```
    elseif α_i · x_{i+1} + β_i · B^{m_{i+1}} overflows, then
        x_i = α_i · x_{i+1}/B + β_i · B^{m_{i+1}-1} (carefully!)
        m_i = m_{i+1} - 1
    elseif α_i · x_{i+1} + β_i · B^{m_{i+1}} underflows, then
        x_i = α_i · x_{i+1} · B + β_i · 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 i
B and add 1 to m_i, then subtract the largest m_i from all of them so the largest is no
zero, and then change x_i to $B^{-m_i} x_i$.

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

Alan Edelman points out that these can all be blocked in straigthfo

6. Compute $y_i = y_2 / D_i$ for all i in one parallel operation.

To protect against over/underflow, one can modify this scheme in o
The easiest way is to compute the logarithms of f_i (assume for the moment $f_i > 0$), then $\log d_i = \sum_{j=1}^{i} \log f_j$ using an add-scan, and $\log e_i = \log d_i - \log a_{i+1}$. 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 d_i and e_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 litt
of them too. But it will not overflow and almost certainly not under
(There are other choices of $\bar{d}$ and $\bar{e}$ that might be marginally safer against underflo
signs of f_i can then be accumulated and applied with a multiply scan operat
only ± 1's.

A better way to protect against overflow is to modify the multiply-
follows. Instead of computing $d_i = \prod_{j=0}^{i} f_j$, compute $\tilde{d}_i$ and integers 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
overflow threshold. Here is a code, which can obviously be "scanned" for $\tilde{d}_i$ 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
    $\tilde{d}_i = \tilde{d}_{i-1} \cdot f_i$
    $m_i = m_{i-1}$
  else if $\tilde{d}_{i-1} \cdot f_i$ would overflow then
    $\tilde{d}_i = \tilde{d}_{i-1} \cdot f_i / B$ (computed carefully!)
    $m_i = m_{i-1} + 1$
  else if $\tilde{d}_{i-1} \cdot f_i$ 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 m_i. When the m_i are made available, the largest one can be subtracted from all of them (so the largest i
$d_i = \tilde{d}_i B^{m_i}$ computed without fear of overflow.

The third approach is to run the recurrence $x_i = \alpha_i x_{i+1} + \beta_i$ sequentially, scaling if necessary as one goes. The code is similar in its u
last code above:

$\tilde{x}_n = \beta_n$; $m_n = 0$
for $i = n-1$ downto 1
  if $\alpha_i x_{i+1} + \beta_i \cdot B^{m_{i+1}}$ 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
spent testing and scaling. If the data is in IEEE single precision,
(normalized!) numbers can be computed without over/underflow in IEEE
and products of 128 in IEEE extended. If the data is IEEE double then p
be computed in IEEE extended. Thus, any scaling tests would only hav
8, 16 or 128 products.

4 Extensions

Of course, it would be nice if the user could supply his or her own da
tive operator, and have the system perform parallel prefix. Given the
handling as described above, it would be nice to have these basic floa
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 bidiagon
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 anal
all these (they are not all equivalent to evaluating the recurrence
nearly perfect backward error), but I don't see any obvious dangers
the scan operation to the extent possible, assuming only scans for fl
floating-point multiply (although the best solution would involve m
scan).

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

1. Compute $f_0 = 1$, $f_i = a_i / b_i$ for all i in one parallel step.
2. Compute $d_i = d_0 \prod_{j=0}^{i} f_i$ for all i using a multiply-scan operation.
3. Compute $e_0 = 1$, $e_i = 1/(b_i d_{i+1})$ 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
   by E^{-1} is).

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

Addition/Subtraction: compute $z \cdot r^k = (x \cdot r^m) \pm (y \cdot r^n)$. *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 ± y
    k = m
    if (overflow) then
        {z = (x/r) ± (y/r)}
        k = k + 1
    elseif (underflow) then
        {z = (x·r) ± (y·r)}
        k = k − 1
    endif
elseif (m = n − 1) then
    z = x ± (y/r)    /* no overflow possible if round to nearest */
    k = n
    if 0 < |z| < t then
        z = (x·r) ± y
        k = m
    endif
elseif (m = n + 1) then
    z = y ± (x/r)    /* no overflow possible if round to nearest */
    k = m
    if 0 < |z| < t then
        z = (y·r) ± 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 the protection against over/underflow I propose here, to see how much t 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 s
number. When $b = 16$ we must clearly take the larger of the two r values t
large safe exponent, and even then it is less than 50000. Choosing b
reasonable. Should we choose $r = 1$ or larger r? I believe the larger r
because of the larger parallel prefix operations it allows, and becaus
to implement, since the representation of a number is almost unique:
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 t
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
  $\{z = (x\sqrt{r}) \cdot (y\sqrt{r})\}$
  $k = k - 1$
endif

recurrences are enough to do 2 term recurrences). At this time, howe
any evidence that we frequently need to solve 3 or more term recurren

This leads us to propose the following building blocks for paralle

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

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

2 Specifications for Parallel Prefix

Basically, each floating point number will be replaced by a pair (f_i, e_i) where f_i is a floating point number and e_i an integer, with the pair representing $f_i r^{e_i}$, r some integer power of 2. The first problem is to choose r and the number of bits to store e_i so as to sto
for easy implementation and the ability to do very large parallel pre
fear of over/underflow. So given r, the number of bits b in which to s
integer, and the largest and smallest positive possible values of a fl
will ask how high a power of the largest and smallest floating point nu
without over/underflow in the form $f_i r^{e_i}$.

We will only consider IEEE single and double precision formats. The
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^{192} for IEEE single precision, and 2^{1536} for IEEE double precision. The source for these last two values is the wrapped expo
arithmetic: If overflow is trapped, the floating point unit is suppos
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 b
2^{192} or 2^{1536} times the true value.

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

The following table enumerates the approximate highest powers to wh
number f can be safely raised using the scaled format as a function of

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

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

Unfortunately, this is very nonrobust because f frequently over 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 part logarithms of the η_i and doing a $\text{ParPrefix}(\log, +)$ operation, but this is not a satisf 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 solv term recurrence

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

(The number of sign changes in the sequence of w_i is the number of eigenvalues o less than σ.) This may be written in terms of parallel prefix as follow

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

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

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

Furthermore, there is a theorem by H. T. Kung which says that if x recurrence relation $x_i = f_i(x_{i-1})$ where each f_i is a scalar 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 eval matrix multiply parallel prefix. It turns out that the only f_i helpful for paralleli

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

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 bel parallelize all parallelizable scalar recurrence relations. On the it is adequate for 3 or more term recurrences (for the same reason I d

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 comp
$\otimes$ be an associative operator defined on these objects. Then $s = \mathrm{ParPrefix}(r, \otimes)$ computes

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

The most basic numerical parallel prefix operations one could support ... r_i and y_i, and $\otimes$ being floating point addition or floating point multiplicati
floating point operations are not truly associative, and the impact o
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 off-diagonal entries $e_1, \ldots, e_{n-1}$. To solve the linear system $Bx = y$, we need to solve t
recurrence

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

This may be done in two mathematically equivalent ways using paralle
we have

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