-1}) \tag{2.8}$$

(cf (2.6) and (2.7)) and let

$$\alpha = \omega^{1/p} \qquad \text{and} \qquad M = \lceil p \rceil \tag{2.9}$$

where $\lceil p \rceil$ is the smallest integer not less than p. Define 2M nonoverlapping intervals by

$$I_m = [\alpha^{m-1}, \alpha^m], \qquad m = -M+1, -M+2, \ldots M \tag{2.10}$$

For each m, α^{-m} is a scale factor that transforms the interval I_m into I_0 by mapping $x \rightarrow \alpha^{-m}x$. By definition, p^{th} powers of numbers in this interval I_0 can be formed in the model without overflow or underflow. The basis of the algorithm is the use of a separate accumulator for each interval I_m.

ALGORITHM 3S

Initialize Set $S_m^{(0)} = 0, \qquad m = -M+1, -M+2, \ldots, M.$

Loop For each $x_i \neq 0$ $(i = 1, 2, \ldots, n)$:

Compute $m_i = \lceil \log_\alpha |x_i| \rceil$

Set $S_{m_i}^{(i)} = S_{m_i}^{(i-1)} + |x_i / \alpha^{m_i}|^p$

and $S_m^{(i)} = S_m^{(i-1)}, \qquad m \neq m_i.$

If $x_i = 0$, set $S_m^{(i)} = S_m^{(i-1)}$ for all m.

Finalize

Set $N = \max \{m : S_m^{(n)} \neq 0\}, \quad P = \min \{m : S_m^{(n)} \neq 0\}$

Set $S = S_N^{(n)} + S_{N-1}^{(n)} / \omega + \ldots + S_P^{(n)} / \omega^{N-P}$

Set $\| x \|_p = \alpha^N S^{1/p}.$

As was the case with Algorithm 2S, a completely robust program must do a little more. Overflow remains possible in the accumulators and the final summation only if $n = O(\Lambda)$, which is highly unlikely unless e_{max} is unusually small. Underflow is highly probable in forming the terms of the final summation. This could result in a genuine loss of precision if underflows are replaced by zero (abrupt underflow) but good protection against such loss would be afforded by use of gradual underflow. To achieve similar protection with abrupt underflow, a more complicated version of the final decision process of Blue's algorithm [2] could be employed. The programming required to render the algorithm robust is nontrivial; this point may be regarded as the central conclusion to be drawn from Blue's paper [2]. That is,

> **even with reasonably good information about the floating-point arithmetic it is still not easy to achieve complete robustness even for such a conceptually simple calculation as a vector norm.**

An advantage of this algorithm over Algorithm 2S is the need to process the data vector once only - although that processing is clearly more expensive. A disadvantage is

the need to obtain the parameters of the floating-point model from the computational environment - a task that is burdensome but necessary when the program is to be portable. The principal additional costs in Algorithm 3S are derived from the need for 2M accumulators and the computation of m_i for each i.

3. Vectorized Algorithms

We present vectorized versions of Algorithms 1S, 2S and 3S. First we describe, as a subsidiary algorithm, the familiar cascading algorithm for vectorized summation to form

$$S = \sum_{i=1}^{n} \xi_i \tag{3.1}$$

where we assume n is given in the form

$$n = 2^k + m \tag{3.2}$$

where k and m are the unique nonnegative integers satisfying (3.2) with $m < 2^k$.

ALGORITHM VS

Initialize

Set $n_0 = 2^k$.

For each i = 1, 2, ..., m set $t_i^{(0)} = \xi_{2i-1} + \xi_{2i}$.

For each i = m+1, m+2, ..., n_0 set $t_i^{(0)} = \xi_{m+i}$.

Loop For each j = 1, 2, ..., k

set $n_j = n_{j-1} / 2$,

set $t_i^{(j)} = t_{2i-1}^{(j-1)} + t_{2i}^{(j-1)}$, $i = 1, 2, ..., n_j$.

Finalize

Set $S = t_1^{(k)}$

If the add-and-store operations inside the loops were executed entirely in parallel, this complete summation would require just k+1 such operations. That is, parallel execution of Algorithm VS requires $O(\log_2 n)$ operations as opposed to the O(n) that are

necessary for serial execution. Because operations on a vector processor are not completely overlapped, Algorithm VS needs O(n) operations on such a machine but with a reduced execution time per operation. The saving becomes more significant as the vector length increases.

Algorithm VS employs 2^k temporary storage locations that are not needed in a serial algorithm. It is characteristic of vector (or parallel) algorithms that reduced execution time is achieved at the expense of additional storage requirements. The initialization phase of this algorithm compresses the original vector into one of length 2^k in a way which minimizes the arithmetic operation count for that phase.

