Journal of Computer Science and Information Technology
December 2014, Vol. 2, No. 3 & 4, pp. 01-13
ISSN: 2334-2366 (Print), 2334-2374 (Online)

Published by American Research Institute for Policy Development
DOI: 10.15640/jcsit.v2n3-4a1
URL: http://dx.doi.org/10.15640/jcsit.v2n3-4a1

Binary Tree Banded Linear Equation System Solver

Read, William[1]

Abstract

Although linear equation system solvers have been extensively treated, the state of the art for large equation systems -the blocking algorithm- does not have to be the end of the line. The author proposes the use of binary trees for very large systems, obtaining a far superior, timeeffective result. A computer test-program shows the advantages of the method. The paper concludes that the proposed tool could open new horizons to research in the fields of artificial intelligence and neural analysis.

Keywords: Large linear equations systems, sparse populated matrices, double binary tree, equations solver

Introduction

A system of n simultaneous linear equations in n unknowns has a unique solution if the n equations are linearly independent. In that case the determinant of the matrix coefficient is nonzero. When the system is small, the solution may be obtained by simple manual methods. Larger systems make use of well developed, known algorithms and methods, like the direct Gauss algorithm and derivatives, or of iterative methods.

In engineering and other scientific applications, large systems of symmetrical linear equations frequently appear, which are also positive definite with prevailing terms on the main diagonal. These systems are susceptible of being solved with simple and efficient computational algorithms, especially when they are symmetrical about the main diagonal.

[1] CE L.f.M. THH (Germany), Researching Professor, School of Civil Engineering, Universidad Nacional Pedro Henríquez Ureña (Santo Domingo). Email: wread@unphu.edu.do

Furthermore, there are banded symmetric systems, for which techniques have been developed that make this process still more efficient. The idea is to use at maximum the storage capacity for equations coefficients. As the solving process for symmetric banded systems is based on reducing it first to a triangular matrix, and in that reduction each row of the matrix changes the rows below it, the band character means that only "iband" equations will be modified below each row. This leads to solution methods for solving blocks of still larger systems, swapping to the central memory each time, the block to be solved from the peripheral memory (hard disk, etc.). In all these cases, the storage of the matrix is done with matrix arrays, generally with two-indices.

In many cases, the symmetric band matrices can be categorized as sparse populated, that is, contain more zero entries than nonzero terms. The global stiffness matrix of a structural system belongs to that category. With this premise in mind the author developed in 1992 a modified algorithm to exclude the storage of the zero-terms of the array, while avoiding unnecessary operations for this concept.

The Double Binary Tree

Thus the idea was born of applying the concept of the binary tree instead of the matrix. Using a binary tree of tree roots, an equivalent of a two-index matrix is achieved. The search operation is now super-efficient: to find the row is to find the root of the tree columns, and thus the zero terms are nil and do not call for any operations. Of course we had to program a series of additional support functions, but they were simple and powerful algorithms based on recursion.

To test the method, we initially programmed the construction of a system of equations of all 10.0's in the main diagonal, and all 1.0's on both sides of the main diagonal; right hand terms are the sum of the coefficients of each row. The solution should be then all 1.0's in the output vector. Entry parameters are: number of equations n and bandwidth iband. Applying the method to problems that handle sparse populated matrices gives outstanding efficiencies, then there is no time lost handling zero-operations. That is the case in structural engineering, where large assemblies of finite elements leads to large system of that type. The original test suite can be downloaded from

http://www.mediafire.com/download/26lz5cqba44x2bd/BlessTest.rar. Using this tool, one can verify the theoretical relationships between time consumption and number of equations (linear) and bandwidth (quadratic) of the system. These relationships are stated assuming that the *search time* is negligible; this is true for "normal" sized equations system, and that depends upon what we consider "normal" and what we call "large". These criteria have been bound to the size of the resulting matrix compared with the available storage capacity of existing electronic computing devices.

