elseif $\alpha_i \cdot x_{i+1} + \beta_i \cdot B^{m_{i+1}}$ overflows, then
$x_i = \alpha_i \cdot x_{i+1}/B + \beta_i \cdot B^{m_{i+1}-1}$ (carefully!)
$m_i = m_{i+1} - 1$
elseif $\alpha_i \cdot x_{i+1} + \beta_i \cdot B^{m_{i+1}}$ underflows, then
$x_i = \alpha_i \cdot x_{i+1} \cdot B + \beta_i \cdot B^{m_{i+1}+1}$ (carefully!)
$m_i = m_{i+1} + 1$
endif
endfor

The true values of $x_i$ are gotten from the computed $x_i$ and $m_i$ the same way $d_i$ is gotten from $\tilde{d}_i$ and $m_i$ as described above: If some $x_i$ is less than 1 in magnitude, multiply it by $B$ and em add 1 to $m_i$, then subtract the largest $m_i$ from all of them so the largest is now zero, and then change $x_i$ to $x_i B^{-m_i}$.

I have not debugged this pseudocode, but I think it is basically correct.

Alan Edelman points out that these can all be blocked in straigthforward ways.

6. Compute $y = D_2 y_2$ for all $i$ in one parallel operation.

To protect against over/underflow, one can modify this scheme in one of two ways. The easiest way is to compute $\log f_i$ (assume for the moment all $f_i > 0$), then $\log d_i = \sum_{j=1}^{i} \log f_i$ using an add-scan, and finally $\log e_i = -\log d_i - \log a_{i+1}$. Let $\bar{d} = \max \log d_i$ and $\bar{e} = \max \log e_i$. Replace $\log d_i$ by $\log d_i - \bar{d}$ and replace $\log e_i$ by $\log e_i - \bar{e}$. Exponentiate the new $\log d_i$ and $\log e_i$ to get scaled values of $e_i$ and $d_i$ the largest of each of which is 1. Now perform steps (4) through (6) of the above algorithm.

This has the added cost of a logarithm and exponent, and loses a little accuracy because of them too. But it will not overflow and almost certainly not underflow dangerously. (There are other choices of $\bar{e}$ and $\bar{d}$ that might be marginally safer against underflow.) The signs of the $f_i$ can be accumulated and applied with a multiply scan operation involving only $\pm 1$'s.

A better way to protect against overflow is to modify the multiply-scan operation as follows. Instead of computing $d_i = \prod_{j=0}^{i} f_i$, one computes a $\tilde{d}_i$ and integer $m_i$ such that $\tilde{d}_i$ cannot over/underflow, and $d_i = \tilde{d}_i B^{m_i}$, where $B$ is a big power of the radix near the overflow threshold. Here is a code, which can obviously be "scanned" for computing $\tilde{d}_i$ and $m_i$:

$\tilde{d}_0 = 1$; $m_0 = 0$
for $j = 1, n$
    if $\tilde{d}_{i-1} \cdot f_i$ neither overflows nor underflows then
        $\tilde{d}_i = \tilde{d}_{i-1} \cdot f_i$
        $m_i = m_{i-1}$
    elseif $\tilde{d}_{i-1} \cdot f_i$ would overflow then
        $\tilde{d}_i = \tilde{d}_{i-1} \cdot f_i / B$ (computed carefully!)
        $m_i = m_{i-1} + 1$
    elseif $\tilde{d}_{i-1} \cdot f_i$ would underflow then
        $\tilde{d}_i = \tilde{d}_{i-1} \cdot f_i \cdot B$ (computed carefully!)
        $m_i = m_{i-1} - 1$
    endif
endfor

If $B$ is close to overflow, at this point one should take all $\tilde{d}_i$ less than 1 in magnitude, multiply them by $B$ and subtract 1 from their corresponding $m_i$. When the $m_i$ are available, the largest one can be subtracted from all of them (so the largest is now 0), and then $d_i = \tilde{d}_i B^{m_i}$ computed without fear of overflow.

