BASIC LINEAR ALGEBRA OPERATIONS IN SLI ARITHMETIC

MICHAEL A. ANUTA, DANIEL W. LOZIER,
NICOLAS SCHABANEL, AND PETER R. TURNER

ABSTRACT. Symmetric level-index arithmetic was introduced to overcome recognized limitations of floating-point systems, most notably overflow and underflow. The original recursive algorithms for arithmetic operations could be parallelized to some extent, particularly when applied to extended sums or products, and a SIMD software implementation of some of these algorithms is described. The main purpose of this paper is to present parallel SLI algorithms for arithmetic and basic linear algebra operations.

1. INTRODUCTION

This paper reports on a continuing project to develop, implement and apply parallel algorithms for SLI (symmetric level-index) arithmetic. The algorithms are being developed with a view toward a possible future implementation in hardware but at this stage they are being coded for a particular SIMD (single instruction, multiple data) computer system, a DEC MasPar MP-1[1]. The algorithms cover individual arithmetic operations and extensions to the BLAs (basic linear algebra operations) such as the product of a scalar times a vector, the scalar product of two vectors, and the 'saxpy' operation [7] consisting of a scalar times a vector plus another vector. All these operations, especially the BLAs, can benefit from the parallel execution of appropriate algorithms.

A parallel implementation of individual SLI arithmetic operations for the MP-1 was developed in 1995 by Schabanel under the direction of Turner [11]. This is part of an envisioned 'computer arithmetic laboratory' which will use the MP-1 to implement different kinds of computer arithmetic and compare them on representative numerical problems; see Anuta, Lozier and Turner [1].

This paper has been submitted to Workshop 10, Computer Arithmetic, Euro-Par'96, August 27–29, 1996, Lyon, France.

Cray Research Inc., Suite 600, 4041 Powder Mill Road, Calverton, MD 20705, e-mail: mike.anuta@cray.com, voice: 1-301-595-2692, fax: 1-301-595-2647.

Applied and Computational Mathematics Division, National Institute of Standards and Technology, Gaithersburg, MD 20899, e-mail: dlozier@nist.gov, voice: 1-301-975-2706, fax: 1-301-990-4127.

École Normale Supérieure de Lyon, Lyon, France, e-mail: Nicolas.Schabanel@ens.ens-lyon.fr.

Mathematics Department, United States Naval Academy, Annapolis, MD 21402, e-mail: prt@sma.usna.navy.mil, voice: 1-410-293-6732, fax: 1-410-293-4883.

[1] This computer system is identified in this paper to foster understanding. Such identification does not imply recommendation or endorsement by any of the institutions represented by the authors, nor does it imply that the MP-1 is necessarily the best available for the purpose.

The MP-1 is located at the U. S. Naval Academy. Its processor array has a 4-bit architecture with 4096 processors connected in a 64×64 square array. The communication network consists of a 4-bit bus connecting the rows and columns of the array with toroidal wraparound to provide edge as well as interior elements with exactly four nearest neighbors. Each processor executes instructions on 4-bit operands using its own registers and memory. In typical SIMD fashion, the processors are separated at any instant into an *active set* and an inactive set, and an instruction stream issued from a special control processor called the ACU (array control unit) is executed simultaneously on all processors in the active set.

The MP-1 is programmed using a version of ANSI C called MPL[2]. MPL programs are compiled on a DEC (Digital Equipment Corporation) workstation that is connected directly to the ACU and processor array. Arithmetic on integer operands consisting of 8, 16, 32 or 64 bits, or FLP (floating-point) operands consisting of 32 or 64 bits, is built up in each active processor using 4-bit operations according to instructions issued by the ACU. The MPL data types corresponding to these operand types are `char`, `short`, `long`, `long long`, `float` and `double`. Since one MPL operation or 4096 take the same time, the power of MP-1 computing depends on algorithms with high data parallelism. This is typical of SIMD computing in general.

The paper is organized as follows. After a review of the SLI representation in Section 2, we review the original arithmetic algorithms and our MasPar implementation in Section 3. This implementation simulates individual SLI operations with a modest degree of parallelism using the MPL data type `long long`. In Section 4 we present modified algorithms that are more parallelizable, particularly in a SIMD architecture. The original and modified algorithms can all be executed in finite-precision fixed-point arithmetic. However, it should be noted here that error analyses of the original algorithms in earlier papers provide an accurate determination of the number of guard bits needed to support a specified precision, at least for individual operations. The number of guard bits is modest. Corresponding error analyses of the modified algorithms will be the subject of future work. In the final section of this paper the BLAs needed for Gauss elimination will be developed.

2. The SLI Computer Arithmetic System

The LI (level-index) representation of real numbers, its application to computer arithmetic, and recursive algorithms for arithmetic operations, were introduced in 1984 and 1987 by Clenshaw and Olver [3, 4]. The *symmetric* form of the LI system was developed in 1988 by Clenshaw and Turner [6]. See also the survey [5].

