Reliable Computing 1 (3) (1995), pp. 251–263

A general iterative sparse linear solver and its parallelization for interval Newton methods

Chenyi Hu, Anna Frolov*, R. Baker Kearfott, and Qing Yang

Interval Newton/Generalized Bisection methods reliably find all numerical solutions within a given domain. Both computational complexity analysis and numerical experiments have shown that solving the corresponding interval linear system generated by interval Newton's methods can be computationally expensive (especially when the nonlinear system is large).

In applications, many large-scale nonlinear systems of equations result in sparse interval Jacobian matrices. In this paper, we first propose a general indexed storage scheme to store sparse interval matrices We then present an iterative interval linear solver that utilizes the proposed index storage scheme It is expected that the newly proposed general interval iterative sparse linear solver will improve the overall performance for interval Newton/Generalized bisection methods when the Jacobian matrices are sparse. In Section 1, we briefly review interval Newton's methods. In Section 2, we review some currently used storage schemes for sparse systems. In Section 3, we introduce a new index scheme to store general sparse matrices. In Section 4, we present both sequential and parallel algorithms to evaluate a general sparse Jacobian matrix. In Section 5, we present both sequential and parallel algorithms to solve the corresponding interval linear system by the all-row preconditioned scheme. Conclusions and future work are discussed in Section 6.

Обобщенный итеративный линейный решатель для разреженных систем и его параллелизация для интервальных методов Ньютона

Ч. Ху, А. Фролов, Р. Б. Кирфотт, К. Янг

Интервальный метод Ньютона и обобщенный метод половинного деления гарантированно находят все численные решения в заданной области. Как анализ вычислительной сложности, так и численные эксперименты показали, что решение соответствующей интервальной линейной системы, полученной интервальными методами Ньютона, может потребовать значительного объема вычислений (особенно если нелинейная система велика по размерам)

На практике системы нелинейных уравнений большой размерности нередко сводятся к разреженным интервальным матрицам Якоби. В настоящей работе предлагается обобщенная индексированная схема памяти для хранения разреженных интервальных матриц, а затем вводится итеративный интервальный линейный решатель, использующий эту схему Ожидается, что предложенный обобщенный итеративный интервальный линейный решатель повысит общую производительность методов Ньютона и обобщенного метода половинного деления для разреженных матриц Якоби В разделе 1 кратко описаны интервальные методы Ньютона. В разделе 2 рассматриваются некоторые используемые в настоящее время схемы памяти для разреженных систем В

This research was partially supported by NSF Grant No. MIP–9208041.

*This co-author is a NASA funded undergraduate research assistant.

разделе 3 вводится новая индексированная схема памяти для хранения обобщенных разреженных матриц. В разделе 4 представлены последовательный и параллельный алгоритмы для решения соответствующей интервальной линейной системы по строчной преобусловленной схеме Выводы и планы на будущее обсуждаются в разделе 6.

1. Introduction

To find *all* numerical solutions for nonlinear systems of equations

$$F(X) = \Big(f_0(x_0, x_1, \ldots, x_{N-1}), f_1(x_0, x_1, \ldots, x_{N-1}), \ldots, f_{N-1}(x_0, x_1, \ldots, x_{N-1})\Big) = 0 \quad (1)$$

in a given box[1] $\mathbf{X} = (\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{N-1})$, where $x_i \in \mathbf{x}_i$ and $\mathbf{x}_i = \{x_i | \underline{x}_i \leq x_i \leq \overline{x}_i\}$ for $0 \leq i \leq N-1$, with mathematical and numerical certainty is a very important problem in scientific computation.

There are many methods for solving nonlinear systems of equations in the literature. In contrast to other methods, interval Newton's methods [5, 7, 8, 10] bound *all solutions* in a given domain with *mathematical certainty*, even in the presence of uncertainty in the data, roundoff error, and nonlinearities. Like the classical Newton's method, interval Newton's methods transform the nonlinear system (1) into a linear interval system:

$$\mathbf{F}'(\mathbf{X}^{(k)})(\tilde{\mathbf{X}}^{(k)} - X^{(k)}) = -\mathbf{F}(X^{(k)}). \quad (2)$$