Matrices are "scanned" row- or column-wise to search for the terms Aij. So the average search time is proportional to the number of terms of the matrix. Opposed to that, binary trees search by a recursive bisection exponential function that is extremely efficient. Furthermore, the binary tree can consider non existing terms *without assigning storage* to it! So we can now think of very large matrices considering an increase of order of magnitude never thought before (millions of equations) where the search time starts to be a very important factor. For those arrays, the *double binary tree* seem to be the best solution, then the search time can still be neglected. Research conducted by the author in the early 90's led to that conclusion, but with little incentive to apply this tool in *current* structural analysis, where the established methods also do the job flawlessly.

New Horizons

But giving it a second thought, with the advances made in *artifficial intelligence* and the studies of *neural networks* in the last decade, researchers have recognized the analogy of neural networks with classical mathematical matrices, where the terms of the main diagonal act as neurons and the terms outside the main diagonal are links beween two neurons that represent *activation functions.*

These links are sparse and again non existing links mean a nil in a binary tree. Neuronal models can grow to millions and activation functions to thousands and still that could be represented as double binary tree matrices easily. The neural functions are of matrix-character with input layers followed by a "black box" of matricial operations that lead to the output layer. With computers getting more and more powerful, the use of double binary trees could give a handshake to the research of neural behavior.

C++ Source Code

The original code for the above mentioned test-suite BlessTest was written in Pascal (Delphi); the equivalent command line interface code written in C++ follows in the annex. It was compiled and tested with a Gnu gcc compiler, version 4.x in a Linux PC. The test program considers a full populated band matrix with 10's and 1's, then this was the worst case for timing measurements; nevertheless, the code has been used for years in a proprietary finite elements application by the author, that uses sparse populated global matrices, with success.

References

Wilson, Edward L.: Three Dimensional Static and Dynamic Analysis of Structures, Berkeley, California, USA, 2000.

Jensen, Kathleen and Wirth, Niklaus: Pascal User Manual and Report, Springer Verlag, 1978.

Manguoglu, Murat et al: Weighted Matrix Ordering and Paralell Banded Preconditioners for Iterative Linear System Solvers, SIAM J. Sci. Comput. Vol. 32, No.3, pp 1201-1216.

Rojas, R: Neural Networks, Springer Verlag, Berlin 1966.