Define the infinite sequence

$$E_0 = 0, E_{\ell+1} = e^{E_\ell}, \ell = 0, 1, 2, \ldots. \tag{1}$$

If X is an arbitrary real number, define

$$\text{level}(X) = \ell \qquad \text{if } E_\ell \le |X| < E_{\ell+1}, \ell = 0, 1, 2, \ldots. \tag{2}$$

If $\ell = \text{level}(X)$, define

$$\text{index}(X) = \ln^{(\ell)} |X| \tag{3}$$

[2]High Performance Fortran is present also but only MPL is of interest in this paper.

where

$$\ln^{(0)}|X| = |X|, \qquad \ln^{(\ell+1)}|X| = \ln\ln^{(\ell)}|X|, \ell = 0, 1, 2, \ldots. \tag{4}$$

The *generalized logarithm*, defined as

$$\psi(X) = \text{level}(X) + \text{index}(X), \tag{5}$$

maps $[0, \infty)$ onto itself monotonically and so it is invertible on this interval. The inverse, a *generalized exponential function*, is defined recursively by

$$\phi(x) = \begin{cases} x & \text{if } 0 \le x < 1, \\ e^{\phi(x-1)} & \text{if } x > 1; \end{cases} \tag{6}$$

compare (1). Properties of generalized exponential and logarithmic functions are given in [2]. For example, $\phi, \psi \in C^1(0, \infty)$.

For computer arithmetic with w bits per word, LI representations are stored in two fields: (sign bit, generalized logarithm). That is, for an arbitrary nonzero X,

$$X = s_X \phi(x), \qquad x = \psi(X), \tag{7}$$

and x is stored in ordinary $(w-1)$-bit fixed-point format. The sequence (1) grows very quickly. The first few terms are

$$E_0 = 0, E_1 = 1, E_2 = e \approx 2.72, E_3 = e^e \approx 15.2, \tag{8}$$

$$E_4 = e^{e^e} \approx 3,810,000 \approx 10^{6.58} \approx 2^{21.9}, \tag{9}$$

$$E_5 \approx 10^{1,660,000} \approx 2^{5,500,000} \approx 2^{2^{22.4}}, \tag{10}$$

$$E_6 \approx 10^{10^{1,660,000}} \approx 2^{2^{5,500,000}}. \tag{11}$$

As a consequence of this growth, Lozier and Olver proved [8] that the *finite* set of LI representations with level not more than 6, and w (the word length) not more than 5.5 million or so, is *closed* under individual arithmetic operations (excluding, of course, division by zero). Therefore, taking 3 bits in the integer and $w-4$ bits in the fractional part of x always suffices in practice.

The w-bit LI representation has, in effect, an 'accumulation point' at infinity, but not at zero. The w-bit SLI representation

$$X = s_X \phi(x)^{r_X}, \qquad x = \psi(\max(X, X^{-1})) = \psi(X^{r_X}), \tag{12}$$

where $X \neq 0$ and x is stored in $(w-2)$-bit fixed-point format, allows for an 'accumulation point' at both infinity and zero. As a consequence of the closure property cited above, *the SLI system is free of both overflow and underflow.*

Figures 1 and 2, taken from [10], compare SLI and typical FLP representations for $w = 32, 64, 128$; see also [8]. The FLP overflow limits for these word lengths are $10^{38}, 10^{308}$ and 10^{4932}, approximately, assuming significand lengths of 23, 52 and 112 bits. The SLI index lengths are 27, 59 and 123 bits. The vertical axis in each figure measures 'significance'

$$-\log_{10} \frac{X^+ - X}{X}$$

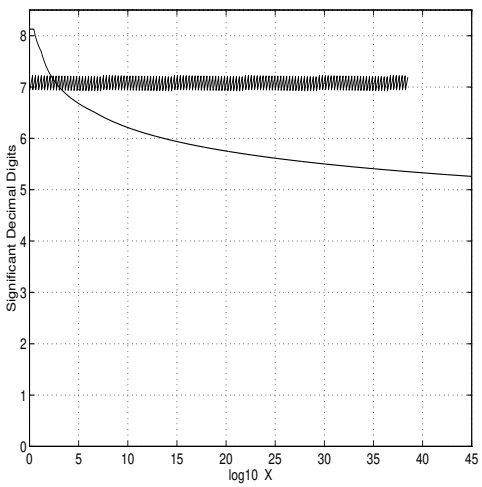

FIGURE 1. SLI vs. FLP for $w = 32$ bits.

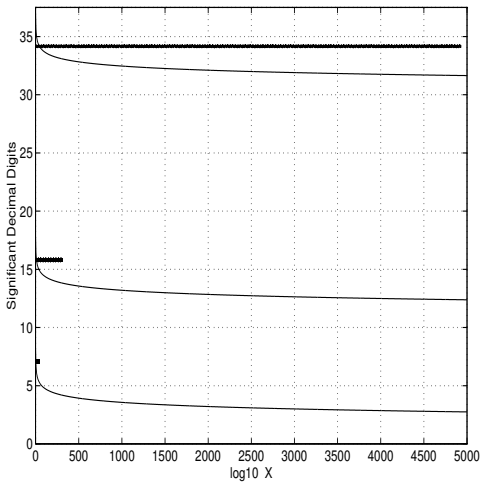

FIGURE 2. SLI vs. FLP for $w = 32, 64, 128$ bits.

where X^+ is the successor of X in a particular computer arithmetic. The horizontal axis measures $\log_{10} X$. As expected, the FLP significance is constant 'on average' but exhibits an oscillation due to binary normalization. The SLI significance exhibits no such oscillation. The SLI significance is better at first but it decreases slowly and crosses over the FLP significance. The FLP systems fail beyond their overflow limits while the SLI systems retain useful significance far beyond the limits of the graphs. The initial superiority of the SLI significance is due to the index having a greater number of bits than the FLP significand for a given word length.

The LI and SLI arithmetic algorithms apply not only to individual operations but, through appropriate adaptations, to expressions of the form

$$X_1 \pm X_2 \pm \cdots \pm X_N, \qquad X_1^{\pm 1} X_2^{\pm 1} \ldots X_N^{\pm 1}.$$

These adaptations reduce the number of recursive steps in comparison to building up the expressions by individual arithmetic operations. Further, the computations can be arranged to take advantage of parallel computing facilities since the algorithms generate N sequences which are completely independent of one another.