The third approach is to run the recurrence $x_i = (y_i - b_i x_{i+1})/a_i \equiv \alpha_i x_{i+1} + \beta_i$ sequentially, scaling if necessary as one goes. The code is similar in its use of a counter $m_i$ to the last code above:

$\tilde{x}_n = \beta_n$; $m_n = 0$
for $i = n - 1$ downto 1
    if $\alpha_i \cdot x_{i+1} + \beta_i \cdot B^{m_{i+1}}$ neither over/underflows,
        $x_i = \alpha_i \cdot x_{i+1} + \beta_i \cdot B^{m_{i+1}}$
        $m_i = m_{i+1}$

## 3 Exploiting extra range

If we have extra exponent range available, we can greatly diminish the amount of time spent testing and scaling. If the data is in IEEE single precision, then products of 8 such (normalized!) numbers can be computed without over/underflow in IEEE double precision, and products of 128 in IEEE extended. If the data is IEEE double then products of 16 can be computed in IEEE extended. Thus, any scaling tests would only have to be done after 8, 16 or 128 products.

## 4 Extensions

Of course, it would be nice if the user could supply his or her own data type and associative operator, and have the system perform parallel prefix. Given the details of exception handling as described above, it would be nice to have these basic floating point operations be done automatically, rather than expecting the user to handle them.

## Appendix: Comments on Inverse Iteration on the CM-2[1]

The task at hand is to solve $Bx = y$ where $B$ is an $n$ by $n$ upper bidiagonal matrix and $x$ and $y$ are $n$-vectors. Let $a_1, \ldots, a_n$ be the diagonal entries of $B$ and $-b_1, \ldots, -b_{n-1}$ be the superdiagonal entries. The usual recurrence is $x_i = (y_i - b_i x_{i+1})/a_i \equiv \alpha_i x_{i+1} + \beta_i$, for $i = n$ down to $i = 1$, with $b_n \equiv 0$ and $x_{n+1} \equiv 0$. I can think of three ways to solve this, with various kinds of immunity against overflow. I have not analyzed the accuracy of all these (they are not all equivalent to evaluating the recurrence sequentially, which has nearly perfect backward error), but I don't see any obvious dangers. I have tried to use the scan operation to the extent possible, assuming only scans for floating-point add and floating-point multiply (although the best solution would involve modifying the multiply scan).

The first two, and most parallel, methods involve the following factorization: $E = D_1 \cdot B \cdot D_2$, where $D_1$ and $D_2$ are diagonal, and $E$ is bidiagonal with 1 on the diagonal and $-1$ off. $D_2 = diag(d_0, \ldots, d_{n-1})$ is given by $d_0 = 1$, and $d_i = \prod_{j=1}^{i}(a_j/b_j)$. $D_1 = diag(e_0, \ldots, e_{n-1})$ is given by $e_i = 1/(d_i a_{i+1})$. One can also verify $E^{-1}$ is upper triangular with all ones on and above the diagonal. Thus, computing $y = B^{-1}x = D_2 E^{-1} D_1 x$ involves the following:

1. Compute $f_0 = 1$, $f_i = a_i/b_i$ for all $i$ in one parallel step.
2. Compute $d_i = \prod_{j=0}^{i} f_i$ for all $i$ using a multiply-scan operation.
3. Compute $e_0 = 1$, $e_i = 1/(d_i a_{i+1})$ for all $i$ in two parallel steps.
4. Compute $y_1 = D_1 x$ for all $i$ in one parallel operation.
5. Compute $y_2 = E^{-1} y_1$ for all $i$ using an add-scan operation (which is all multiplying by $E^{-1}$ is).

[1] These are some early notes written for J.-P. Brunet.