In (2), $\mathbf{X}^{(k)}$ and $X^{(k)}$ are input values of $\mathbf{X}$ and X at the k-th iteration; $\tilde{\mathbf{X}}^{(k)}$ is the newly estimated bound of $\mathbf{X}$ at the k-th iteration; $\mathbf{F}'(\mathbf{X}^{(k)})$ can be an interval extension to the Jacobian matrix, or a slope matrix, over $\mathbf{X}^{(k)}$. It is known that all roots of the system (1) that belong to the interval box $\mathbf{X}^{(k)}$ also belong to the interval box $\tilde{\mathbf{X}}^{(k)}$ of the linear interval system (2). The solutions of the nonlinear system (1) can be found by iteratively solving the corresponding linear system (2), with the new iteration value $\mathbf{X}^{(k+1)}$ defined as

$$\mathbf{X}^{(k+1)} = \mathbf{X}^{(k)} \bigcap \tilde{\mathbf{X}}^{(k)}. \quad (3)$$

For the reader's convenience, we review the basic interval Newton/Generalized bisection method here.

Algorithm 1: (Interval Newton/Generalized Bisection)

(1.1) Input the current interval box $\mathbf{X}^{(k)}$ and a guess point $X^{(k)} \in \mathbf{X}^{(k)}$ (e.g., the midpoint of $\mathbf{X}^{(k)}$);

(1.2) Evaluate the value of $F(X)$ at $X = X^{(k)}$;

(1.3) Evaluate the interval Jacobian matrix $F'(\mathbf{X})$ on $\mathbf{X} = \mathbf{X}^{(k)}$;

(1.4) Find $\tilde{\mathbf{X}}^{(k)}$ by solving equation (2);

(1.5) For j=0 to $N-1$ do

Compute: $\mathbf{x}_j^{(k+1)} = \tilde{\mathbf{x}}_j \bigcap \mathbf{x}_j^{(k)}$;

If $\mathbf{x}_j^{(k+1)} = \emptyset$, then

there is no root in current box.

Go to step (1.8)

[1]Throughout the paper we will use boldface letters and capital letters to denote interval quantities and vectors, respectively. We use $\underline{x}$ and $\overline{x}$ to denote the lower bound and the upper bound for an interval variable $\mathbf{x}$, respectively

Else
Compute: $\omega_j^{(k+1)} = (\overline{x}_j^{(k+1)} - \underline{x}_j^{(k+1)})^2$;

(1.6) Compute: $\text{diam}(\mathbf{X}^{(k+1)}) = (\sum_{j=0}^{N-1} \omega_j)^{\frac{1}{2}}$;

(1.7) Check convergence:

(a) If $\text{diam}(\mathbf{X}^{(k+1)}) \leq \varepsilon$, and $|\mathbf{F}(X^{(k)})| \leq \xi$, where ε and ξ are given tolerances, then a root is found. Output the root in root list and goto step (1.8);

(b) If $\text{diam}(\mathbf{X}^{(k)}) - \text{diam}(\mathbf{X}^{(k+1)}) \leq \delta$, where δ is a given tolerance, then bisect the longest side of current box. Put half of it in the box stack and keep another half as the new box for the next loop of iteration. Goto step (1.1);

(c) If $\text{diam}(\mathbf{X}^{(k)}) - \text{diam}(\mathbf{X}^{(k+1)}) > \delta$, goto step (1.1);

(1.8) If the box stack is not empty, then pop a box from the stack, goto step (1.1). Otherwise, end the algorithm.

Step (1.3) of Algorithm 1 evaluates N^2 functions of N variables to determine the Jacobian matrix in general. Step (1.4) then solves an N-dimensional linear interval system. Both steps can be computationally expensive when N is large. In the past few years, a special techniques using so-called *preconditioners* have been developed to solve interval linear systems [9]: namely, instead of solving a linear equation $Ax = b$, we solve an equation $(YA)x = Yb$ fort an appropriately chosen Y (or several equations, with different Y). These *preconditioned linear solvers* require $\mathcal{O}(N^2)$ to $\mathcal{O}(N^3)$ computations; so, we need that many computational steps on each iteration of the Newton's method. Among these preconditioners, the *all-row preconditioned scheme* consists of using N preconditioners: namely, we find the interval estimates for all the variables $\tilde{\mathbf{x}}_0, \tilde{\mathbf{x}}_1, \ldots, \tilde{\mathbf{x}}_{N-1}$ from every linear equation of the system (2), and then take the intersection of the corresponding interval estimates $\tilde{\mathbf{x}}_i$ as a solution $\tilde{\mathbf{x}}_i$ (for $0 \leq i \leq N-1$). The all-row preconditioned scheme has the advantages of lower computational complexity and a fast convergence rate for many problems. In [6, 4], we described the results of parallelization of the all-row preconditioned scheme for large systems. In [6, 4], these methods were tried on *dense* matrices, i.e., matrices for which the majority of coefficients are non-zero.