Annex

bless.h

```
x-------------------------------------------------------------------------------x
 // Header file for high efficiency banded linear equation system solver (BLESS)
 // Author: William Read, Universidad Nacional Pedro Henríquez Ureña, Santo
Domingo.
 // May be used, copied, reproduced or modified, giving credit to the author.
 //
#ifndef BLESS_H
#define BLESS_H
#endif

enum task { trian, reslv
};

// Declare a row binary tree structure

struct Row_tr {
       float e;
       int j;
       struct Row_tr *left, *right;
};
// Another binary tree holds the roots of every row

struct Header {
       Row_tr *diag;
       int i, n;  // row is I, row elements are n
       struct Header *top, *bottom;
};

// Les is the linear equation system class

class Les  {
       int neq, iband;
       bool banded, reduced;
       struct Header *heads;
public:
       Les(int ne, int ib);                    // constructor functions
       ~Les();                                 // destructor functions
       Header *LocHeader(int nd1);             // locate header to row nd1
       Row_tr *Loc(int nd1, int nd);           // locate element nd2 of row nd1
       float Get(int nd1, int nd2);            // get element at nd1, nd2
       void Update(int nd1, int nd2, float newx);    // change value at nd1, nd2
       void AddElem(float x, int nn1, int nn2);      // create value at nn1, nn2
       void DelRow(Row_tr *p);                 // delete row tree pointed to by p
       void DelMatrix();                       // delete system from memory
       Row_tr *RowsTree(int nel);              // build a balanced row tree
                                               // initialized to 0.0 and 0
       void Balance(int nd1);                  // balance the nd1-th row tree
```

```
        int GetLevel(int I);                    // get the level of the I-th tree
        long Count();                           // get the total number of elements
        void Bansol(task duty);                 // solve the symmetric linear
                                                //  equations system
 protected:
        Header *HeadersTree(int ne);            // builds a balanced tree of ne
                                                   header pointers
        void InitTree(Header *h);               // initializes the row number on
                                                   each header structure
        void newelem(float xx, int ii, Row_tr **p);   // add elem to col i of row
                                                         // tree *p
        void DelHeads(Header (**q));            // deletes the roots of rows
        Row_tr *GetNext(int nd1);               // get pointer to next elem of
                                                // nd1-th row tree
        void CopyTree(Row_tr *p, int nd1);      // copy tree to a balanced new tree
        void level(Row_tr *a, int l);           // get the level of the a-tree
        void countelem(Row_tr *p, long *n);     // count of row elements pointed to
                                                //  by p
        void countrow(Header *q, long *n);      // count of system rows
};

x-------------------------------------------------------------------------------x

bless.cpp
x-------------------------------------------------------------------------------x
 // High efficiency linear equations system solver module
 // Author: William Read, Universidad Nacional Pedro Henríquez Ureña, Santo
Domingo.
 // May be applied, copied or reproduced, giving credit to the author.
 //
#include <iostream>
#include <iomanip>
#include <stdio.h>
#include <bless.h>

using namespace std;

Header *CurrHead;                               // Current head
Row_tr *q;                                      // q and
int ni;                                             // ni are global for bless

// Builds a balanced tree of headers; returns pointer to the root

Header *Les::HeadersTree(int ne)
{
      if (ne == 0) return NULL;
      else
      {     int ni = ne / 2;
            int nd = ne - ni - 1;
            Header *pij = new Header;
            pij->diag = NULL;
            pij->i = 0;
            pij->n = 0;
            pij->top = HeadersTree(ni);
            pij->bottom = HeadersTree(nd);
            return pij;
      }
}
```

```
// initializes row numbers of headers structures

void Les::InitTree(Header *h)
{
	if ( h != NULL) {
		InitTree(h->top);
		h->i = ni; ++ni;
		InitTree(h->bottom);
	}
}

// the constructor function

