Basic Linear Algebra Operations in SLI Arithmetic

Michael A. Anuta[1], Daniel W. Lozier[2], Nicolas Schabanel[3] and Peter R. Turner[4]

[1] Cray Research Inc., Calverton, MD 20705, USA
[2] National Institute of Standards and Technology, Gaithersburg, MD 20899, USA
[3] École Normale Supérieure de Lyon, Lyon, France
[4] United States Naval Academy, Annapolis, MD 21402, USA

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 are parallelizable to some extent, particularly when applied to extended sums or products. 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; for a more detailed progress report, see [1]. 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. 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 [8] consisting of a scalar times a vector plus a vector. All these operations, especially the BLAs, can benefit from the parallel execution of appropriate algorithms.

Extending [13], a parallel software implementation of individual SLI arithmetic operations was developed in 1995 by Schabanel and Turner [12]. This is part of an envisioned 'computer arithmetic laboratory' which will implement different kinds of computer arithmetic and compare them on representative numerical problems; see Anuta, Lozier and Turner [2].

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 [4, 5]. The *symmetric* form of the LI system was developed in 1988 by Clenshaw and Turner [7]. See also the survey [6].

An account of *generalized* logarithmic and exponential functions is given in [3]. The LI and SLI computer arithmetic systems are based on a generalized exponential function ϕ and its inverse function, a generalized logarithm ψ. These functions are defined by $\phi(x) = \psi(x) = x$ when $0 \le x < 1$ and by the relations

$$\phi(x) = e^{\phi(x-1)}, \qquad \psi(x) = 1 + \psi(\ln x) \tag{1}$$

when $x \ge 1$. Repeated application of the relation for ψ is possible up to ℓ times, where $\ell \ge 1$ is determined by the condition that the ℓ-fold repeated logarithm of x lies in the interval $[0, 1)$. By definition, ℓ is the *level* of x, and the ℓ-fold repeated logarithm is the *index* of x. When $0 \le x < 1$, the level of x is zero and the index of x is x itself.

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

$$X = s_X \phi(x) \tag{2}$$

where $x = \psi(|X|)$, then x is rounded and stored in ordinary $(w-1)$-bit fixed-point format. As a consequence of the exceedingly rapid growth of ϕ, Lozier and Olver proved in [9] that the *finite* set of w-bit LI representations with level not more than 6 is *closed* under individual arithmetic operations (excluding, of course, division by zero), provided that w does not exceed 5.5 million or so. Therefore, 3 bits in the integer and $w-4$ bits in the fractional part of x always suffices in practice.

The LI representation has, in effect, an 'accumulation point' at infinity, but not at zero. The SLI representation of $X \ne 0$ is

$$X = s_X \phi(x)^{r_X} \tag{3}$$

where $r_X = +1$ if $|X| \ge 1$, $r_X = -1$ if $|X| < 1$, and $x = \psi(|X|^{r_X}) \ge 1$. This 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.* Comparisons of SLI to typical floating-point systems for $w = 32, 64, 128$ are given in [9] and [11].

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 problems of overflow and underflow 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 [10], add significantly to the cost of creating and maintaining robust software.

2 Original SLI Arithmetic Algorithms

In this section, to fix notation, we review the original recursive algorithms given in [7]. If $\circ$ is an arithmetic operation, the problem is to solve the equation

$$Z = s_Z \phi(z)^{r_Z} = s_X \phi(x)^{r_X} \circ s_Y \phi(y)^{r_Y} = X \circ Y \tag{4}$$

for s_Z, r_Z, z given s_X, s_Y, r_X, r_Y, x, y. It suffices to restrict

$$s_Z = s_X = s_Y = 1, \qquad X \ge Y > 0. \tag{5}$$

The more difficult operations are the additive ones, for which three cases exist:

$$\begin{array}{ll} \text{large} & \phi(z)^{r_Z} = \phi(x)^{+1} \pm \phi(y)^{+1}, \\ \text{mixed} & \phi(z)^{r_Z} = \phi(x)^{+1} \pm \phi(y)^{-1}, \\ \text{small} & \phi(z)^{r_Z} = \phi(x)^{-1} \pm \phi(y)^{-1}. \end{array} \tag{6}$$