The principles of Algorithm VS extend naturally to other *reduction processes*, a term that is sometimes used in compiler manuals to describe mappings from vector to scalar quantities. For example it extends to the computation of maxima or minima of vectors. The operation count and the temporary storage requirements are unchanged, provided that the *operation* is suitably defined. In the case of finding the maximum of a vector, the *operation* is finding the maximum of two numbers and storing the result.

Algorithm VS vectorizes Algorithm 1S by simply modifying the initialization step to use the components $|x_i|^p$ and the finalization step to compute $(t_1^{(k)})^{1/p}$.

ALGORITHM 1V

Use Algorithm VS modified as above to compute $\|\mathbf{x}\|_p$.

Algorithm 2S can be similarly vectorized.

ALGORITHM 2V

Use Algorithm VS, suitably modified, to compute $m = \max |x_i|$.

Use Algorithm VS, suitably modified, to compute $\|\mathbf{x}\|_p$.

Each of the Algorithms VS, 1V and 2V is easily expressed in standard Fortran and a vectorizing compiler can be expected to vectorize all of the (inner) loops. We turn now to the possible vectorization of Algorithm 3S. The initialization step vectorizes without difficulty. The loop presents two difficulties:

(i) The need to distinguish the cases in which $x_i = 0$, and

(ii) the need to select the appropriate accumulator when $x_i \neq 0$.

In the finalization step, the vectorized determination of N and P is not readily expressed in standard Fortran, and a choice must be made between Algorithm VS and Horner's rule for evaluating the polynomial expression.

Let us consider the loop, in Algorithm 3S, which is where most of the work is done. The difficulties (i) and (ii) arise from the need to maintain an uninterrupted flow of operands to the vector pipeline of the floating-point processor. An uninterrupted flow is possible when the operands are spaced regularly in the computer memory and all operands undergo exactly the same sequence of arithmetic operations. These conditions translate into restrictions on the Fortran statements that are allowed inside a loop in order that the loop be vectorized by a Fortran compiler.

These restrictions are not the same for different Fortran compilers.For example one reference manual [Cray] states that "Loops containing a GO TO, IF, CALL or I/O statement are not vectorizable" while another [Convex] states that "DO loops containing nested IF statements and nonlinear subscripts (subscripts whose values do not form arithmetic progressions) **can** be vectorized." We remark that the code for the loop of Algorithm 3S contains both IF statements and nonlinear subscripts. The difference between the statements made in the two different compiler manuals can be understood in the following way.

First, we remark that it is important to minimize restrictions on inner-loop programming because they exclude important algorithms. In recognition of this fact, every vectorizing compiler provides nonstandard capabilities for doing so. However the price of these nonstandard capabilities is a loss of portability. This places a burden on programmers and, especially, on providers of software libraries.

Secondly, we remark that, conceptually at least, a loop containing logical branches can be separated into two loops, one containing the logic and one containing the

arithmetic. Some compilers, for example [CDC], support the separation of arithmetic and logic in a natural, though nonstandard, way. The programmer can write a loop defining a bit-vector containing one bit for each component of the arithmetic vector operands. The logic loop contains logical and relational operations and the arithmetic loop refers to the previously computed bit-vector. For those operations where the appropriate bit is set, the arithmetic result is stored while for those for which this bit is not set, the arithmetic result is not stored. In this latter case, any floating-point exceptions which may arise such as underflow/overflow or invalid arguments to functions must be suppressed. For this separation to be effective, both the logic loop and the arithmetic loop must be vectorizable.

Finally, we observe that several compilers now perform the separation of logic and arithmetic through analysis of the source code, at least in sufficiently simple cases.

The following is one plausible vectorization of Algorithm 3S. An analog of Algorithm VS is used to perform the summations subject to control by a bit-vector.

ALGORITHM 3V

Initialize

Set $S_m^{(0)} = 0, \qquad m = -M+1, -M+2, ..., M.$

Set $m_i = \begin{cases} \lceil \log_\alpha |x_i| \rceil & \text{if } x_i \neq 0 \\ 0 & \text{if } x_i = 0 \end{cases} \qquad i = 1, 2, ..., n.$

Loop

For each $m = -M+1, -M+2, ..., M$:

Set $b_i^{(m)} = \begin{cases} 0 & \text{if } m \neq m_i \\ 1 & \text{if } m = m_i \end{cases} \qquad i = 1, 2, ..., n.$

Use Algorithm VS (controlled by $b^{(m)}$ and appropriately initialized) to compute $S_m^{(n)}$.

Finalize as in Algorithm 3S.

A general remark is in order on the efficiency of vectorizing inner loops when some of the loop iterations do not store a result. Such iterations take the same time as do iterations for which the result is stored. The separation approach maintains an uninterrupted flow in the vector pipeline but clearly the effective vector rate is degraded. If the proportion of null operations is large enough, a vector loop with its associated overhead can produce an effective rate slower than would be achieved by a scalar algorithm. In Algorithm 3V, the effective vector rate is eroded rapidly as M increases. Therefore this algorithm is not attractive for computing $\|\mathbf{x}\|_p$ for large p.