Les::Les(int ne, int ib)
{
	neq = ne;
	iband = ib;
	banded = (neq > iband);
	heads = HeadersTree(neq);
	ni = 1; InitTree(heads);
}

// the destructor function

Les::~Les()
{
	DelMatrix();
}

// the locate header function

Header *Les::LocHeader(int nd1)
{
	Header *p = heads;
	bool found = false;
	while ((p != NULL) && (found == false)) {
		if (p->i == nd1) found = true;
		else
		if (p->i > nd1) p = p->top;
		else
		if (p->i < nd1) p = p->bottom;
	};
	return p;
}

// the locate element function

Row_tr *Les::Loc(int nd1, int nd2)
{
	Row_tr *p = LocHeader(nd1)->diag;
	bool found = false;
	while ((p != NULL) && (found == false)) {
		if (p->j == nd2) found = true;
		else
		if (p->j > nd2) p = p->left;
		else
		if (p->j < nd2) p = p->right;
```

```
        };
        return p;
}

// get element function

float Les::Get(int nd1, int nd2)
{
        Row_tr *p = Loc(nd1, nd2);
        if (p != NULL) return p->e;
        else
        cout << "\nThe element "<< nd1 <<", "<< nd2 <<" does not exist\n";
}

// update element

void Les::Update(int nd1, int nd2, float newxx)
{
        Row_tr *p = Loc(nd1, nd2);
        if (p != NULL) {
        p->e = newxx; return;
        }
        else
        cout << "\nThe element "<< nd1 <<", "<< nd2 <<" does not exist\n";
}

// add element at nn2 to row pointd to by p

void Les::newelem( float xx, int jj, Row_tr **p)
{
        if ((*p) == NULL) {
                (*p) = new Row_tr;
                (*p)->e = xx;
                (*p)->j = jj;
                (*p)->left = NULL;
                (*p)->right = NULL;
        }
        else
        if ((*p)->j > jj) newelem(xx, jj, &(*p)->left);
        else
        if ((*p)->j < jj) newelem(xx, jj, &(*p)->right);
}

// add element to matrix at nn1 nn2

void Les::AddElem(float x, int nn1, int nn2)
{
        CurrHead = LocHeader(nn1);
        newelem(x, nn2, &CurrHead->diag);
        ++CurrHead->n;
}

// delete the row tree pointed to by p

void Les::DelRow(Row_tr *q)
{
        if (q != NULL) {
                DelRow(q->left);
```

```
            DelRow(q->right);
            if ((q->left == NULL) && (q->right == NULL)) {
                  delete q;
                  q = NULL;
                  }
            }
}

// deletes the roots of rows

void Les::DelHeads(Header (**q))
{
      if ((*q) != NULL) {
            DelHeads(&(*q)->top);
            DelRow((*q)->diag);
            DelHeads(&(*q)->bottom);
            if (((*q)->top == NULL) && ((*q)->bottom == NULL)) {
                  delete (*q);
                  (*q) = NULL;
                  }
            }
}

// deletes system from memory

void Les::DelMatrix() {
      DelHeads(&heads);
}

// builds a balanced row tree initialized to 0.0 and 0

Row_tr *Les::RowsTree(int nel) {
      if (nel != 0) {
            int ni = nel / 2;
            int nd = nel -ni -1;
            Row_tr *pij = new Row_tr;
            pij->e = 0.0;
            pij->j =0;
            pij->left = RowsTree(ni);
            pij->right = RowsTree(nd);
            return pij;
      }
      else
      return NULL;
}

// get the pointer to next elem of nd1-th row tree

Row_tr *Les::GetNext(int nd1) {
      Row_tr *ptst = NULL;
      while((ni < (iband+1)) && (ptst == NULL)) {
            ++ni;
            ptst = Loc(nd1, ni);
            if (ptst->j == ni) return ptst;
      }
}

// copy tree to a balanced new one
```

```
void Les::CopyTree(Row_tr *p, int nd1) {
        if (p != NULL) {
                CopyTree(p->left, nd1);
                q = GetNext(nd1);
                p->e = q->e;
                p->j = q->j;
                CopyTree(p->right, nd1);
        }
}

// balance the nd1-th row tree

void Les::Balance(int nd1) {
        CurrHead = LocHeader(nd1);
        Row_tr *qbal = RowsTree(CurrHead->n);
        ni = 0;
        CopyTree(qbal, nd1);
        DelRow(CurrHead->diag);
        CurrHead->diag = qbal;
}

// get the level of the a-tree

void Les::level(Row_tr *a, int l) {
        if (a != NULL) {
                level(a->left, l++);
                if (l > ni) ni = l;
                level(a->right, l++);
        }
}

// get the level of the i-th tree

int Les::GetLevel(int i) {
        Header *p = LocHeader(i);
        ni = 0;
        level(p->diag, ni);
        return ni;
}

// count row elements

void Les::countelem(Row_tr *p, long *n) {
        if (p != NULL) {
                countelem(p->left, &(*n));
                (*n)++;
                countelem(p->right, &(*n));
        }
}

// count of systems row

void Les::countrow(Header *q, long *n) {