*Addition/Subtraction: compute $z \cdot r^k = (x \cdot r^n) \pm (y \cdot r^m)$. Statements in braces are unnecessary on machines that returned wrapped results on over/underflow. It assumes there are sticky overflow and underflow flags as in IEEE arithmetic. Multiplications and divisions by $r$ can be done by modifying the exponent directly. It assumes round to nearest mode and flush to zero underflow (i.e.* not *gradual underflow), although the changes to account for other assumptions are simple. Besides $r$ the machine constant $t$ = underflow_threshold/machine_epsilon will be used. I have arranged the "if" tests in decreasing order of likelihood of their being executed.*

if $(m = n)$ then
    $z = x \pm y$
    $k = m$
    if (overflow) then
        $\{z = (x/r) \pm (y/r)\}$
        $k = k + 1$
    elseif (underflow) then
        $\{z = (x \cdot r) \pm (y \cdot r)\}$
        $k = k - 1$
    endif
elseif $(m = n - 1)$ then
    $z = x \pm (y/r)$  /* no overflow possible if round to nearest */
    $k = n$
    if $0 < |z| < t$ then
        $z = (x \cdot r) \pm y$
        $k = m$
    endif
elseif $(m = n + 1)$ then
    $z = y \pm (x/r)$  /* no overflow possible if round to nearest */
    $k = m$
    if $0 < |z| < t$ then
        $z = (y \cdot r) \pm x$
        $k = n$
    endif
elseif $(m < n - 1)$ then
    $z = x$
    $k = n$
elseif $(n < m - 1)$ then
    $z = y$
    $k = m$
endif

It would be interesting to benchmark this parallel prefix operation both with and without the protection against over/underflow I propose here, to see how much this protection costs us.

| Safe Limits for Exponentiation | | | |
|---|---|---|---|
| IEEE Single | | $r = 2$ | $r = 2^{192}$ |
| $f = 2^{128}$ | $b = 16$ | 255 | 49150 |
| | $b = 32$ | $1.67 \cdot 10^7$ | $3.22 \cdot 10^9$ |
| $f = 2^{-126}$ | $b = 16$ | 260 | 49930 |
| | $b = 32$ | $1.70 \cdot 10^7$ | $3.27 \cdot 10^9$ |
| $f = 2^{-149}$ | $b = 16$ | 219 | 42223 |
| | $b = 32$ | $1.44 \cdot 10^7$ | $2.76 \cdot 10^9$ |
| IEEE Double | | $r = 2$ | $r = 2^{1536}$ |
| $f = 2^{1024}$ | $b = 16$ | 31 | 49150 |
| | $b = 32$ | $2.09 \cdot 10^6$ | $3.22 \cdot 10^9$ |
| $f = 2^{-1022}$ | $b = 16$ | 32 | 49246 |
| | $b = 32$ | $2.10 \cdot 10^6$ | $3.22 \cdot 10^9$ |
| $f = 2^{-1074}$ | $b = 16$ | 30 | 46775 |
| | $b = 32$ | $1.99 \cdot 10^6$ | $3.06 \cdot 10^9$ |

¿From this table, we see the limiting case is taking power of the smallest subnormal number. When $b = 16$ we must clearly take the larger of the two $r$ values to get a reasonably large safe exponent, and even then it is less than 50000. Choosing $b = 32$ is clearly more reasonable. Should we choose $r = 1$ or larger $r$? I believe the larger $r$ is preferable, both because of the larger parallel prefix operations it allows, and because it appears to be easier to implement, since the representation of a number is almost unique: there are at most two ways to store a nonzero number in the form $f \cdot r^n$.

In order to implement our two operations it suffices to explain how to do addition and multiplication of numbers in this scaled format.

*Multiplication: compute $z \cdot r^k = (x \cdot r^n) \times (y \cdot r^m)$. Statements in braces are unnecessary on machines that returned wrapped results on over/underflow. It assumes there are sticky overflow and underflow flags as in IEEE arithmetic. Multiplications and divisions by $\sqrt{r}$ can be done by modifying the exponent directly.*

$z = x \cdot y$
$k = m + n$
if (overflow) then
    $\{z = (x/\sqrt{r}) \cdot (y/\sqrt{r})\}$
    $k = k + 1$
elseif (underflow) then
    $\{z = (x \cdot \sqrt{r}) \cdot (y \cdot \sqrt{r})\}$
    $k = k - 1$
endif

recurrences are enough to do 2 term recurrences). At this time, however, I have not seen any evidence that we frequently need to solve 3 or more term recurrences.