We conclude this section with the observation that an extremely simple serial algorithm, Algorithm 1S, vectorizes easily. But neither Algorithm 1S nor 1V is at all robust, i.e resistant to overflow and underflow. The greater robustness of Algorithms 2S and 3S is, of course, retained by their vectorization. But it is probably impossible to vectorize Algorithm 3S in a way which is both portable and efficient.

4. The Symmetric Level-Index System

In this section, we review briefly the basic definitions and properties of the symmetric level-index representation and its arithmetic and go on to the summation of a vector as an sli operation.

4.1 Symmetric Level-Index Arithmetic

A positive number X is represented in the level-index, *li*, system by x where

$$X = \phi(x) \tag{4.1}$$

and the *generalized exponential function* ϕ is defined by

$$\phi(x) = \begin{cases} x & 0 \le x \le 1, \\ \exp(\phi(x-1)) & x > 1. \end{cases} \tag{4.2}$$

The representation x is written in the form $l + f$ where the *level* l is a nonnegative integer and the *index* $f \in [0, 1)$. The various quantities are related by the equation

$$X = \exp(\exp(\ldots(\exp f)\ldots)) \tag{4.3}$$

where the exponential function is applied l times.

For the symmetric level-index system, numbers in the interval (0, 1) are represented by the li images of their reciprocals together with an indicator to show that they are in reciprocal form. Thus the sli representation can be described as representing a real number X by $\pm\,\phi(x)^{\pm 1}$. (There are other ways to describe this representation which are sometimes convenient for the analytic development of the theory.) The arithmetic algorithms for the li and sli systems are described in detail in [7] and [9], while possible hardware implementation schemes are discussed in [20] and [18].

A minimal description of the arithmetic algorithms is necessary in order that some of the available simplicity of the vector operations can be clearly demonstrated. Consider the addition (or subtraction) of two numbers $\phi(x)$ and $\phi(y)$, with $x \geq y$, to form their sum (or difference) $\phi(z)$. The problem is to find z, the sli representation of this sum or difference; that is, we seek z such that

$$\phi(z) = \phi(x) \pm \phi(y) \tag{4.4}$$

This is achieved via the computation of the members of three short sequences defined by

$$a_j = 1 / \phi(x-j), \qquad b_j = \phi(y-j) / \phi(x-j), \quad c_j = \phi(z-j) / \phi(x-j). \tag{4.5}$$

These sequences are computed from appropriate starting values by simple recurrence relations involving evaluation of exponential or logarithmic functions for special and restricted ranges of their arguments. Slight variations of these sequences are needed for some of the other arithmetic operations involving quantities in reciprocal form, but the principle is similarly straightforward. The details of the computation are not important to the present discussion. For a detailed description of these algorithms and possible implementations see, for example, [8].

The important point to make about the arithmetic algorithms at this point is that all the internal computation is performed in fixed-point fixed-precision form. It is this

fact which makes considerable economy-of-scale available in the implementation of the extended arithmetic operations to be discussed shortly.

For our present purposes, it is not the details of the algorithm that are essential, but more the properties of the sli system once implemented.

One of the most important properties of the sli system is *closure*. Let A(t,*l*) be the (finite) set of all sli numbers with t-bit indices and levels not greater than *l*. In [15] it is proved that A(t,*l*) is closed under the four basic arithmetic operations, excluding division by zero, if *l* is large enough. For example A(27,7) is closed and requires only 32 bits to represent all of its members (27 for the index, 3 for the level and one each for the sign and the reciprocation indicator). In contrast to the IEEE standard P754 [IEEE], it is not necessary to introduce an artificial infinity arithmetic with non-numerical infinity symbols. The error measure for symmetric level-index arithmetic is different from that for floating-point. The appropriate error measure for the sli system is *generalized precision* which is developed and discussed in [6]; it corresponds to fixed *absolute* precision in the index.

The importance of closure is that it renders the system entirely free from overflow or underflow as a result of arithmetic operations. As we shall also observe later, generalized precision is the appropriate measure for the sort of calculation of current concern. It is precisely the right measure to use when comparing very large or very small quantities which are themselves to be the arguments of a high-order root function so that we can draw appropriate conclusions about the accuracy of the final answer. This particular point is also borne out by the computational experience with the sli system reported in [10] on the root-squaring process for polynomial root-finding and [21] where the p-norm calculation was seen to be highly stable as p increased.

4.2 Summation as an sli operation

In following sections, we shall be describing the sli implementation of algorithms for computing scalar products and p-norms of vectors. It is immediately apparent that both of these operations must rely heavily on the summation of the components of a vector. We conclude this section with a look at this operation which is the subject of the error analysis discussion in the next section. The possibility of the efficient computation of extended sums for the level-index system was first considered in [7], a software implementation is discussed in detail in [21] and the error analysis of extended arithmetic operations in the li and sli systems is discussed in [17].