In many applications, large-scale N-variable nonlinear systems of equations often generate *sparse* Jacobian matrices, i.e., matrices which contain only $\leq CN$ nonzero elements (for a small constant C). For example, in VLSI design and structural engineering, Jacobian matrices are block bordered and therefore, sparse; sparse matrices also appear in chemical process flowsheeting [14]. Moreover, usually, in real-life problems, Jacobian matrices *are* sparse. It is surely inefficient and wasteful to use general methods for dense systems to perform step (1.3) and (1.4) for a sparse Jacobian matrix.

In this paper, we report our work on developing an interval linear solver for large-scale general sparse systems with the all-row preconditioned scheme, and its parallelization for interval Newton's methods.

2. Storage schemes for general sparse matrices

Compactly storing a sparse matrix is the first step in developing a general sparse linear solver for interval Newton's methods. A significant amount of research has been done on sparse scalar matrices with special patterns (such as tridiagonal, band diagonal with bandwidth M,

band triangular, block diagonal, block triangular, cyclic banded, bordered block diagonal, and others). For such patterns, efficient storage and processing algorithms have been developed; these algorithms, however, crucially depend on the precise pattern of sparsity of the matrix and are, therefore, not applicable to generic sparse matrices that appear, e.g., in chemical process flowsheeting [14].

Since a general sparse matrix may not fit any of these special patterns, storage schemes for general sparse matrices have also been studied. Index storage schemes store non-zero matrix elements along with auxiliary information which can be used to determine where a non-zero element is located in the original matrix, crucial data in common matrix operations. Some of these index methods can require storage for as much as three to five times the number of nonzero matrix elements. Knuth describes a method in [11]; Duff describes several other methods [2, 3]; Dongarra et al. used a scheme in [1]; and Press et al. [13] claim they favor the scheme used by PCGPAK [12] because the row-indexed storage mode in [12] requires storage of little more than two times the number of nonzero matrix elements. Here, we review the two most popular general sparse storage schemes, the PCGPAK scheme and Duff's scheme.

The indexed storage scheme in PCGPAK: In PCGPAK, a sparse matrix is stored as two arrays: an array called $ija(k)$, which stores the indices (ija stands for *i-j-array*), and an array $sa(k)$ which stores selected elements of the matrix. The rules for defining the two arrays are described below.

1. For an $N \times N$ matrix, $ija(1) = N + 2$.
2. In the array ija, values $ija(2)$ through $ija(N+1)$ are calculated in the following way: $ija(i) = ija(i-1) + y_{i-1}$, where y_{i-1} is the number of nonzero nondiagonal elements in the $(i-1)-$st row.
3. The number of elements in each of these arrays is $ija(N-1) - 1$.
4. In the array sa, $sa(1)$ through $sa(N)$ are diagonal elements of the matrix to be stored.
5. The element $sa(N+1)$ is an extra space. A user may store any number he/she wants.
6. The elements $sa(N+2), sa(N+3), \ldots$ are all non-zero non-diagonal elements stored in the order of their rows (and in the order of columns if they are in the same row), and $ija(N+2), \ldots$ are the corresponding column numbers.

For example, let S be a 4×4 sample sparse matrix defined by

$$S = \begin{pmatrix} 3 & 0 & 4 & 0 \\ 0 & 0 & 5 & 0 \\ 1 & 0 & 0 & 7 \\ 0 & 2 & 0 & 1 \end{pmatrix}. \tag{4}$$

Using the PCGPAK scheme, the matrix S can be stored as:

k	1	2	3	4	5	6	7	8	9	10
$ija(k)$	6	7	8	10	11	3	3	1	4	2
$sa(k)$	3	0	0	1	x	4	5	1	7	2

Duff's index storage scheme: In [2, 3], Duff defines an indexed scheme to store sparse matrices. In Duff's scheme, three arrays are used to store a matrix. The three arrays are called COLPTR (column pointer), ROWIND (row index), and VALUES (values of non-zero elements).

1. COLPTR(k) are calculated in the following way: COLPTR(1) = 1 and

$$\text{COLPTR}(i) = \text{COLPTR}(i-1) + y_{i-1}$$

(where y_{i-1} is the number of nonzero elements in the column $(i-1)$).