The interest in SLI arithmetic stems from its potential for simplifying computer programming. Because of its ability to represent extremely large numbers and their reciprocals in a small number of bits, the vexing overflow and underflow problems of FLP systems are avoided completely. Software engineering experience shows that defensive coding artifices which have been developed to guard against overflow and underflow, such as the ones described in [9], add significantly to the cost of creating and maintaining robust software.

3. A MasPar Implementation of SLI Arithmetic

In this section we review the original recursive algorithms given in [6]. Then we describe our recent implementation on the DEC MasPar MP-1.

3.1. The Original SLI Algorithms. The problem is to solve the equation

$$Z = s_Z \phi(z)^{r_Z} = s_X \phi(x)^{r_X} \circ s_Y \phi(y)^{r_Y} = X \circ Y \tag{13}$$

for s_Z, r_Z, z given $s_X, s_Y, r_X, r_Y, x, y, \circ$. For simplicity of presentation we restrict[3]

$$s_Z = s_X = s_Y = 1, \qquad X \geq Y > 0, \qquad \circ \in \{+, -, \cdot, /\}. \tag{14}$$

We confine our attention here to the additive operations, for which there are three cases:

$$\text{large} \qquad \phi(z)^{r_Z} = \phi(x) \pm \phi(y), \tag{15}$$

$$\text{mixed} \qquad \phi(z)^{r_Z} = \phi(x) \pm \phi(y)^{-1}, \tag{16}$$

$$\text{small} \qquad \phi(z)^{r_Z} = \phi(x)^{-1} \pm \phi(y)^{-1}. \tag{17}$$

In all cases $r_Z = r_X$ *except possibly in large subtraction, mixed subtraction, or small addition.* These exceptional cases have been called *flipover cases.*

Let $\ell_X = \text{level}(X), f_X = \text{index}(X)$ and similarly for ℓ_Y, ℓ_Z, f_Y, f_Z. The algorithm generates the a- and c-sequences

$$a_j = 1/\phi(x - j), \qquad c_j = \phi(z - j)/\phi(x - j), \tag{18}$$

the appropriate form of the b-sequence

$$b_j = \begin{cases} \phi(y - j)/\phi(x - j) & \text{(large)}, \\ 1/\phi(y - j) & \text{(mixed)}, \\ \phi(x - j)/\phi(y - j) & \text{(small)}, \end{cases} \tag{19}$$

and in some situations the h-sequence

$$h_j = \phi(z - j). \tag{20}$$

The a- and b-sequences are generated by

$$a_{\ell_X - 1} = e^{-f_X}, \qquad a_{j-1} = e^{-1/a_j} \qquad (j = \ell_X - 1, \ldots, 1) \tag{21}$$

and

$$\text{large} \quad b_{\ell_Y - 1} = a_{\ell_Y - 1} e^{f_Y}, \qquad b_{j-1} = e^{(b_j - 1)/a_j} \qquad (j = \ell_Y - 1, \ldots, 1), \tag{22}$$

$$\text{mixed} \quad b_{\ell_Y - 1} = e^{-f_Y}, \qquad b_{j-1} = e^{-1/b_j} \qquad (j = \ell_Y - 1, \ldots, 1), \tag{23}$$

$$\text{small} \quad b_{\ell_X - 1} = e^{-\phi(y - \ell_X)} e^{f_X}, b_{j-1} = e^{(b_j - 1)/a_j b_j} \qquad (j = \ell_X - 1, \ldots, 1). \tag{24}$$

Then the starting value for the c-sequence is

$$c = \begin{cases} 1 \pm b_0 & \text{(large)}, \\ 1 \pm a_0 b_0 & \text{(mixed)}, \\ 1 \pm b_0 & \text{(small)}. \end{cases} \tag{25}$$

[3] All these restrictions are removable by basic properties of the SLI representation without any need for additional recursive algorithms.

In the small case, c is the reciprocal of c_0 as defined in (18). Observe that, to a considerable extent, the a- and b-sequences are independent, and that a recurrence like (21) can be used to compute $\exp(-\phi(y-\ell_X))$ in (24).

With the observation that

$$\phi(z)^{r_X} = c\phi(x)^{r_X} = c/a_0^{r_X} \tag{26}$$

we see that flipover occurs when $c < a_0$ in the large and mixed cases, and when $a_0 c \geq 1$ in the small case. Then in the small case, $z = 1 + \ln a_0 c$ and the algorithm is complete. In the other two cases we generate the h-sequence from

$$h_1 = -\ln c/a_0, \qquad h_j = \ln h_{j-1} \qquad (j = 2, 3, \ldots) \tag{27}$$

until $h_j \in [0, 1)$, then we set $z = j + h_j$ and the algorithm is complete.

Now suppose flipover does not occur. If $\ell_X = 1$ then we generate the h-sequence and z as above, but with the starting value $h_1 = f_X + r_X \ln c$. If $\ell_X > 1$ we generate the c-sequence from

$$c_1 = 1 + r_X a_1 \ln c, \qquad c_{j+1} = 1 + a_{j+1} \ln c_j \qquad (j = 1, 2, \ldots) \tag{28}$$

until either

(1) $c_j < a_j$ and $j \leq \ell_X - 1$, which implies $z = j + c_j/a_j$ and the algorithm is complete, or
(2) $j = \ell_X - 1$ and $c_j \geq a_j$, which requires the generation of the h-sequence and z as above, but with the starting value $h_{\ell_X} = f_X + \ln c_{\ell_X - 1}$.