Firstly, the implementation of the operation even for a serial machine becomes highly efficient once the largest term in the sum, x_{max} say, has been identified. In this case only one sequence $\{a_j\}$ and only one $\{c_j\}$ need be computed. The whole of the extended nature of the operation is accounted for by a redefinition of c_0 which depends on values of b_0 (or its equivalent for quantities in reciprocal form) for the terms to be summed. That is, for the simplest case where all terms are greater than unity, we set

$$c_0 = \Sigma\, b_{0,i} \tag{4.6}$$

where the summation is taken over all terms except for the largest and

$$b_{0,i} = \phi(x_i) / \phi(x_{max}) \tag{4.7}$$

This redefinition of c_0 simply amounts to the summation of these fixed-point, fixed-precision quantities $b_{0,i}$ - an operation which is easily achieved since there are no alignment shifts or normalizations to account for. The savings thus achieved amount to about two-thirds of the work which might otherwise be involved. Details of the algorithmic aspects of this can be found in [21].

On a machine with a sufficient degree of parallelism available in its sli processor, it is clear that all the sequences $\{b_j\}$ could be computed simultaneously and so the complete operation would be slowed in comparison to a single addition only to the extent of identifying x_{max} and then computing the redefined c_0 via a tree of adders. Similarly, for

a software implementation on a vector machine, the computation of the various quantities $b_{0,i}$ is readily vectorizable as in Algorithm VS.

5. Error Analysis

In this section we discuss briefly the error analysis of the IEEE standard floating-point system with reference to the effect of gradual underflow and the consequent loss of a uniform relative error bound for arithmetic operations. We shall also consider how this affects other possible floating-point-like arithmetics which have been proposed. The discussion will center on the summation operation.

We shall also include a short discussion of the error analysis for the sli operation of forming sums either by repeated addition or using the summation algorithm described briefly in the last section. The implementation described in [21] incorporates this operation along with the formation of scalar products and vector norms in its standard library of functions and procedures. The vector sum operation is central to all of these.

5.1 Floating-point

In his lengthy paper [11], Demmel highlights some of the advantages, in terms of writing robust programs, which are derived from the inclusion of gradual underflow in the floating-point arithmetic. Many of his examples are drawn from linear algebra applications including the formation of scalar products. The case Demmel puts for the inclusion of gradual underflow is convincing and justifies the extra complication in the floating-point error analysis which is necessitated by the loss of normalization at the underflow threshold λ given by (2.7). One of his principal arguments in favor of gradual underflow is the consequent relative ease of writing "highly robust, expert codes for problems like polynomial root-finding"; the basic belief being, apparently, that it is preferable to ease the programming task at the expense of slightly more complicated error analyses. This is a view which probably meets with very widespread approval and

acceptance - and certainly that of the present authors.

The source of the additional difficulty in the error analysis is that, while relative error in the floating-point representation is bounded by the machine unit ε for quantities in excess of the underflow threshold λ, for smaller quantities the absolute bound $\lambda\varepsilon$ must be used. For a t-digit base β significand with symmetric rounding

$$\varepsilon = \beta^{1-t} / 2.$$

It is apparent from [11] that the detailed analysis of algorithms becomes more intricate - even though the final error bounds achieved, as is the case for summation, are often no more complex than for fully normalized arithmetic.

Specifically for the sum of N+1 floating-point numbers $X_0, X_1, ..., X_N$ of the same sign which are assumed to be exact, we see from [11] that the final rounding error is bounded as follows:

$$\left| \mathrm{Fl}(\Sigma X_i) - \Sigma X_i \right| \leq N\varepsilon\, \mathrm{Fl}(\Sigma X_i) / (1 - \varepsilon) \qquad (5.1)$$

where $\mathrm{Fl}(\cdot)$ stands for the result of the floating-point operation, so that the relative rounding error is bounded by (approximately) N times the bound for a single addition. In the case of other extended calculations such as scalar products, the error bounds are a more complicated combination of relative and absolute bounds depending on the occurrence of (gradual) underflow. For such calculations with gradual underflow, the attractive feature of a fixed relative error bound for all floating-point arithmetic operations is lost.

Other unnormalized floating-point arithmetics have been proposed for purposes of VLSI architectures and bit-by-bit pipelining of arithmetic. The details of these arithmetic systems are not our present concern, but it is worth saying a little about the error analysis requirements of such systems. The additional complication - in relative error terms - is much greater for these general unnormalized systems than for the special case of gradual underflow. It has been studied in some detail by Barlow [1] who pays particular attention to the error analysis of Gaussian elimination. The essential difference

between his analysis and the analysis of conventional floating-point arithmetic lies in the fact that relative error ceases to be the appropriate measure. It is replaced by "fractional error" which is (essentially) the absolute error in the (unnormalized) mantissa. Like the inclusion of gradual underflow, the use of such unnormalized arithmetic demands the use of other error measures as well as relative error.