2. The array VALUES stores all nonzero elements column-wise, and the array ROWIND contains corresponding row numbers.

By using Duff's scheme, the sample matrix S can be stored as

k	1	2	3	4	5	6	7
COLPTR(k)	1	3	4	6	8		
ROWIND(k)	1	3	4	1	2	3	4
VALUES(k)	3	1	2	4	5	7	1

From the above, we can see that the PCGPAK scheme stores all diagonal elements even when most of them are zero. Three arrays are used in Duff's storage scheme. Indirect memory references may significantly reduce the overall speed of computations. In [13], Press et al. state that there is no *standard* scheme in general use. We believe that further studies on indexed storage schemes are needed to handle generic sparse matrices.

3. A new storage scheme for general sparse system

In this section, we propose standards to measure an indexed storage scheme for general sparse matrices. An ideal indexed storage scheme for large-scale general sparse matrices should have the following properties:

- It uses the least amount of memory space.
- The original position of an element in a matrix can be recovered easily.
- Fundamental matrix algebra can be done with minimal effort.
- It can be efficiently implemented on high performance computers.
- It minimizes memory accessing latency[2].
- It is easy to understand.

To address the properties listed the above, we propose the following indexed storage scheme:

Indexed storage scheme 1: Let $\mathbf{A} = \{a_{i,j}\}$ be an $N \times N$ sparse interval matrix, where $0 \le i, j \le N-1$. For a nonzero element of $\mathbf{A}$, say $a_{i,j}$, its row-after-row index is defined as

[2]We will address this in another paper

$iN + j$; its column-after-column storage index is defined as $jN + i$. The indexes will be stored in increasing order.

According to the above definition, we can store the sample matrix S row-after-row by checking the list $s_{0,0}, s_{0,1}, s_{0,2}, s_{0,3}, s_{1,0}, s_{1,1}, \ldots, s_{3,3}$ and assigning the index $iN + j$ for each non-zero element $s_{i,j}$, $0 \leq i, j \leq N - 1$ (In S, $N = 4$).

$k_{i,j}$	0	2	6	8	11	13	15
$\mathbf{a}_{i,j}$	3	4	5	1	7	2	1

Similarly, we may store the sample matrix S in the column-after-column manner as

$k_{m,n}$	0	2	7	8	9	14	15
$\mathbf{a}_{m,n}$	3	1	2	4	5	7	1

Let us list some advantages of storage scheme 1:

1. The indexed storage scheme stores only non-zero elements of a general sparse matrix and associates only one index to each non-zero element. We believe that the storage scheme uses minimum memory space to index store a *general* sparse matrix, since any storage schemes they must store all nonzero elements of the sparse matrix, and must have at least one index to indicate the original position of each non-zero element.

2. It is easy to recover the original location of a nonzero element from its index. Suppose the row-after-row index for a nonzero element is k. Then the element is in the (k div N)-th row, and the (k mod N)-th column.

3. The transpose of the matrix is also very easy to determine. For example, the row-after-row index for a nonzero element of a matrix $\mathbf{A}$ is k, then its logical position in $\mathbf{A}^T$ is in the (k mod N)-th row, and in the (k div N)-th column.

4. It is easy to perform addition and subtraction of sparse matrices. Those elements that have the same logical position have the same index. To perform addition or subtraction, one needs only add or subtract the corresponding elements and place the result in the corresponding position of the result matrix. Those elements that appear in only one of the matrices need only be inserted, along with their indices, into the resulting matrix.

5. It can be efficiently implemented on high performance parallel computers with high scalability. Let p be the number of processors on a parallel machine. We may distribute sparse matrix computations row-wise according to the values of (index div N) mod p or column-wise according to the values of (index mod N) mod p. Applications of this property can be found in Sections 4 and 5 of this paper.

6. Other advantages of the storage scheme, such as improving the performance of a memory hierarchy, will appear in another paper of ours.

Storage scheme 1 uses two divisions to recover the original position of a non-zero element. To make scheme 1 more efficient on a sequential computer, we modify it as follows:

Indexed storage scheme 2: Let $\mathbf{A} = \{\mathbf{a}_{i,j}\}$ be an $N \times N$ sparse matrix, where $0 \leq i, j \leq N-1$. The row-after-row index $k_{i,j}$ of a nonzero element of $\mathbf{A}$, $\mathbf{a}_{i,j}$ is defined as

$$k_{i,j} = \begin{cases} iN + j, & \text{if } \mathbf{a}_{i,j} \text{ is the first nonzero element in the } i\text{-th row;} \\ j, & \text{if } \mathbf{a}_{i,j} \text{ is not the first nonzero element in the } i\text{-th row.} \end{cases}$$

The indexes will be put in increasing order row-wise.