A linearized error analysis in [6] considers the 'working precisions' needed to limit the rounding errors in the algorithms presented above to the size of the inherent errors; see also [4]. The analysis shows, for a word length $w = 32$ bits, it suffices to compute and store all sequences to 6 bits before and 36 bits after the binary point.

3.2. A MasPar Implementation. An early decision to be made in designing an arithmetic unit, whether in software or hardware, is how the exp and ln functions should be computed. The utility of the CORDIC (coordinate rotation digital computer) approach has been proven in handheld calculators [12]. Therefore this technique was chosen and programmed using the 64-bit integer data type `long long` even though use of 64-bit FLP library routines present in the MP-1 would have saved some effort. It will be seen that CORDIC algorithms make effective use of SIMD parallel architecture.

CORDIC algorithms generate a sequence of real triples (x_j, y_j, z_j) from a linear first-order recurrence relation that is defined in terms of a prespecified real sequence ϵ_j. As one component converges to zero, the other components converge to specific function values as determined by the sequence ϵ_j and initial conditions (x_0, y_0, z_0).

Our implementation provides the functions $\lambda \exp(\pm t), \exp((t-1)/\lambda)$ and $\lambda \ln t$ for special and restricted ranges of the arguments. For the first of these functions, λ enters only into the initial condition, and the parallel capability of the MP-1 is not applicable. For the second and third, our CORDIC algorithm multiplies every term of the sequence ϵ_j by λ. A certain number of processors, $2^{N_{\text{CORDIC}}}$ say, is allocated to perform this task. This is called a *CORDIC cluster*.

Since the a- and b-sequences are independent, and each requires a CORDIC evaluation, an individual SLI additive or multiplicative operation is programmed

to use two CORDIC clusters. The remainder of the processor array is used to replicate this structure, so that $2^{11-N_{\text{CORDIC}}}$ SLI arithmetic operations can be executed simultaneously.

Our MasPar implementation, accordingly, provides the C functions `sli_add`, `sli_sub`, `sli_mul`, `sli_div`, `sli_op` in which the first four accept scalar arguments, and the last accepts vector arguments and an operation symbol. Comparison, type conversion, input and output operations are provided also.

The effect of varying the number of processors per CORDIC cluster was studied in [11]. For single operations, allowing N_{CORDIC} to be nonintegral, the minimum execution time occurred with 20 processors. For vector operations, assuming the vector length fits the array exactly, the minimum time occurred at the maximum vector length, which corresponds to only one processor per CORDIC cluster. The compromise $N_{\text{CORDIC}} = 3$ was recommended for the MP-1 because interprocessor communication within groups of 16 processors is especially efficient.

The MasPar 'computer arithmetic laboratory' [1] is intended to be somewhat indicative of tradeoffs that could be expected in hardware implementations of novel computer arithmetic systems. Although communication between neighboring SIMD processors is not directly comparable to communication in hardware, the results described above show that parallel techniques can be used to speed up SLI arithmetic operations.

4. Modified SLI Arithmetic Algorithms

In this section we describe modifications of the previous algorithms which are well-suited to SIMD parallel implementation. The modified algorithms can also be used to advantage in a serial implementation. The modified algorithms for SLI addition and subtraction were first presented in [1]. They are reviewed briefly here for completeness and ease of reference and then extended to multiplication, division and several basic linear algebra operations.

4.1. The Addition and Subtraction Algorithms. We retain the restrictions (14) of the preceding section. Then the basic problem of SLI addition or subtraction is to find z and its associated sign $r_Z = \pm 1$ such that

$$Z = \phi(z)^{r_Z} = \phi(x)^{r_X} \pm \phi(y)^{r_Y} = X \pm Y. \tag{29}$$

We must consider the cases (15–17).

The algorithm begins in the same way for all cases. As before, we denote the level and index of X, Y by $l_X, f_X; l_Y, f_Y$ respectively so that $l_X = [x], l_Y = [y]$, $f_X, f_Y \in [0, 1)$ and

$$x = l_X + f_X, \qquad y = l_Y + f_Y. \tag{30}$$

In addition to the a-sequence in (18), which we now denote as $a_j(x)$, we define $a_j(y) = 1/\phi(y - j)$ by

$$a_{l_Y-1}(y) = e^{-f_Y}, \qquad a_{j-1}(y) = \exp\left(-1/a_j(y)\right) \quad (j = l_Y - 1, \ldots, 1); \tag{31}$$

compare (21). Then the starting value for computing the c-sequence can be expressed as

$$c = \begin{cases} 1 \pm a_0(x)/a_0(y) & \text{(large)}, \\ 1 \pm a_0(x)a_0(y) & \text{(mixed)}, \\ 1 \pm a_0(y)/a_0(x) & \text{(small)}. \end{cases} \tag{32}$$

This definition is equivalent in all mathematical, but not numerical, respects to (25).

Some implementation details are omitted here. For example, the division in the large case of (32) could, for fixed finite precision arithmetic, take the form of $0/0$. However, under our assumptions $a_0(x) \leq a_0(y)$ so that if $a_0(x) = 0$ then one of $c = 0, 1$ or 2 is appropriate. Such considerations were dealt with, for the various cases, in [4] and [6] and can be similarly treated here. The remainder of the addition and subtraction algorithm is performed as described in Section 3 of this paper, with the understanding that the a-sequence in (26–28) is $a_j(x)$. The complete modified algorithm for SLI addition and subtraction is summarized for reference and comparison as Algorithm 1.