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 ℓ_X = level(X) = [x], f_X = index(X) = x − [x], and similarly for ℓ_Y, ℓ_Z, f_Y, f_Z. The algorithm generates the sequences

$$a_j = 1/\phi(x-j), \qquad c_j = \phi(z-j)/\phi(x-j), \tag{7}$$

the appropriate form of the sequence

$$\begin{array}{ll} \text{large} & b_j = \phi(y-j)/\phi(x-j), \\ \text{mixed} & b_j = 1/\phi(y-j), \\ \text{small} & b_j = \phi(x-j)/\phi(y-j), \end{array} \tag{8}$$

and in some situations the h-sequence $h_j = \phi(z-j)$. First the a- and b-sequences are generated by backward recurrence from

$$a_{\ell_X - 1} = e^{-f_X}, \qquad a_{j-1} = e^{-1/a_j} \tag{9}$$

and

$$\begin{array}{lll} \text{large} & b_{\ell_Y-1} = a_{\ell_Y-1}e^{f_Y}, & b_{j-1} = e^{(b_j-1)/a_j}, \\ \text{mixed} & b_{\ell_Y-1} = e^{-f_Y}, & b_{j-1} = e^{-1/b_j}, \\ \text{small} & b_{\ell_X-1} = e^{-\phi(y-\ell_X)}e^{f_X}, & b_{j-1} = e^{(b_j-1)/a_j b_j}. \end{array} \tag{10}$$

Then $c = c_0^{r_X}$ (the starting value for the c-sequence) is given by

$$\text{large } \textit{and} \text{ small} \qquad c = 1 \pm b_0, \qquad \text{mixed} \qquad c = 1 \pm a_0 b_0. \tag{11}$$

Since

$$\phi(z)^{r_X} = c\phi(x)^{r_X} = c/a_0^{r_X}, \tag{12}$$

flipover occurs when $r_X = +1$ and $c < a_0$, or when $r_X = -1$ and $a_0 c \geq 1$. In the latter case, $z = 1 + \ln a_0 c$. In the other two cases we generate the h-sequence from

$$h_1 = -\ln c/a_0, \qquad h_j = \ln h_{j-1} \qquad (j = 2, 3, \ldots) \tag{13}$$

until $h_j \in [0, 1)$, then $z = j + h_j$.

Now suppose flipover does not occur. If $\ell_X = 1$ we generate the h-sequence starting from $h_1 = f_X + r_X \ln c$. If $\ell_X > 1$ we generate the c-sequence from

$$c_1 = 1 + r_X a_1 \ln c, \qquad c_j = 1 + a_j \ln c_{j-1} \qquad (j = 2, 3, \ldots) \tag{14}$$

until either

1. $c_j < a_j$ and $j \leq \ell_X - 1$, in which case $z = j + c_j/a_j$, 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 [7] considers the 'working precisions' needed to limit the rounding errors in the algorithms presented above to the size of the inherent errors; see also [5] and [14]. The analysis shows, for a word length $w = 32$ bits, that it suffices to compute and store all sequences to 6 bits before and 36 bits after the binary point.

3 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 for SLI addition and subtraction were first presented in [2]. They are reviewed briefly here and then extended to multiplication, division and basic linear algebra operations.

3.1 The Addition and Subtraction Algorithms

Again we consider the problem (4) under the restrictions (5), and we take into account all of the cases in (6). The new algorithm begins in the same way for all cases. In addition to the a-sequence (7), which we now denote as $a_j(x)$, we define $a_j(y) = 1/\phi(y-j)$; this is generated by a recurrence analogous to (9). Then the starting value for computing the c-sequence can be expressed as

$$c = 1 \pm a_0(x)^{r_X}/a_0(y)^{r_Y} \tag{15}$$

This is equivalent in all mathematical, but not numerical, respects to (11).