To store a nonzero element $\mathbf{a}_{i,j}$ of A column-after-column, we assign its index $k_{i,j}$ as

$$k_{i,j} = \begin{cases} jN + i, & \text{if } \mathbf{a}_{i,j} \text{ is the first nonzero element in the } j\text{-th column;} \\ i, & \text{if } \mathbf{a}_{i,j} \text{ is not the first nonzero element in the } j\text{-th column.} \end{cases}$$

The indexes will be put in increasing order column-wise.

With the row-after-row index storage scheme 2, the sample matrix S introduced above an be stored as:

$k_{i,j}$	0	2	6	8	3	13	3
$\mathbf{a}_{i,j}$	3	4	5	1	7	2	1

The sample matrix S can be also stored column-after-column in scheme 2 as:

$k_{i,j}$	0	2	7	8	1	14	3
$\mathbf{a}_{i,j}$	3	1	2	4	5	7	1

We list two additional advantages of the scheme 2. First, to sequentially recover the original positions of nonzero elements in a sparse matrix stored by scheme 2, we only need to perform divisions for the index of the first nonzero element in each row (or column). Second, to interchange two rows (columns), we only need to change the indexes for the first non-zero elements in those two rows (columns).

We recommend scheme 2 for sequential processing and scheme 1 for parallel processing. To support this recommendation, we have compared the performance of scheme 2 (row-after-row) with that of both the PCGPAK scheme and Duff's scheme.

In the remainder of this section, we report our numerical experiments comparing the performance of these storage schemes. The data set was picked from Harwell-Boeing Sparse Matrix Collection. In the collection, the CHEMWEST set contains general sparse matrices from modeling of chemical engineering plants at University of Pittsburgh; the FACSIMILE set is "representative of the type of matrices which occur in spatially homogeneous problems from straight chemical kinetics calculations and mixed kinetics diffusion problems." Our FORTRAN−77 programs were run on a VAX 4000−300 computer with VAX/VMS version V5.5−2 operating system. The compiler is DEC FORTRAN for OpenVMS VAX Systems.

The Table 1 below reports memory space used to index store these matrices with different schemes.

The basic operations of most iterative linear solvers (such as the Jacobi, Gauss-Seidel, and successive overrelaxation methods, conjugate gradient, generalized minimal residual, biconjugate gradient, and some other methods) are matrix-vector multiplications. In this multiplication, a vector is multiplied either by a coefficient matrix or by its transpose. To test the efficiency of our method on these multiplications, we measured the time required to multiply an index stored matrix by a randomly generated vector (see Table 2).

The Table 3 reports time required to multiply the *transpose* of an indexed stored matrix by a vector.

N	Nonzero elements	PCGPAK Method	Duff's Method	Scheme 2
67	294	722	656	588
132	414	1084	961	828
156	371	1058	899	742
167	507	1324	1182	1014
183	1069	2142	2322	2138
183	1069	2142	2322	2138
183	1069	2142	2322	2138
183	1069	2142	2322	2138
381	2157	5078	4696	4314
479	1910	4766	4300	3820
497	1727	4440	3952	3454
655	2854	7012	6364	5708
680	2646	5296	5973	5292
680	2646	5296	5973	5292
680	2646	5296	5973	5292
760	5976	11956	12713	11952
760	5976	11956	12713	11952
760	5976	11956	12713	11952

Table 1. Memory space used

N	Nonzero elements	PCGPAK Method	Duff's Method	Scheme 2
67	294	0.0E+00	3.7E−02	0.0E+00
132	414	0.0E+00	4.5E−02	1.0E−03
156	371	2.0E−03	4.7E−02	1.0E−03
167	507	1.0E−03	4.6E−02	0.0E+00
183	1069	3.0E−03	5.1E−02	3.0E−03
183	1069	4.0E−03	5.6E−02	1.0E−03
183	1069	4.0E−03	5.0E−02	1.0E−03
183	1069	2.0E−03	5.5E−02	2.0E−03
381	2157	5.0E−03	9.2E−02	7.0E−03
479	1910	6.0E−03	1.1E−01	6.0E−03
497	1727	6.0E−03	1.2E−01	7.0E−03
655	2854	8.0E−03	1.4E−01	1.0E−02
680	2646	1.0E−02	1.4E−01	9.0E−03
680	2646	2.0E−03	1.5E−01	1.0E−02
680	2646	6.0E−03	1.4E−01	9.0E−03
760	5976	1.3E−02	1.6E−01	2.0E−02
760	5976	1.2E−02	1.6E−01	2.0E−02
760	5976	1.3E−02	1.7E−01	1.9E−02