Algorithm 1. Modified SLI Addition and Subtraction of Positive Arguments

Input	$(r_X, x), (r_Y, y)$, and $s_{\text{op}} = +1$ (for addition) or $s_{\text{op}} = -1$ (for subtraction)
Initialize	Booleans *large, mixed, small*, and $r_Z = r_X$
Compute	a-sequence $\xi_j = a_j(x)$ and $a_0(y)$
	$j = 1$
	If *large* then $c = 1 + s_{\text{op}}\xi_0/a_0(y)$
	if $c < \xi_0$ then $r_Z = -r_X, h = -\ln c/\xi_0$, go to h-step
	If *mixed* then $c = 1 + s_{\text{op}}\xi_0 a_0(y)$
	if $c < \xi_0$ then $r_Z = -r_X, h = -\ln c/\xi_0$, go to h-step
	If *small* then $c = 1 + s_{\text{op}}a_0(y)/\xi_0$
	if $c\xi_0 \geq 1$ then $r_Z = -r_X, z = 1 + \ln c\xi_0$, go to Output
	If $\ell_X = 1$ then $h = f_X + r_X \ln c$, go to h-step
	else $c = 1 + r_X\xi_1 \ln c$
	While $j < \ell_X - 1$ and $c \geq \xi_j$
	$j = j + 1, c = 1 + \xi_j \ln c$
	If $c < \xi_j$ then $z = j + c/\xi_j$, go to Output
	$j = \ell_X, h = f_X + \ln c$
h-step	While $h \geq 1$
	$j = j + 1, h = \ln h$
	$z = j + h$
Output	(r_Z, z)

The stopping conditions for the c-sequence are simpler than they appear. For $r_X = +1$, for example, the first condition governs addition; the second governs subtraction. At most one step of the h-sequence is needed for addition. More are needed only for the most severe cases of cancellation.

The overall structure of Algorithm 1 is simpler than the original SLI algorithm in that the special forms of the b-sequence (22) and (24) are not needed. Of course the error analysis of the algorithm is different but that is not the subject of the

present work. The complexity of the algorithm in terms of its use of special exponential and logarithmic functions is not significantly different but there is a more natural parallelism in the computation of the two a-sequences that facilitates SIMD computation, and perhaps also computation in hardware.

Extension of Algorithm 1 to extended summation similar to that described in [14] for the original algorithm is readily achieved. This is summarized in Algorithm 2 for the computation of

$$Z = s_Z \phi\,(z)^{r_Z} = \sum_{i=0}^{N} s_i \phi(x_i)^{r_i} = \sum_{i=0}^{N} X_i$$

where we assume that $\phi(x_0)^{r_0} \geq \phi(x_i)^{r_i}$ for all i and that $s_0 = +1$.

Algorithm 2. Modified SLI Summation of N Arguments
Input $(s_i, r_i, x_i)_{i=0}^{N}$ with $s_0 = +1$ and $|X_0| \geq |X_i|$ for $i \geq 1$
Initialize Booleans *large*, *small*, and $r_Z = r_0$
Compute a-sequence $\xi_j = a_j(x_0)$ and $a_0(x_i)$ for $i \geq 1$
$j = 1$
If *large* then $c = 1 + \sum s_i \xi_0 a_0(x_i)^{-r_i}$, $s_Z = \operatorname{sgn} c, c = |c|$
if $c < \xi_0$ then $r_Z = -r_0$, $h = -\ln c/\xi_0$, go to h-step
If *small* then $c = 1 + \sum s_i \xi_0^{-1} a_0(x_i)$, $s_Z = \operatorname{sgn} c, c = |c|$
if $c\xi_0 \geq 1$ then $r_Z = -r_0$, $h = \ln c\xi_0$, go to h-step
If $\ell_{X_0} = 1$ then $h = f_{X_0} + r_0 \ln c$, go to h-step
else $c = 1 + r_0 \xi_1 \ln c$
Complete the algorithm exactly as in Algorithm 1

For serial computation, Algorithm 2 represents a saving of approximately 66% since the repeated serial application of Algorithm 1 requires N such operations which typically entail $2N$ a-sequences and N c-sequences. A serial implementation of Algorithm 2 needs just $(N+1)$ a-sequences and just a single c-sequence. In a parallel environment, Algorithm 2 has essentially the same complexity as Algorithm 1: all the a-sequences can be computed simultaneously and the completion of the algorithm is unchanged. The only extra work is the *fixed-point* summation to obtain c which can use an efficient reduction algorithm. Even for terms of mixed sign it is unnecessary to exercise special care over the arrangement of this internal summation because all internal arithmetic to the SLI algorithms is fixed-point in nature.

4.2. Multiplication and Division. Again we adopt the restrictions (14), using $Y/X = (X/Y)^{-1}$ where necessary. Let $\diamond$ denote either multiplication or division, and consider (15–17) with $\diamond$ in place of $\pm$. The only cases of flipover occur for mixed multiplication with $\phi(x) < \phi(y)$, and for small division, but even these do not need special treatment in the algorithm. The following table shows that it suffices to consider only computations of the form

$$\phi(z) = \phi(u) \diamond \phi(v) \tag{33}$$

where $u \geq v \geq 1$:

Original operands	Multiplication	Division
large	$r_Z = +1,\ \phi(z) = \phi(x) * \phi(y)$	$r_Z = +1,\ \phi(z) = \phi(x)/\phi(y)$
mixed $\phi(x) \geq \phi(y)$	$r_Z = +1,\ \phi(z) = \phi(x)/\phi(y)$	$r_Z = +1,\ \phi(z) = \phi(x) * \phi(y)$
mixed $\phi(x) < \phi(y)$	$r_Z = -1,\ \phi(z) = \phi(y)/\phi(x)$	$r_Z = +1,\ \phi(z) = \phi(y) * \phi(x)$
small	$r_Z = -1,\ \phi(z) = \phi(y) * \phi(x)$	$r_Z = +1,\ \phi(z) = \phi(y)/\phi(x)$