This leads us to propose the following building blocks for parallel prefix:

1. Scalar multiply parallel prefix with scaling to avoid over/underflow.

2. 2 by 2 matrix multiply parallel prefix with scaling to avoid over/underflow.

## 2 Specifications for Parallel Prefix

Basically, each floating point number $x_i$ will be replaced by a pair $(f_i, n_i)$, where $f_i$ is a floating point number and $n_i$ an integer, with the pair representing $f_i \cdot r^{n_i}$, $r$ an integer power of 2. The first problem is to choose $r$ and the number of bits to store $n_i$ so as to allow for easy implementation and the ability to do very large parallel prefix operations without fear of over/underflow. So given $r$, the number of bits $b$ in which to store $n_i$ as a signed integer, and the largest and smallest positive possible values of a floating point number, we will ask how high a power of the largest and smallest floating point numbers can be stored without over/underflow in the form $f_i \cdot r^{n_i}$.

We will only consider IEEE single and double precision formats. The largest and smallest numbers are given in the following table:

| | IEEE Single | IEEE Double |
|---|---|---|
| Approximate overflow threshold | $2^{128}$ | $2^{1024}$ |
| Underflow threshold | $2^{-126}$ | $2^{-1022}$ |
| Smallest subnormal number | $2^{-149}$ | $2^{-1074}$ |

Reasonable values for $r$ are 2, $2^{192}$ for IEEE single precision, and $2^{1536}$ for IEEE double precision. The source for these last two values is the wrapped exponent feature of IEEE arithmetic: If overflow is trapped, the floating point unit is supposed to return the true answer times $2^{-192}$ in single precision and the true answer times $2^{-1536}$ in double precision. Similary, if underflow is trapped the value returned is supposed to be either $2^{192}$ or $2^{1536}$ times the true value.

Reasonable values for $b$, the number of bits in which to store $n_i$, are 16 and 32.

The following table enumerates the approximate highest powers to which a floating point number $f$ can be safely raised using the scaled format as a function of $b$ and $r$:

So we need to compute $N = \text{ParPrefix}(M, \cdot)$, where each $M_i$ is a 2 by 2 matrix, and $\cdot$ is matrix multiply. Alternatively, we could use the following, equivalent algorithm:

$f = \text{ParPrefix}(\eta, \cdot)$
$g = \tau / f$ (componentwise vector division)
$h = \text{ParPrefix}(g, +)$
$x = h \cdot f$ (componentwise vector multiplication)

Unfortunately, this is very nonrobust because $f$ frequently overflows or underflows. Even in IEEE double precision, with a range of $10^{\pm 308}$, it does not take many consecutive floating point multiplies to get a number out of range. This can be partly resolved by taking logarithms of the $\eta_i$ and doing a $\text{ParPrefix}(\log \eta, +)$ operation, but this is not a satisfactory solution because it is slower and less accurate (see the appendix).

Let $T$ be an $n$ by $n$ symmetric tridiagonal matrix with diagonal entries $a_1, \ldots, a_n$ and offdiagonal entries $b_1, \ldots, b_{n-1}$. In order to find its eigenvalues, we need to solve the two term recurrence

$$w_i = (a_i - \sigma) w_{i-1} - b_{i-1}^2 w_{i-2} \quad . \tag{2}$$

(The number of sign changes in the sequence of $w_i$'s equals the number of eigenvalues of $T$ less than $\sigma$.) This may be written in terms of parallel prefix as follows:

$$\begin{bmatrix} w_i \\ w_{i-1} \end{bmatrix} = \begin{bmatrix} a_i - \sigma & -b_{i-1}^2 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} w_{i-1} \\ w_{i-2} \end{bmatrix} \equiv P_i \cdot \begin{bmatrix} w_{i-1} \\ w_{i-2} \end{bmatrix} = P_i \cdot P_{i-1} \cdots P_1 \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} \equiv Q_i \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{3}$$

So we need to compute $Q = \text{ParPrefix}(P, \cdot)$, again a 2 by 2 matrix multiply parallel prefix.