Some implementation details are omitted here. For example, the division in the large case of (15) could, for fixed finite precision arithmetic, take the form of 0/0. However, under our assumptions $a_0(x) \le a_0(y)$ so that if $a_0(x) = 0$ then one of $c = 0, 1$ or 2 is appropriate. Such considerations were dealt with in [5] and [7] and can be similarly treated here. The remainder of the addition and subtraction algorithm is performed as before. The complete modified algorithm for SLI addition and subtraction is summarized as Algorithm 1:

```
Algorithm 1. Modified SLI Addition and Subtraction of Positive Arguments
Input       $(r_X, x), (r_Y, y)$, $s_{\mathrm{op}} = +1$ (for addition) or $-1$ (for subtraction)
Initialize  $r_Z = r_X$
Compute     $a$-sequences $\xi_j = a_j(x)$ and $a_j(y)$ in parallel
            $j = 1$
            If $r_X = +1$ then $c = 1 + s_{\mathrm{op}}\xi_0 a_0(y)^{-r_Y}$
                if $c < \xi_0$ then $r_Z = -r_X, h = -\ln c/\xi_0$, go to h-step
            else $c = 1 + s_{\mathrm{op}} a_0(y)/\xi_0$
                if $c\xi_0 \ge 1$ 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 $c \ge \xi_j$
                $j = j + 1, c = 1 + \xi_j \ln c$
            If $c < \xi_j$ then $z = j + c/\xi_j$, go to Output
            $j = \ell_X, h = f_X + \ln c$
h-step      While $h \ge 1$
                $j = j + 1, h = \ln h$
            $z = j + h$
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.

Algorithm 1 is simpler than the original SLI algorithm in that only the mixed-case b-sequence in (8) is used. 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.

The following algorithm extends Algorithm 1 to 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  $r_Z = r_0$
Compute     $a$-sequences $\xi_j = a_j(x_0)$ and $a_j(x_i)$ for $i \geq 1$ in parallel
            $j = 1$
            If $r_0 = +1$ then $c = 1 + \sum s_i\xi_0 a_0(x_i)^{-r_i}$, $s_Z = \text{sgn} c, c = |c|$
                if $c < \xi_0$ then $r_Z = -r_0$, $h = -\ln c/\xi_0$, go to h-step
            else $c = 1 + \sum s_i\xi_0^{-1}a_0(x_i)$, $s_Z = \text{sgn} c, c = |c|$
                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%: repeated application of Algorithm 1 requires $2N$ a-sequences and N c-sequences, whereas Algorithm 2 needs just $(N+1)$ a-sequences and a single c-sequence. In a parallel environment, the two algorithms have essentially the same complexity: all the a-sequences are computed simultaneously and the completion of the algorithm is unchanged. The extra work to obtain c can use an efficient *fixed-point* reduction algorithm for summation. No special arrangement of the work is necessary because all the internal arithmetic is fixed-point.

3.2 Multiplication and Division

Again we adopt the restrictions (5), using $Y/X = (X/Y)^{-1}$ where necessary. Let $\diamond$ denote either multiplication or division, and consider (6) with $\diamond$ in place of $\pm$. The following table shows that it suffices to consider only computations of the form

$$\phi(z) = \phi(u) \diamond \phi(v) \tag{16}$$

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 (7), (11) and (12) we can write

$$c = a_0(v) \tag{17}$$

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 (17) 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    $a$-sequences $\xi_j = a_j(u)$ and $a_j(v)$ in parallel
           $j = 1, c = a_0(v), \ell_X = \ell_U, f_X = f_U, r_X = r$
           Complete the algorithm exactly as in Algorithm 1
```

4 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. 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. The complete scalar product operation is then just one extended SLI operation. The saxpy operation is treated similarly in Section 4.2.

4.1 Dot Product

Our objective here is to obtain $s_Z\phi(z)^{r_Z} = X^TY$ 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 (16).

Simultaneously for each i we set

$$\begin{aligned} &s_i = s_{X_i} \cdot s_{Y_i}, \rho_i = r_{X_i} \cdot r_{Y_i} \\ &u_i = \max(x_i, y_i), v_i = \min(x_i, y_i) \\ &r_i = \begin{cases} r_{X_i} \text{ if } u_i = x_i \\ r_{Y_i} \text{ otherwise} \end{cases} \end{aligned} \tag{18}$$

Then the required dot product given by

$$s_Z \phi(z)^{r_z} = \sum_{i=1}^{N} s_i \left(\phi(u_i) \cdot \phi(v_i)^{\rho_i}\right)^{r_i} \tag{19}$$

where each of the internal operations is in the desired form.

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

$$\phi(w_i) = \phi(u_i) \cdot \phi(v_i)^{\rho_i} \tag{20}$$

For Algorithm 2, the a-sequence of each component was required. The full sequence is required only for the largest component of the sum; for the others just $a_0(w_i)$ suffices. Using (20), we have

$$a_0(w_i) = a_0(u_i) \cdot a_0(v_i)^{\rho_i}. \tag{21}$$

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

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

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

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 (21) for the appropriate i. From (17), we have $c = a_0(\widehat{v})$ and modifying the definition in the summation algorithm to this situation

$$C = \widehat{s} \cdot A_0^{\widehat{r}} \sum_{i=1}^{N} s_i \cdot a_0(w_i)^{-r_i} \tag{22}$$

with the various terms given by (21).

It remains to obtain the rest of the sequence A_j. The algorithm is completed as in the regular summation using the recurrence $C_1 = 1 + \widehat{r} A_1 \ln C$, $C_j = 1 + A_j \ln C_{j-1}$ 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 $c_j = \phi(\widehat{w}-j)/\phi(\widehat{u}-j)$. As usual, $c_j = 1 + \alpha_j \ln c_{j-1}$. Multiplying the first two of these, $A_j c_j = 1/\phi(\widehat{u} - j) = \alpha_j$

and hence $A_j = \alpha_j/(1+\alpha_j \ln c_{j-1})$ which can be computed in parallel with c_j. It follows 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 SLI dot product is thus equivalent to an extended summation (Algorithm 2) which has essentially the same complexity as a single SLI operation. A greater fixed-point wordlength is needed for the reduction summation in (22) since each summand is obtained from its factors (21).