In all four cases we have $z \geq 1$ and either $u = x, v = y$ or $u = y, v = x$. By analogy with (18), (25) and (26) we can write

$$c = a_0(v) \tag{34}$$

where $c_0 = c^{-1}$ for multiplication and $c_0 = c^{+1}$ for division. Then Algorithm 3 proceeds as the large case of Algorithm 1 with simplifications because flipover is impossible. The initialization (34) has the merit of being universally applicable. For a SIMD or potential hardware design, this may be preferable to the initialization that was used in the original serial algorithm.

Algorithm 3. Modified SLI Multiplication and Division of Positive Arguments
Input u, v, and $r = -1$ (for multiplication) or $r = +1$ (for division)
Compute a-sequence $\xi_j = a_j(u)$ and $c = a_0(v)$
$j = 1$
Complete the algorithm exactly as in Algorithm 1,
but with $\ell_X = \ell_U$, $f_X = f_U$ and $r_X = r$

5. Parallel SLI Algorithms for Basic Linear Algebra Operations

Two of the more important low level BLAs (basic linear algebra operations) are the scalar product and saxpy. The scalar product is easily performed by the simultaneous SLI elementwise multiplication of the components using Algorithm 3, followed by the summation Algorithm 2. This idea generalizes in the obvious manner to all the standard vector norms. This has been discussed using the original SLI arithmetic algorithms in [9] and [13].

It is possible to design an SLI dot-product operation which does not complete all the elementwise products but instead uses the information from the a-sequences to obtain those for the summands and therefore to reduce the complete scalar product operation to just one extended SLI operation. The saxpy operation is a fundamental operation of the Gauss elimination algorithm; it is treated similarly in Section 5.2.

5.1. Dot Product. Our objective here is to obtain

$$s_Z \phi(z)^{r_Z} = X^T Y$$

where the components of the N-vectors X, Y are stored in SLI form:

$$X^T = (X_i)_{i=1}^N = (s_{X_i} \phi(x_i)^{r_{X_i}})_{i=1}^N \text{ and } Y^T = (Y_i)_{i=1}^N = (s_{Y_i} \phi(y_i)^{r_{Y_i}})_{i=1}^N$$

The first step is to rearrange the data so that each elementwise product is of the form (33).

Simultaneously for each i we set

$$
\begin{aligned}
s_i &= s_{X_i} \cdot s_{Y_i} \\
\rho_i &= r_{X_i} \cdot r_{Y_i} \\
u_i &= \max(x_i, y_i) \\
v_i &= \min(x_i, y_i) \\
r_i &= \begin{cases} r_{X_i} & \text{if } u_i = x_i \\ r_{Y_i} & \text{otherwise} \end{cases}
\end{aligned} \tag{35}
$$

Then the required dot product given by

$$
s_Z \phi(z)^{r_Z} = \sum_{i=1}^{N} s_i \left(\phi(u_i) \cdot \phi(v_i)^{\rho_i}\right)^{r_i} \tag{36}
$$

where each of the internal operations is in the desired form. Our objective is however to compute the sum without completing these internal multiplications and divisions explicitly.

Although we do not compute these component products, it will be useful to denote them by w_i. That is, for each $i = 1, 2, \ldots, N$

$$
\phi(w_i) = \phi(u_i) \cdot \phi(v_i)^{\rho_i} \tag{37}
$$

For the extended summation Algorithm 2, the a-sequence of each component was required. Strictly, the full sequence is required only for the largest component of the sum; for the others just $a_0(w_i)$ suffices. Using (37), we have

$$
a_0(w_i) = a_0(u_i) \cdot a_0(v_i)^{\rho_i}. \tag{38}
$$

To generate the initial value for the c-sequence of the final summation without completing these products, it only remains to identify the largest component of the sum. In order to complete the summation, we shall also need to obtain the complete a-sequence for this term.

The first of these tasks is simply achieved. The function $a_0(x)$ is monotone decreasing, and the largest term in the sum corresponds to $\min\{a_0(w_i) \,:\, r_i = +1\}$ assuming this set is nonempty and to $\max\{a_0(w_i)\}$ otherwise. This extreme value can be obtained by the usual reduction process. Let $\hat{w}$ denote this largest term:

$$
\widehat{s\phi(\widehat{w})^{r}} = \widehat{s}\left(\phi(\widehat{u})\phi(\widehat{v})^{\widehat{\rho}}\right)^{\widehat{r}}
$$

and let $A_j = a_j(\hat{w})$ and $\alpha_j = a_j(\widehat{u})$. Also, we shall denote by C_j the c-sequence of the summation and by c_j that for $\phi(\widehat{u})\phi(\widehat{v})^{\widehat{\rho}}$. The sequence α_j is already known. Also A_0 is given by (38) for the appropriate i. From (34), we have $c = a_0(\widehat{v})$ and modifying the definition in the summation algorithm to this situation

$$
C = \widehat{s} \cdot A_0^{\widehat{r}} \sum_{\cdot i=1}^{N} s_i \cdot a_0(w_i)^{-r_i} \tag{39}
$$

with the various terms given by (38).