Table 2. Time required for matrix-vector multiplication

N	Nonzero elements	PCGPAK Method	Duff's Method	Scheme 2
67	294	2.00E−03	2.00E−03	0.00E+00
132	414	3.00E−03	1.00E−03	0.00E+00
156	371	0.00E+00	0.00E+00	0.00E+00
167	507	1.00E−03	2.00E−03	0.00E+00
183	1069	1.00E−03	4.00E−03	2.00E−03
183	1069	1.00E−03	4.00E−03	0.00E+00
183	1069	1.00E−03	5.00E−03	3.00E−03
183	1069	3.00E−03	2.00E−03	2.00E−03
381	2157	7.00E−03	9.00E−03	6.00E−03
479	1910	5.00E−03	4.00E−03	4.00E−03
497	1727	6.00E−03	4.00E−03	6.00E−03
655	2854	1.00E−02	9.00E−03	1.00E−02
680	2646	7.00E−03	9.00E−03	8.00E−03
680	2646	8.00E−03	8.00E−03	9.00E−03
680	2646	5.00E−03	5.00E−03	7.00E−03
760	5976	1.56E−02	1.40E−02	1.80E−02
760	5976	1.20E−02	1.60E−02	1.70E−02
760	5976	1.50E−02	1.70E−02	1.80E−02

Table 3. Time required for matrix-transpose times a random vector

4. Evaluating general sparse Jacobian matrices

With the storage schemes defined in the previous section, we propose efficient algorithms to evaluate the interval Jacobian matrix $\mathbf{F}'(\mathbf{X}^{(k)})$ in step (1.3) of Algorithm 1. We assume that the analytical form of the Jacobian $\mathbf{F}'$ is known, and the Jacobian matrix is general sparse. Algorithm 2 below evaluates a sparse interval Jacobian matrix $\mathbf{F}'(\mathbf{X}^{(k)}) = \mathbf{f}'_{i,j}(\mathbf{X}^{(k)})$, and stores it row-after-row with scheme 2.

Algorithm 2: (sequential algorithm)

(2.1) $m = 0$

(2.2) For $i = 0$ to $n - 1$ do $\text{First}_i = 1$;

(2.3) For $i = 0$ to $n - 1$ do
  For $j = 0$ to $n - 1$ do
    If $\mathbf{f}'_{i,j} \not\equiv 0$ then
      If $\text{First}_i = 1$ then
        $\text{Index}_m = i * n + j$
        $\mathbf{a}_m = \mathbf{f}'_{i,j}(\mathbf{X}^{(k)})$
        $m = m + 1$
        $\text{First}_i = 0$
      else
        $\text{Index}_m = j$
        $\mathbf{a}_m = \mathbf{f}'_{i,j}(\mathbf{X}^{(k)})$
        $m = m + 1$
      endif

The major computation of the above sequential algorithm is to evaluate $\mathbf{f}'_{i,j}(\mathbf{X}^{(k)})$ for different i and j. Since there is absolutely no data dependency in evaluating $\mathbf{f}'_{i,j}(\mathbf{X}^{(k)})$ for different i and j, we may evaluate the Jacobian matrix for large-scale problems in parallel. In the following parallel program, p is the number of processors available, and each available processor has its own processor-id (which is called my-id). The algorithm stores the Jacobian matrix row-after-row with scheme 1.

Algorithm 3: (parallel algorithm)

(3.1) Do all $m = 0$

(3.2) For $i = 0$ to $n - 1$ do
  For $j = 0$ to $n - 1$ do
    If $\mathbf{f}'_{i,j} \not\equiv 0$ then
      $\text{Index}_m = i * n + j$
      $m = m + 1$
    endif

(3.3) For $l = 0$ to $m - 1$ do
  If my-id $= l \bmod p$, then
    $i = \text{Index}_l \text{ div } n$
    $j = \text{Index}_l \bmod n$
    $\mathbf{a}_l = \mathbf{f}'_{i,j}(\mathbf{X}^{(k)})$
  endif

(3.4) All-to-all broadcast $\mathbf{a}_l$

Remark: Step (3.4) provides the interval Jacobian matrix to all p processors. It is a very expensive communication step. However, because the parallel iterative linear solver (Algorithm 5 from the next section) does not require the entire Jacobian matrix on each processor, we can communicate only part of the matrix.