        if (q != NULL) {
                countrow(q->top, &(*n));
                countelem(q->diag, &(*n));
                countrow(q->bottom, &(*n));
```

```
	}
}

// count elements of system

long Les::Count() {
	long number = 0;
	countrow(heads, &number);
	return number;
}

// minor of two integers

int minor(int ii, int kk) {
	if (ii < kk) return ii;
	else return kk;
}

// Banded symmetric equation system solver

void Les::Bansol(task duty) {
	int m, mr, i, j;
	float pivot, cp;
	Row_tr *a, *a1, *r, *r1;
// reduce system to triangle matrix
	int nrs = neq - 1;
	if (duty == trian) {
		for (int nn = 1; nn <= nrs; nn++) {
			m = nn -1;
			mr = minor(iband, neq - m);
			pivot = Get(nn, 1);
			for (int l = 2; l <= mr; l++) {
				a = Loc(nn, l);
				if (a == NULL) cp = 0.0;
				else cp = a->e / pivot;
				i = m + l;
				j = 0;
				for (int k = l; k <= mr; k++) {
					j+= 1;
					a1 = Loc(nn, k);
					if (a1 != NULL) {
						a = Loc(i, j);
						if (a == NULL) {
							AddElem(0.0, i, j);
							a = Loc(i, j);
						}
						a->e -= cp * a1->e;
					}
				}; a = Loc(nn, l);
				a->e = cp;
			}
		}; reduced = true;
		return;
	}
// solve reduced system for the right hand side — forward reduction

	if (duty == reslv) {
		for (int nn = 1; nn <= nrs; nn++) {
```

```
                        m = nn-1;
                        mr = minor(iband, neq - m);
                        r = Loc(nn, iband + 1);
                        a = Loc(nn, 1);
                        cp = r->e;
                        r->e = cp / a->e;
                        for (int l = 2; l <= mr; l++) {
                              i = m + l;
                              r = Loc(i, iband + 1);
                              a = Loc(nn, l);
                              if (a != NULL) r->e -= a->e * cp;
                        }
                  };
                  r = Loc(neq, iband +1);
                  a = Loc(neq, 1);
                  r->e /= a->e;
// back substitution

                  for (i = 1; i <= nrs; i++) {
                        int nn = neq - i;
                        m = nn -1;
                        mr = minor(iband, neq - m);
                        for (int k = 2; k <= mr; k++) {
                              j = m + k;
                              r = Loc(nn, iband +1);
                              a = Loc(nn, k);
                              if (q != NULL) {
                                    r1 = Loc(j, iband + 1);
                                    r->e -= a->e * r1->e;
                              }
                        }
                  }
            }
}

// test-program

Les *Coef;
void InputMat(int ne, int ib) {
      float xl;
      for (int im = 1; im <= ne; im ++) {
            int mr = minor(ib, ne - im +1);
            for (int jm = 1; jm <= mr; jm++) {
                  if (jm == 1)
                        xl = 10.0;
                  else
                        xl = 1.0;
                  Coef->AddElem(xl, im, jm);
            }
            Coef->Balance(im);
      };
}

void PrintMat(int ne, int ib) {
      for (int i = 1; i <= ne; i++) {
            int mr = minor(ib, ne - i + 1);
            for (int j = 1; j <= mr; j++) {
                  float xl = Coef->Get(i, j);
```

```
                cout << setiosflags(ios::left)
                     << setw(10)
                     << setprecision(6);
                cout << xl << " ";
                }; cout << "\n";
           }; return;
}

void InputRHS(int ne, int ib) {
      int n1 = ne -ib +1;
      int i3 = ib + 8;
      for (int i = 1; i <= ne; i++) {
            if (i <= ib) i3 += 1;
            if (i > n1) i3 -= 1;
            Coef->AddElem(i3*1.0, i, ib+1);
      }
}

void PrintRHS(int ne, int ib) {
      for(int i = 1; i <= ne; i++) cout << Coef->Get(i, ib+1) <<"\n";
}

main()
{
      int ne, ib;
      char ch;
      cout << "\nNumber of equations ";
      cin >> ne;
      cout << "\nHalf bandwidth ";
      cin >> ib;
      Coef = new Les(ne, ib);
      InputMat(ne, ib);
      PrintMat(ne, ib);
      cout << "In total were generated "<< Coef->Count() << " elements\n";
      cout << "The maximum level of the tree is " << Coef->GetLevel(1) <<"\n";
      cin >> ch;
      InputRHS(ne, ib);
      PrintRHS(ne, ib);
      cin >> ch;
      Coef->Bansol(trian);
      PrintMat(ne, ib);
      cin >> ch;
      Coef->Bansol(reslv);
      PrintRHS(ne, ib);
      Coef->~Les();
}
x------------------------------------------------------------------------------x
```