It remains only to obtain the rest of the sequence A_j. The rest of the algorithm can then be completed just as for the regular summation using the recurrence

$$
C_1 = 1 + \widehat{r} A_1 \ln C, \qquad C_j = 1 + A_j \ln C_{j-1}
$$

and any terms of the h-sequence which may be needed.

By definition, $A_j = a_j(\widehat{w}) = 1/\phi(\widehat{w} - j)$ and $c_j = \phi(\widehat{w} - j)/\phi(\widehat{u} - j)$. We already have, as usual, $c_j = 1 + \alpha_j \ln c_{j-1}$. Multiplying the first two of these, we get $A_j c_j = 1/\phi(\widehat{u} - j) = \alpha_j$ from which we deduce that

$$A_j = \frac{\alpha_j}{1 + \alpha_j \ln c_{j-1}}$$

which can be computed in parallel with c_j. It follows therefore that the required sequences can be computed in (staggered) parallel so that (c_j, A_j, C_{j-1}) are all obtained simultaneously which effectively adds just one step to the c-sequence.

The overall effect of this algorithm for the SLI dot product is that the complete operation becomes equivalent to just an extended summation (Algorithm 2) which we have already seen has essentially the same complexity as a single SLI operation. Of course we would anticipate that it is necessary to use a greater fixed-point wordlength for the reduction summation in (39) due to the fact that each summand is obtained from its factors (38).

5.2. Saxpy. In this section we turn to the vector operation

$$Z = \alpha X + Y$$

where X, Y, Z are N-vectors and α is a scalar with all components given in their SLI representations. This of course includes, as an important special case, multiplying or dividing a vector by a constant. The parallelism of the operations for the individual components of Z is apparent and so we concentrate on the single multiply-accumulate operation

$$s_Z \phi(z)^{r_Z} = s_\alpha \phi(a)^{r_\alpha} \cdot s_X \phi(x)^{r_X} + s_Y \phi(y)^{r_Y} \tag{40}$$

Although we can concentrate on just one such operation, it is necessary to observe that we cannot insist on any particular (magnitude) ordering among the operands or the partial results since all possible combinations may be encountered within a single SLI saxpy operation.

The signs s_α and s_X can be immediately combined to yield the sign s_w of the product term which we denote temporarily by

$$s_W \phi(w)^{r_W} = (s_\alpha s_X) \cdot \phi(a)^{r_\alpha} \cdot \phi(x)^{r_X}$$

The two factors of the product can also be arranged as for SLI multiplication so that (cf. (33))

$$\phi(w) = \phi(u) \diamond \phi(v)$$

with $u \geq v \geq 1$:

Original operands	$x \geq a \Rightarrow u = x,\ v = a$	$x < a \Rightarrow u = a,\ v = x$
large, $r_X = r_\alpha = +1$	$r_W = +1,\ \diamond = *$	$r_W = +1,\ \diamond = *$
mixed, $r_X = +1, r_\alpha = -1$	$r_W = +1,\ \diamond = /$	$r_W = -1,\ \diamond = /$
mixed, $r_X = -1, r_\alpha = +1$	$r_W = -1,\ \diamond = /$	$r_W = +1,\ \diamond = /$
small, $r_X = r_\alpha = -1$	$r_W = -1,\ \diamond = *$	$r_W = -1,\ \diamond = *$

Now, in a similar manner to that used for the dot product, we have

$$a_0(w) = a_0(u) \cdot a_0(v)^r \tag{41}$$

where, as in Algorithm 3, $r = \mp 1$ for multiplication and division respectively. In order to define c for this combined operation we must determine which is the larger operand for the addition. Again the decreasing nature of the function $a_0(\bullet)$ is used to help decide this as follows:

Operands		s_Z	ξ_j	c definition
large	$a_0(w) < a_0(y)$	s_W	$a_j(w)$	$c = 1 + a_0(w)/a_0(y)$
$r_W = r_Y = +1$	$a_0(w) \geq a_0(y)$	s_Y	$a_j(y)$	$c = 1 + a_0(y)/a_0(w)$
mixed	$r_W = +1, r_Y = -1$	s_W	$a_j(w)$	$c = 1 + a_0(w) \cdot a_0(y)$
	$r_W = -1, r_Y = +1$	s_Y	$a_j(y)$	$c = 1 + a_0(w) \cdot a_0(y)$
small	$a_0(w) < a_0(y)$	s_Y	$a_j(y)$	$c = 1 + a_0(w)/a_0(y)$
$r_W = r_Y = -1$	$a_0(w) \geq a_0(y)$	s_W	$a_j(w)$	$c = 1 + a_0(y)/a_0(w)$

In the cases where $\xi_j = a_j(y)$ the algorithm can now be completed exactly as in the standard SLI addition Algorithm 1. For the other cases, the quantities $\xi_j = a_j(w)$ are not readily available and must be computed from the sequences $a_j(u), a_j(v)$. It is on these cases that we concentrate. The details here are similar to those of the dot product algorithm in that again the a-sequence which is needed must be obtained on-the-fly and this entails the simultaneous computation of two related c-sequences.