Gradual underflow (and any other unnormalized system) has the effect of extending the range of representable numbers close to zero. Such expedients do nothing to alleviate the potential dangers at the other end of the range. Matsui and Iri [16] and Hamada [13] have suggested extensions of the floating-point system to alleviate the overflow problem by allocating variable-sized segments of the computer word to the exponent and mantissa of floating-point representations. The complication of the relative error analysis of such systems is even greater than those mentioned above. Very little of this analysis exists.

5.2 Symmetric level-index

For symmetric level-index arithmetic, too, it is necessary to use a different error measure, *generalized precision* [6]. Generalized precision has some significant advantages compared to relative error, not least of which is that it is a metric so that the symmetry of x approximating $\bar{x}$ and $\bar{x}$ approximating x is a natural aspect of the error analysis. Detailed error analyses of numerical processes will inevitably be different in this system than for any of the floating-point systems but significant progress has already been made in this respect. (See, for example, [6], [8] and [17].)

We turn now to questions of the precision that can be achieved in extended calculations. Olver [17] has demonstrated that, at relatively low cost, it is possible to perform a Wilkinson-style running error analysis. Such a running analysis is particularly well-suited to a parallel environment since it would be performed by simultaneous duplication of the operations for slightly adjusted data. The adjustments are similar to the

use of directed rounding in interval arithmetic and have a similar effect in yielding guaranteed error bounds for the results obtained.

A first-order error analysis for extended sums and products leads to conclusions that are broadly similar to those for the floating-point system in that the generalized precision of the final result is bounded by N times the generalized precision for individual operations. If δ is the generalized precision of the sli representation, so that

$$\delta = \beta^{1-t}/2$$

for symmetric rounding with a t-digit base β index, and taking all the terms in the sum to have the same sign, we find that the final bound for the error δx in the sli representation x of the extended sum

$$\phi(x) = \phi(x_0) + \phi(x_1) + \ldots + \phi(x_N)$$

is given by

$$|\delta x| \leq N\delta. \tag{5.2}$$

Compare (5.1). Similar conclusions apply to extended products.

Examples of such computations are presented in [21]. They demonstrate that, even in cases where underflow or overflow are not significant problems, sli arithmetic provided answers of similar or greater relative precision than their floating-point counterpart.

We turn now to the error analysis of the direct extended sum algorithm described in the previous section. There is not the space to include a fully detailed analysis at this point. These details will be included in a forthcoming paper. We confine ourselves to a brief outline of the analysis in the simplest case and the comment that the other cases can be handled by comparably straightforward extensions of the error analyses in [7] and [9].

The important finding is that the extended summation algorithm can reduce the roundoff error significantly by comparison with (5.2) above. Indeed the error δx above can be bounded by $(N+1)\delta/2$ which is only about half the error committed by repeated addition. Furthermore, it is apparent that by making the appropriate choices of working

precisions an extended summation processor could be designed to reduce the error here to the same magnitude as that for a single operation. The price of this would be that all internal calculation is computed to approximately $\log_2 K$ extra bits of precision where K is the maximum vector-dimension available.

6. Parallelization in an Sli Environment

In Sections 2 and 3, we introduced possible algorithms for the computation of p-norms and studied their relative efficiencies in various computer architectures. In this section we reexamine those algorithms from the point of view of efficiency in parallel architectures **with a built-in sli arithmetic processor**. Of course, no such machine exists at present but one of our conclusions is that it should for the reasons that will become apparent.

Let us first recall the basic definition of the p-norm of a vector **x**, namely

$$\| \mathbf{x} \|_p = \{ |x_1|^p + |x_2|^p + \ldots + |x_n|^p \}^{1/p}. \qquad (6.1)$$

The first observation to make is that because of the freedom from overflow and underflow afforded by the sli system, the algorithm implied by the definition is a perfectly feasible method for the computation and indeed it may appear to be the optimal such algorithm. (We discuss a still more efficient algorithm later.) This would be summarized by the Pascal-like procedure:

```
for i := 1 to n do u_i := |x_i|^p ;
pnorm := Root(SumVector(u),p);
```

The operation within the loop is clearly immediately parallelizable for any parallel architecture and, since the formation of the powers $|x_i|^p$ is a very simple operation for sli arithmetic (as is the taking of the p^{th} root), this overall operation is particularly well-suited to any parallel environment which is equipped with a symmetric level-index arithmetic processor. This is especially true in the case where the sli processor itself has a high degree of parallelism so that the SumVector operation can benefit from all the acceleration described in the previous section.