4.2 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 all given by their SLI representations. The parallelism of the operations for the individual components of Z is apparent. We concentrate on the single multiply-accumulate operation

$$s_Z \phi(z)^{r_Z} = s_\alpha \phi(a)^{r_\alpha} \cdot s_X \phi(x)^{r_X} + s_Y \phi(y)^{r_Y} \tag{23}$$

Although we concentrate on just one such operation, we cannot insist on any particular (magnitude) ordering among the operands or the partial results since all possible combinations may be encountered within a 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 in the form (16)), i.e. $\phi(w) = \phi(u) \diamond \phi(v)$ 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

$$a_0(w) = a_0(u) \cdot a_0(v)^r \tag{24}$$

where, as in Algorithm 3, $r = \mp 1$ for multiplication and division respectively. To define c for this combined operation we must identify the larger operand for the addition. The decreasing function $a_0(\bullet)$ helps 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	$a_j(w)$	$c = 1 + a_0(y)/a_0(w)$

If $\xi_j = a_j(y)$ the algorithm can be completed exactly as in Algorithm 1. For the other cases, the quantities $\xi_j = a_j(w)$ are not readily available and must be computed from the sequences $a_j(u), a_j(v)$. The details of these cases are similar to those of the dot product in that an a-sequence is needed 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 (24). The a-sequence $a_j(u)$ and the initial value of the c-sequence for the multiplication given by $b = a_0(v)$ are available. The first step consists of computing $b_1 = 1 + ra_1(u) \cdot \ln b$ and $\xi_1 = a_1(u)/(1 + ra_1(u) \cdot \ln b)$. Subsequent steps require three 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, and they can be computed in parallel with minimal delays. Any further steps that are needed may be completed much as in Algorithm 1. By definition, $c_{L-1} = \phi(z - L + 1)/\phi(w - L + 1)$ 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

$$h_L = f_u + \ln b_{L-1} + \ln c_{L-1} \tag{25}$$

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

Finally, if $h_L \geq 1$, the algorithm is completed by computing additional terms of the h-sequence as necessary. Since each underlying operation 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 approximately a 17% saving relative to SLI multiplication and then 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.

5 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 may make SLI attractive for numerical linear algebra. A SIMD parallel computer system is ideal for demonstrating the parallel advantages of SLI algorithms.

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, N. Schabanel, and P. R. Turner, *Basic linear algebra operations in SLI arithmetic*, Nat. Inst. Stand. and Tech. Report NISTIR 5811, March 1996, 15 pages.
2. M. A. Anuta, D. W. Lozier, and P. R. Turner, *The MasPar MP-1 as a computer arithmetic laboratory*, J. Res. Nat. Inst. Stand. and Tech. **101** (1996), 165–174.
3. 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.
4. C. W. Clenshaw and F. W. J. Olver, *Beyond floating point*, J. Assoc. Comput. Mach. **31** (1984), 319–328.
5. ———, *Level-index arithmetic operations*, SIAM J. Numer. Anal. **24** (1987), 470–485.
6. 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.
7. C. W. Clenshaw and P. R. Turner, *The symmetric level-index system*, IMA J. Numer. Anal. **8** (1988), 517–526.
8. G. H. Golub and C. F. van Loan, *Matrix Computations*, 2nd ed., Johns Hopkins University Press, Baltimore, MD, 1989.
9. D. W. Lozier and F. W. J. Olver, *Closure and precision in level-index arithmetic*, SIAM J. Numer. Anal. **27** (1990), 1295–1304.
10. D. W. Lozier and P. R. Turner, *Robust parallel computation in floating-point and SLI arithmetic*, Computing **48** (1992), 239–257.
11. ———, *Error-bounding in level-index computer arithmetic*, Numerical Methods and Error Bounds (G. Alefeld and J. Herzberger, eds.), Akademie Verlag, Berlin, 1996, pp. 138–145.
12. 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.
13. P. R. Turner, *A software implementation of SLI arithmetic*, Proc. ARITH9, IEEE Computer Society Press, Washington, DC, 1989, pp. 18–24.
14. ———, *Implementation and analysis of extended SLI operations*, Proc. ARITH10, IEEE Computer Society Press, Washington, DC, 1991, pp. 118–126.