At the beginning of this phase of the algorithm, for the cases where $\xi_j = a_j(w)$, we have, summarizing the above table,

$$c = 1 + a_0(w)^{r_W} \cdot a_0(y)^{-r_Y}$$

where $\xi_0 = a_0(w)$ is given by (41). We also have available the a-sequence $a_j(u)$ and the initial value of the c-sequence for the multiplication given by $b = a_0(v)$. The first step then consists of (simultaneously) computing

$$b_1 = 1 + r a_1(u) \cdot \ln b\,, \qquad \xi_1 = \frac{a_1(u)}{1 + r a_1(u) \cdot \ln b}$$

while the subsequent steps require three simultaneous (and very similar) computations:

$$c_j = 1 + \xi_j \ln c_{j-1}\,, \qquad b_{j+1} = 1 + a_{j+1}(u) \cdot \ln b_j\,, \qquad \xi_{j+1} = \frac{a_{j+1}(u)}{1 + a_{j+1}(u) \cdot \ln b_j}$$

except $c_1 = 1 + r_W \xi_1 \ln c$. These steps can be completed as far as obtaining b_{L-1}, ξ_{L-1} and c_{L-1} where $L = [u]$ is the level of u. At this stage, if further steps are needed the algorithm may be completed in a similar fashion to Algorithm 1. By definition,

$$c_{L-1} = \frac{\phi(z - L + 1)}{\phi(w - L + 1)}$$

from which it follows that

$$h_L = \phi(z - L) = \ln \phi(z - L + 1) = \ln \phi(w - L + 1) + \ln c_{L-1}$$

Now, since $\ln \phi(w - L + 1) = \ln \phi(u - L + 1) + \ln b_{L-1}$, this yields

$$h_L = f_u + \ln b_{L-1} + \ln c_{L-1} \tag{42}$$

where $f_u = u - L$ is the index of u.

Finally, if $h_L \geq 1$, the algorithm is completed by computing as many additional terms of the h-sequence as necessary. Since each of the underlying operations can increase the level by at most one, no more than two such steps are required. When $\xi_j = a_j(y)$, at most one step of the corresponding h-sequence is needed.

In a serial computing environment, each component of the resulting vector is computed using three a-sequences and two c-sequences which represents only a relatively small (approximately 17%) saving relative to performing the SLI multiplication and then the addition each requiring two a-sequences and a c-sequence. In a SIMD parallel environment (with sufficient processors) all the a-sequences are computable simultaneously as are all the c-sequences so that the complete parallel saxpy operation has similar parallel complexity to Algorithm 1 for scalar SLI addition.

6. Conclusions

In this paper we have demonstrated that fundamental vector and scalar processes in SLI arithmetic have essentially the same computational complexity through effective use of parallel recursive algorithms. The main source of this parallelism is in the simultaneous computation of sequences which are independent of one another, essentially one for each component of the vector operands. This kind of parallelism, which is unavailable in floating-point arithmetic, makes SLI especially attractive for numerical linear algebra.

We discussed also a software implementation that is being developed on a 4096-processor SIMD parallel computer system. This system is ideal for demonstrating the parallel advantages of SLI algorithms with a view toward a possible future hardware design. Currently the software implementation includes individual SLI arithmetic operations on scalar and vector operands. It is being extended to include all the algorithms discussed in this paper. Ultimately it will be used to solve linear algebraic systems in SLI arithmetic for demonstration and comparison purposes.

All SLI algorithms can be executed in fixed-point arithmetic. A suitable number of guard digits is needed, as determined by appropriate error analysis. Results have been obtained by a priori error analysis for individual arithmetic operations in earlier papers, and these were incorporated into our software implementation. Further work in error analysis will be the subject of future papers.

References

1. M. A. Anuta, D. W. Lozier, and P. R. Turner, *The MasPar MP-1 as a computer arithmetic laboratory*, J. Res. Nat. Inst. Standards and Technology **101** (1996), to appear.
2. C. W. Clenshaw, D. W. Lozier, F. W. J. Olver, and P. R. Turner, *Generalized exponential and logarithmic functions*, Comput. Math. Appl. **12B** (1986), 1091–1101.
3. C. W. Clenshaw and F. W. J. Olver, *Beyond floating point*, J. Assoc. Comput. Mach. **31** (1984), 319–328.
4. ———, *Level-index arithmetic operations*, SIAM J. Numer. Anal. **24** (1987), 470–485.
5. C. W. Clenshaw, F. W. J. Olver, and P. R. Turner, *Level-index arithmetic: An introductory survey*, Numerical Analysis and Parallel Processing (P. R. Turner, ed.), Springer-Verlag, 1989, pp. 95–168.
6. C. W. Clenshaw and P. R. Turner, *The symmetric level-index system*, IMA J. Numer. Anal. **8** (1988), 517–526.

7. G. H. Golub and C. F. van Loan, *Matrix Computations*, 2nd ed., Johns Hopkins University Press, Baltimore, MD, 1989.
8. D. W. Lozier and F. W. J. Olver, *Closure and precision in level-index arithmetic*, SIAM J. Numer. Anal. **27** (1990), 1295–1304.
9. D. W. Lozier and P. R. Turner, *Robust parallel computation in floating-point and SLI arithmetic*, Computing **48** (1992), 239–257.
10. _____, *Error-bounding in level-index computer arithmetic*, Numerical Methods and Error Bounds (G. Alefeld and J. Herzberger, eds.), Akademie Verlag, Berlin, 1996, to appear.
11. N. Schabanel and P. R. Turner, *Parallelization and parallel implementation on the MasPar of SLI arithmetic*, Mathematics Department, U. S. Naval Academy, Annapolis, MD 21402, September 13, 1995.
12. C. W. Schelin, *Calculator function approximation*, Amer. Math. Monthly **90** (1983), 317–325.
13. P. R. Turner, *A software implementation of SLI arithmetic*, Proc. 9th Symposium on Computer Arithmetic, IEEE Computer Society Press, Washington, DC, 1989, pp. 18–24.
14. _____, *Implementation and analysis of extended SLI operations*, Proc. 10th Symposium on Computer Arithmetic, IEEE Computer Society Press, Washington, DC, 1991, pp. 118–126.