In the event that such parallelism is not available, then the summation is immediately amenable to pipelining or to some tree-structured addition procedure for a multiprocessor architecture. Of course, for either of these latter operations, similar considerations as for the floating-point system must be made with regard to organization of the data in order to maximize the overall precision of the final result.

At this stage then, it would appear that the sli system is ideally suited to parallel computing environments and that much of the expected slowdown of individual calculations in comparison with floating-point will be handsomely compensated in such situations. However, we have barely scratched the surface of the available savings.

Consider again the situation of a symmetric level-index arithmetic processor with a high-level of on-chip parallelism. In this situation, we find that the operation of computing vector norms could sensibly be incorporated as a single built-in vector function which could be computed in not much more than a single sli operation time.

The algorithm is based on a similar approach to that outlined above for summation; it entails a simple redefinition of c_0, the starting point of the sequence $\{c_j\}$. Just as with summation, a single sequence $\{a_j\}$ suffices for the whole operation, while the several sequences $\{b_j\}$ can be computed simultaneously.

The appropriate value for c_0 is given by

$$c_0 = \{ \Sigma b_{0,i}^p \}^{1/p}; \qquad (6.2)$$

compare (4.6). At first sight this looks almost as complicated as the original operation, but this first impression is misleading. Since $b_{0,i}$ is computed by an evaluation of the exponential function, $b_{0,i}^p$ can be obtained simply by multiplying the argument by p. (Again this is a fixed absolute precision operation.) Similarly, the formation of the p^{th} root in (6.2) is simply incorporated into the computation of

$$c_1 = 1 + a_1 \ln c_0 \qquad (6.3)$$

by the fixed-precision division of the logarithm of $\Sigma b_{0,i}^p$ by p. The only time penalties incurred here relative to a single sli operation are in identifying the largest element of the

vector and in performing the extended fixed-point summation using an efficient tree-structure.

The location of the dominant element - or perhaps a more complex sorting operation - is required of all the algorithms under serious consideration. In the case of the symmetric level-index representation, this operation need be no more complicated than for integer variables since the bit-patterns used for the representation can be organized to preserve the natural integer ordering. This fact is utilized in the implementation described in [21] in which vector norms are computed using the algorithm just outlined - in a serial implementation.

We see here that the symmetric level-index system allows the straightforward definition of the vector norms, Algorithm 1S, to be used for their computation and that this permits immediate vectorization or high-degree parallelization for any supercomputer architecture. In the event that this architecture allows significant parallelism within the sli processor the operation can be made especially efficient.

At this point it is natural to investigate the more general task of computing scalar products of two vectors. (See [11], Section 7 for a simple example which illustrates the difficulty in producing robust floating-point algorithms for this task.) This operation has been one of the main planks on which the interval arithmetic packages such as that described in [14] have been built.

One likely suggestion for the computation of the inner product $(\mathbf{x}, \mathbf{y})$ of the two vectors $\mathbf{x}$ and $\mathbf{y}$ in a floating-point environment would be that each vector should be scaled by its largest element and the final result scaled in a compensatory manner. However, as we see in the next section, this may be totally inappropriate since these elements may themselves contribute to almost insignificant terms of the inner product. Perhaps any scaling should be with respect to the largest term of the inner product itself but this would add a generally unacceptable burden to the whole procedure.

It is perfectly conceivable to achieve some similar effect by considering the scaling as an exponent shift where the largest sum of exponents for the terms $x_i y_i$ would be used. However this requires a considerably more complicated sorting procedure to be adopted and then a decision as to how the overall shift should be divided between the two vectors. The additional start-up time for any vectorized algorithm utilizing any of these approaches would be prohibitive in almost all cases.

The corresponding scalar product algorithm for the symmetric level-index computing environment is described by the following PASCAL-like code

```
for i := 1 to n do w_i := x_i*y_i;
ScalarProd := SumVector(w);
```

that is, by a straightforward application of the definition. As we have already seen, the SumVector operation is well-suited to parallel implementation, and the loop is clearly immediately vectorizable or adaptable to any parallel environment.

7. Computational Examples

Among the algorithms discussed in Section 2, Algorithm 3S is the algorithm of choice for complete robustness in a serial floating-point environment. In a vector processor environmment Algorithm 2V becomes preferable due to its relative ease of vectorization and normally adequate robustness. If a robust arithmetic such as symmetric level-index is used then the simplest algorithm, Algorithm 1S (or 1V), is the obvious choice. As a first example, we compare the results of computing $\|x\|_1$, $\|x\|_{10}$, $\|x\|_{100}$ and $\|x\|_{1000}$ for vectors of length 10, 100 and 1000 using Algorithm 1S in 32-bit sli arithmetic and both Algorithms 2S and 3S in IEEE standard (single precision) arithmetic with abrupt rather than gradual underflows.