5. Iterative general sparse linear solver

The interval Jacobi method, interval Gauss-Seidel method, and some other iterative interval linear solvers have been used to perform step (1.4) of Algorithm 1 for bounding equation (2). Preconditioned schemes [9] have also been proposed to improve the efficiency for dense interval systems. Among them the all-row preconditioned interval linear solver [9, 6, 4] bounds every variable $\mathbf{x}_i$ from each of the n equations of (2), then takes the intersection for the n different (some of them may be same) interval values of $\mathbf{x}_i$, thus forming $\tilde{\mathbf{x}}_i$. It has lower computational complexity and a faster convergence rate for many interval linear systems. With the Jacobian matrix index stored by Algorithm 2, we propose the all-row preconditioned interval iterative linear solver for sparse systems as[3]:

Algorithm 4: (sequential algorithm)

(4.1) For $i = 0$ to $n - 1$ do
  $\mathbf{b}_i = \mathbf{f}_i(\mathbf{X}_k)$

[3]The operation $\ominus$ is used in Algorithms 4 and 5. It is defined as $\mathbf{a} \ominus \mathbf{b} = [\underline{a}, \overline{a}] \ominus [\underline{b}, \overline{b}] = [\underline{a} - \underline{b}, \overline{a} - \overline{b}]$.

(4.2) $i = 0$

(4.3) For $l = 0$ to $m - 1$ do
  If $\text{Index}_l < n$, then
    $j = \text{Index}_l$
    $\mathbf{b}_i = \mathbf{b}_i + \mathbf{a}_l\mathbf{x}_j$
  else
    $i = \text{Index}_l$ div n
    $j = \text{Index}_l$ mod n
    $\mathbf{b}_i = \mathbf{b}_i + \mathbf{a}_l\mathbf{x}_j$
  endif

(4.4) $i = 0$

(4.5) For $l = 0$ to $m - 1$ do
  If $\text{Index}_l < n$, then
    $j = \text{Index}_l$
    $\tilde{\mathbf{x}}_{j_i} = x_j + \frac{\mathbf{b}_i \ominus \mathbf{a}_l\mathbf{x}_j}{\mathbf{a}_l}$
  else
    $i = \text{Index}_l$ div n
    $j = \text{Index}_l$ mod n
    $\tilde{\mathbf{x}}_{j_i} = x_j + \frac{\mathbf{b}_i \ominus \mathbf{a}_l\mathbf{x}_j}{\mathbf{a}_l}$
  endif

(4.6) For $j = 0$ to $n - 1$ do
  $\tilde{\mathbf{x}}_j = (\bigcap_{i=0}^{i=n-1} \tilde{\mathbf{x}}_{j_i}) \cap \mathbf{x}_j$

Associated with Algorithm 3, the following algorithm bounds $\tilde{\mathbf{X}}^{(k)}$ in (2) with parallel computations.

Algorithm 5: (parallel algorithm)

(5.1) For $i = 0$ to $n - 1$ do
  $\mathbf{y}_i = \mathbf{x}_i$

(5.2) For $l = 0$ to $m - 1$ do
  $i = \text{Index}_l$ div n
  If my-id $= i$ mod p, then
    $\mathbf{b}_i = \mathbf{f}_i(\mathbf{X}_k)$
    $j = \text{Index}_l$ mod n
    $\mathbf{b}_i = \mathbf{b}_i + \mathbf{a}_l\mathbf{x}_j$
  endif

(5.3) For $l = 0$ to $m - 1$ do
  $i = \text{Index}_l$ div n
  If my-id $= i$ mod p, then
    $j = \text{Index}_l$ mod n
    $\mathbf{y}_j = \mathbf{y}_j \cap (x_j + \frac{\mathbf{b}_i \ominus \mathbf{a}_l\mathbf{x}_j}{\mathbf{a}_l})$
  endif

(5.4) All-to-all broadcast $\mathbf{y}_j$ for $j = 0$ to $n - 1$

(5.5) Replace $\mathbf{y}_j$ with $\mathbf{y}_j$ (local) $\cap$ $\mathbf{y}_j$ (incoming)

(5.6) $\mathbf{x}_j = \mathbf{y}_j$ for $j = 0$ to $n - 1$

After the completion of Algorithm 5, there will be a copy of $\mathbf{X}^{(k+1)}$ in the local memory of every processor. As written, Algorithms 4 and 5 each complete only one iteration for solving the linear interval system (2). One may want to perform more such iterations before reevaluating the interval Jacobian matrix by Algorithm 2 or 3 over $\mathbf{X}^{(k+1)}$.