This recurrence suffers from the same sensitivity to over/underflow as the last one: $w_i$ is the determinant of the leading $i$ by $i$ submatrix of $T - \sigma I$, and so is subject to over/underflow even for matrices of modest norm and modest dimension. Furthermore, there is no known way to express its solution using scalar multiply and scalar addition parallel prefix as building blocks. In fact, we strongly suspect that no such expression exists, although we lack as yet a formal proof.

Furthermore, there is a theorem by H. T. Kung which says that if $x_i = f_i(x_{i-1})$ is a recurrence relation where $x_i$ is a scalar an $f_i$ is a rational function, then $x = [x_1, \ldots x_n]$ can be evaluated in faster than $\Omega(n)$ time if and only if it can be evaluated using a 2 by 2 matrix multiply parallel prefix. It turns out that the only parallelizable $f_i$ are of the form

$$f_i(x_{i-1}) = \frac{\alpha_i x_{i-1} + \beta_i}{\gamma_i x_{i-1} + \delta_i}$$

which can be parallelized by computing $x_i = u_i / v_i$ where

$$\begin{bmatrix} u_i \\ v_i \end{bmatrix} = \begin{bmatrix} \alpha_i & \beta_i \\ \gamma_i & \delta_i \end{bmatrix} \cdot \begin{bmatrix} u_{i-1} \\ v_{i-1} \end{bmatrix} \equiv S_i \cdot \begin{bmatrix} u_{i-1} \\ v_{i-1} \end{bmatrix} = S_i \cdots S_1 \cdot \begin{bmatrix} x_0 \\ 1 \end{bmatrix}$$

Thus, 2 by 2 matrix multiply parallel prefix is sufficient (and we believe necessary) to parallelize all parallelizable scalar recurrence relations. On the other hand, I do not believe it is adequate for 3 or more term recurrences (for the same reason I do not believe 1 term

# DRAFT
# A Specification for Floating Point Parallel Prefix

James Demmel
Mathematics Department and Computer Science Division
University of California
Berkeley, CA 94720

July 3, 1992

**Abstract**

Parallel prefix is a useful operation for various linear algebra operations, including solving bidiagonal systems of equations and finding the eigenvalues of a symmetric tridiagonal matrix. However, the simplest implementations of parallel prefix for the operations of scalar floating point add and scalar floating point multiply are inadequate to solve these important problems. This is because they are too susceptible to over/underflow, and because they apparently cannot solve the general two term recurrence needed to find eigenvalues. In this note we propose a specification for parallel prefix operations overcoming these drawbacks.

## 1 Motivation

Our notation for parallel prefix will be as follows. Let $r = [r_1, \ldots, r_n]$ and $s = [s_1, \ldots, s_n]$ denote $n$ vectors of data objects, which could be scalars or more complicated objects. Let $\otimes$ be an associative operator defined on these object. Then $s = \mathrm{ParPrefix}(r, \otimes)$ computes

$$s_i = r_1 \otimes \cdots \otimes r_i \quad .$$

The most basic numerical parallel prefix operations one could support are for scalar $x_i$ and $y_i$, and $\otimes$ being floating point addition or floating point multiplication. Of course these floating point operations are not truly associative, and the impact of this is a question of numerical analysis we will not pursue here.

Let $B$ be an $n$ by $n$ bidiagonal matrix with diagonal entries $s_1, \ldots, s_n$ and subdiagonal entries $e_1, \ldots, e_{n-1}$. To solve the linear system $Bx = y$, we need to solve the linear recurrence

$$x_i = \frac{-e_{i-1}}{s_i} x_{i-1} + \frac{y_i}{s_i} \equiv \eta_i x_{i-1} + \tau_i \tag{1}$$

This may be done in two mathematically equivalent ways using parallel prefix. ¿From (1) we have

$$\begin{bmatrix} x_i \\ 1 \end{bmatrix} = \begin{bmatrix} \eta_i & \tau_i \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_{i-1} \\ 1 \end{bmatrix} \equiv M_i \cdot \begin{bmatrix} x_{i-1} \\ 1 \end{bmatrix} = M_i \cdot M_{i-1} \cdots M_1 \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} \equiv N_i \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$