The sli computation was performed using the Turbo PASCAL unit, *SLIUNIT*, which was described in [21]. This unit implements all the standard arithmetic operations, elementary functions and the extended operations such as computation of p-norms and

scalar products of present interest. Table 1 presents the sli results and the floating-point results for Algorithm 2S. The error estimates were made by comparison with extended precision floating-point computations.

Table 1

Relative error in $\|\mathbf{x}\|_p$ for vectors $\mathbf{x} = (1, 2, ..., n)$ measured in units of 10^{-8}.
(Upper entries for floating-point, lower entries for sli.)

n \ p	1	10	100	1000
10	0	3	2	0
	0	3	2	0
100	10	2	1	4
	0	6	1	4
1000	6	2	2	2
	12	2	2	2

For n = 10, both the floating-point and symmetric level-index computations returned 55 exactly for $\|\mathbf{x}\|_1$ and 10 exactly for $\|\mathbf{x}\|_{1000}$. Since $\|\mathbf{x}\|_{1000} = 10(1 + O(10^{-20}))$ this accounts for the lack of error in this case. In general, the relative error is quite flat throughout the entire test for both arithmetics.

Although many (abrupt) underflows were reported in the floating-point programs for p = 100 and p = 1000, the error in these results does not reflect any undue loss of precision. We ran the same tests using Algorithm 3S. The errors observed were not much different - in some cases slightly smaller and in others slightly larger. No underflows were reported by Algorithm 3S.

The error analysis of Section 5 applies directly to the cases in the first column of Table 1. Let us examine the case n = 1000, p = 1. Using (5.1) with N = 999 and $\varepsilon = 2^{-24}$, we find that the relative error should be bounded by

$$N\varepsilon/(1-\varepsilon) \approx 6\times 10^{-5}.$$

The floating-point error shown in Table 1 is 1000 times smaller, even though scalings in the algorithm are not accounted for in the bound (5.1). Similarly, using the bound (5.2) - which is the appropriate one for the algorithm used here - with N = 999 and $\delta = 2^{-27}$, the generalized precision of $\|\mathbf{x}\|_1$ should be bounded by

$$N\,\delta \approx 7.4 \times 10^{-6},$$

which corresponds to a relative error of approximately 2.5×10^{-4}. The relative error of the sli result shown is about 2000 times smaller.

As a second example, we consider the computation of a scalar product. Specifically, consider the vectors **u**, **v** whose components are given by:

$u_i = u_{i-1}^2$ $\quad (i = 1,2,...,6)$; u_0 given and

$v_0 = -u_6$, $v_1 = -u_5$, $v_5 = u_1$, $v_6 = u_0$, $v_i = u_i$, otherwise.

The scalar product of these two vectors was computed both in sli arithmetic and using floating-point with scaling by the largest element as described in the previous section. (This is also the scaling used for Algorithm 2S.) For the first and simplest case with $u_0 = 1$, both systems, of course, produced the correct answer. However on setting $u_0 = 2$, so that $u_i = 2\text{^}(2^i)$, the floating-point computation yielded $\mathbf{u}^T\mathbf{v} = 0$ because the scaling by the largest elements in the arrays (which in this case are both 2^{64}) results in all the middle terms in the scalar product underflowing to zero, while the terms u_0v_0 and u_1v_1 are exactly cancelled by the last two terms.

The sli result here is 4.29497 E+09. The exact scalar product is 4 295 033 088 so that the relative error in the final result is less than 1.5×10^{-5}.

8. Conclusions

The principal conclusions to be drawn from this work are as follows.

1. The simplest algorithms for computing p-norms or scalar products of vectors are readily vectorizable but not robust.
2. Completely robust accurate algorithms for the p-norm are, at best, very difficult to vectorize efficiently.
3. Algorithm 2V represents a normally acceptable compromise between the goals of robustness and vectorization.
4. Much of the difficulty in obtaining parallel algorithms that are completely robust in floating-point arithmetic for these vector operations is caused by the need to safeguard against overflow and (potentially harmful) underflow and to preserve accuracy in the computed result.
5. The symmetric level-index arithmetic system alleviates all of these difficulties because, within that system, the simplest algorithms are robust, portable, accurate and immediately parallelizable owing to the provision of a (completely) robust arithmetic.

The computation of p-norms and scalar products are merely illustrations of a more general pattern in scientific computing applications where, often, considerations of robustness and the desire for parallelization of floating-point algorithms conflict. For all of these reasons we draw the final conclusion

6. The more "super" the computer, the greater the need for "super" arithmetic such as the symmetric level-index system.

REFERENCES

[1] J.L.Barlow, *Error analysis in unnormalized floating point arithmetic*, Report CS-88-10, April 1988, Dept Computer Science, Pennsylvania State University.

[2] J.L.Blue, *A portable Fortran program to find the euclidean norm of a vector*, ACM Trans Math Software 4 (1978) 15-23.

[3] W.S.Brown, *A realistic model of floating-point computation*, Mathematical Software III (J.R.Rice, Ed.) Academic Press, New York, 1977, pp 343-360.