6. Conclusions and future work

The computationally most expensive procedures when using interval Newton's methods to solve nonlinear systems of equations are evaluations of its Jacobian matrix and solving the corresponding interval linear systems. The matrix storage schemes, Jacobian evaluation and linear solver algorithms in this paper are specifically designed for general sparse systems. The comparison with previous algorithms (based on dense systems) make us believe that the methods proposed in this paper will save both memory space and computation time.

With the parallelized algorithms, one may be able to effectively solve real-life very large-scale systems (that in applications are usually sparse) in parallel.

We are currently implementing parallel algorithms proposed in this paper on real parallel computers in order to compare the performance of these algorithms with other schemes. We will continue this research as follows:

- Study how to balance work load for Algorithms 3 and 5 to achieve high efficiency.
- Develop preconditioned linear solvers, other than the all-row preconditioned scheme, with the general sparse storage scheme.

Acknowledgments

We wish to acknowledge the referees for their careful reading and useful suggestions.

References

[1] Dongarra, J. et al. *Solving linear systems on vector and shared memory computers*. SIAM, 1991.

[2] Duff, I. *Direct methods for sparse matrices*. Oxford University Press, 1986.

[3] Duff, I. *Sparse matrix test problems*. ACM Trans. Math. Software **15** (1) (1989), pp. 1–14.

[4] Gan, Q., Yang, Q., and Hu, C. *Parallel all-row preconditioned interval linear solver for nonlinear equations on multiprocessor*. Parallel Computing **20** (9) (1994), pp. 1249–1268.

[5] Hansen, E. R. and Sengupta, S. *Bounding solutions of systems of equations using interval arithmetic.* BIT **21** (1981), pp. 203–211.

[6] Hu, C., Bayoumi, M., Kearfott, R. B., and Yang, Q. *A parallelized algorithm for all-row preconditioned interval Newton/generalized bisection method.* In: "Proc. SIAM 5th Conf. on Paral. Proc. for Sci. Comp.", SIAM, 1991, pp. 205–209.

[7] Kearfott, R. B. *Abstract generalized bisection and a cost bound.* Math. Comp. **49** (179) (1987), pp. 187–202.

[8] Kearfott, R. B. *Some tests of generalized bisection.* ACM Trans. Math. Software **13** (3) (1987), pp. 197–220.

[9] Kearfott, R. B., Hu, C., and Novoa, M. *A review of preconditioners for the interval Gauss-Seidel method.* Interval Computations 1 (1991), pp. 59–85.

[10] Kearfott, R. B. and Novoa, M. *A program for generalized bisection.* ACM Trans. Math. Software **16** (2) (1990), pp. 152–157.

[11] Knuth, D. *The art of computer programming, Vol. 1, Fundamental algorithms.* Addison-Wesley, 1968.

[12] *PCGPAK user's guide.* Scientific Computing Associates, New Haven.

[13] Press, W. et al. *Numerical recipes.* Cambridge, 1992.

[14] Schnepper, C. and Stadther, M. *Application of a parallel interval Newton/generalized bisection algorithm to equation-based chemical process flowsheeting.* Interval Computations 4 (1993), pp. 40–64.

Received: March 1, 1994
Revised version: September 9, 1994

C. Hu
Department of Computer and Mathematical Sciences
University of Houston-Downtown
Houston, Texas 77002, USA

A. Frolov
Department of Computer and Mathematical Sciences
University of Houston-Downtown
Houston, Texas 77002, USA

R. B. Kearfott
Department of Mathematics
University of Southwestern Louisiana
U.S.L. Box 4–1010
Lafayette, LA 70504–1010, USA
E-mail: rbk@usl.edu

Q. Yang
Department of Electrical and Computer Engineering
University of Rhode Island
Kingston, Rhode Island 02881, USA