[4] W.S.Brown, *A simple but realistic model of floating-point computation*, ACM Trans Math Software 7 (1981) 445-480.

[5] W.S.Brown and S.I.Feldman, *Environment parameters and basic functions for floating-point computation*, ACM Trans Math Software, 6 (1980) 510-523.

[6] C.W.Clenshaw and F.W.J.Olver, *Beyond floating point*, J. ACM 31 (1984) 319-328.

[7] C.W.Clenshaw and F.W.J.Olver, *Level-index arithmetic operations*, SIAM J Num Anal 24 (1987) 470-485.

[8] C.W.Clenshaw, F.W.J.Olver and P.R.Turner, *Level-index arithmetic: An introductory survey*, Proc. Numerical Analysis Summer School, Lancaster, 1987, Springer Verlag (1989) to appear.

[9] C.W.Clenshaw and P.R.Turner, *The symmetric level-index system*, IMA J Num Anal 8 (1988) 517-526.

[10] C.W.Clenshaw and P.R.Turner, *Root-squaring using level-index arithmetic*, to appear.

[11] J.Demmel, *Underflow and the reliability of numerical software*, SIAM J Sci Stat Comp 5 (1984) 887-919.

[12] P.A.Fox, A.D.Hall and N.L.Schryer, *The PORT mathematical subroutine library*, ACM Trans Math Software 4 (1978) 104-126.

[13] H.Hamada *URR: Universal representation of real numbers,* New Generation Computing, 1 (1983) 205-209.

[14] U.W.Kulisch and W.L.Miranker, *The arithmetic of the digital computer: A new approach*, SIAM Review 28 (1986) 1-40.

[15] D.W.Lozier and F.W.J.Olver, *Closure and precision in level-index arithmetic*, Manuscript.

[16] S.Matsui and M.Iri *An overflow/underflow - free floating-point representation of numbers,* J. Information Proc. 4 (1981) 123-133

[17] F.W.J.Olver, *Rounding errors in algebraic processes - in level-index arithmetic*, Proc. Reliable Numerical Computation (M.G.Cox and S.Hammarling, eds.)

Oxford, 1989, to appear.

[18] F.W.J.Olver and P.R.Turner, *Implementation of level-index arithmetic using partial table look-up*, Proc. ARITH8, (M.J.Irwin and R.Stefanelli, Eds.) IEEE Computer Society, Washington, DC, 1987, 144-147.

[19] M.J.D.Powell, *Radial basis functions for multivariable interpolation: A review*, Algorithms for Approximation 143 - 167 (M.G.Cox and J.C.Mason, eds.) Oxford, 1987.

[20] P.R.Turner, *Towards a fast implementation of level-index arithmetic*, Bull IMA 22 (1986) 188-191.

[21] P.R.Turner, *A software implementation of sli arithmetic,* pp. 18-24, Proc. ARITH9, (Ercegovac and Swartzlander, Eds) IEEE Computer Society, Washington DC, September 1989.

[CDC] Fortran 200 Version 1 Reference Manual, 60480200, Rev. 5, Oct 23, 1987, Control Data Corporation, Sunnyvale CA.

[Convex] Convex Fortran Language Reference Manual, 720-000050-202, 6th edition, Rev 1, May 1988, Convex Computer Corporation, Richardson, TX.

[Cray] Fortran (CFT) Reference Manual, SR-0009, August 1981, Cray Research Inc., Mendota Heights, MN

[IEEE] IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985, IEEE Inc., New York, NY

U.S. DEPT. OF COMM. BIBLIOGRAPHIC DATA SHEET (See instructions)	1. PUBLICATION OR REPORT NO. NISTIR 89-4135	2. Performing Organ. Report No.	3. Publication Date OCTOBER 1989

4. TITLE AND SUBTITLE

Supercomputers Need Super Arithmetic

5. AUTHOR(S)

D. W. Lozier and P. R. Turner

6. PERFORMING ORGANIZATION (If joint or other than NBS, see instructions)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
GAITHERSBURG, MD 20899

7. Contract/Grant No.

N/A

8. Type of Report & Period Covered

Final

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (Street, City, State, ZIP)

NIST
Department of Commerce
Gaithersburg, MD 20899

Mathematics Department
U. S. Naval Academy
Annapolis, MD 21402

10. SUPPLEMENTARY NOTES

☐ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT (A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)

The title of the paper is justified by the consideration of the parallel computation of vector norms and inner products in floating-point and a proposed new form of computer arithmetic, the symmetric level-index system.

12. KEY WORDS (Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)

Error analysis; floating-point; parallel computation; symmetric level-index; vector norms; vectorized algorithms.

13. AVAILABILITY

☒ Unlimited

☐ For Official Distribution. Do Not Release to NTIS

☐ Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.

☒ Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES

30

15. Price

A03