| routines | ch_fac | lu_fac | lq_fac | rt_ltr | lf_ltr | mm_prd | mo_prd | mt_trp |
|---|---|---|---|---|---|---|---|---|
| time [sec] | 14.313 | 20.801 | 45.619 | 0.654 | 0.679 | 47.777 | 20.853 | 0.786 |
| mflops | 186.31 | 256.40 | 467.64 | 6.12 | 5.89 | 334.89 | 767.28 | 0.0 |

Table 3. Performance examples for the single precision PRDLA1.

| routines | ch_fac | lu_fac | lq_fac | rt_ltr | lf_ltr | mm_prd | mo_prd | mt_trp |
|---|---|---|---|---|---|---|---|---|
| time [sec] | 26.020 | 38.850 | 99.924 | 0.767 | 0.831 | 90.773 | 72.665 | 1.570 |
| mflops | 102.49 | 137.28 | 213.50 | 5.22 | 4.81 | 176.26 | 220.19 | 0.0 |

Table 4. Performance examples for the double precision PRDLA1.

- **lq_fac** $(A, B, m, n, r, p)$ — $error = \frac{fnorm(LQ-A)}{fnorm(A)}$.
- **rt_ltr** $(A, B, n, r, p)$ — $error = \frac{fnorm(AX-B)}{fnorm(A)}$.
- **lf_ltr** $(A, b, n, p)$ — $error = \frac{fnorm(x^T A-b^T)}{fnorm(A)}$.

## C.2 Sample Performance

- **Dimensions of input matrices** — $2000 \times 2000$.
- **Number of processors used** — 32.

| routines | ch_fac | lu_fac | lq_fac | rt_ltr | lf_ltr | mm_prd | mo_prd | mt_trp |
|---|---|---|---|---|---|---|---|---|
| time [sec] | 37.396 | 49.750 | 213.950 | 0.654 | 0.676 | 169.678 | 185.767 | 0.786 |
| mflops | 71.31 | 107.20 | 99.71 | 6.12 | 5.92 | 94.30 | 86.13 | 0.0 |

Table 1. Performance examples for the single precision PRDLA0.

| routines | ch_fac | lu_fac | lq_fac | rt_ltr | lf_ltr | mm_prd | mo_prd | mt_trp |
|---|---|---|---|---|---|---|---|---|
| time [sec] | 49.443 | 65.575 | 264.765 | 0.767 | 0.824 | 267.109 | 229.034 | 1.570 |
| mflops | 53.93 | 81.33 | 80.57 | 5.22 | 4.85 | 59.90 | 69.86 | 0.0 |

Table 2. Performance examples for the double precision PRDLA0.

## C.1 Online Test

A group of major subroutines in PRDLA can be tested by running the program *PFM_host* and then simply following the instruction printed on the screen. The performance results include the running time, megaflop rate, and error estimate. Matrices of given sizes are randomly generated. The number of processors used may vary from 1 to 32.

The following formulas are used to estimate the number of operations of the major subroutines:

- **ch_fac** $(A, n, p)$ — $\frac{n^3}{3}$.
- **lu_fac** $(A, ipvt, r, n, p)$ — $\frac{2n^3}{3}$.
- **lq_fac** $(A, B, m, n, r, p)$ — $2(m+r)^2(n - \frac{m+r}{3})$.
- **rt_ltr** $(A, B, n, r, p)$ — $rn^2$.
- **lf_ltr** $(A, b, n, p)$ — $n^2$.
- **mm_prd** $(A, B, C, m, k, n, p)$ — $2kmn$.
- **mo_prd** $(A, B, C, m, k, n, p)$ — $2kmn$.

The following examples illustrate how the errors are estimated, where $fnorm$ is the matrix Frobenius norm:

- **ch_fac** $(A, n, p)$ — $error = \frac{fnorm(R^T R - A)}{fnorm(A)}$.
- **lu_fac** $(A, ipvt, r, n, p)$ — $error = \frac{fnorm(LU - AQ)}{fnorm(A)}$.

# Appendix C

# Performance

```
            send0(&A[ip][0],n*sizeof(double),i,kmp);
            recv0(&t[0],n*sizeof(double),i);
            }

         for (j=0;j<n;j++) A[ip][j]=t[j];

           }

      }

   else {           /* The ith row is not on this processor.  */

      if (kmp==myid) {  /* The kth row is on this processor.  */

         kp=k/p;

      /* Permute (i,k) rows with processor imp.                */

         if (p>1) {
            recv0(&t[0],n*sizeof(double),i);
            send0(&A[kp][0],n*sizeof(double),i,imp);
            }

         for (j=0;j<n;j++) A[kp][j]=t[j];

         }

      }

   N0: if (f==1) i++; else i--;

   }

free_v1(t);                /* Free allocated memory for t.  */

}
```

```
mp=m/p;
if (myid<m%p) mp++;          /* The # of local rows of A.     */

malloc_v1(&t,n);             /* Allocate memory for t.        */

if (f==1) i=0; else i=m-1;  /* If f=1 forward else backward. */

while (i>=0 && i<m) {

  if (ipvt[i]==i) goto N0;  /* No permutation.               */

  k=ipvt[i];
  imp=i%p;        /* The processor containing ith row of A.  */
  kmp=k%p;        /* The processor containing kth row of A.  */

  if (imp==myid) {     /* The ith row is on this processor.  */

     ip=i/p;

     if (kmp==myid) {  /* The kth row is on this processor.  */

        kp=k/p;

        for (j=0;j<n;j++) {       /* Permute (i,k) rows.     */
            temp=A[ip][j];
            A[ip][j]=A[kp][j];
            A[kp][j]=temp;
            }

        }

     else {        /* The kth row is not on this processor.  */

     /* Permute (i,k) rows with processor kmp.               */

        if (p>1) {
```

## B.9 mt_rpm

```
/*                                                                    */
/*   File Name:                                                       */
/*                                                                    */
/*         mt_rpm.c                                                   */
/*                                                                    */
/*   Function:                                                        */
/*                                                                    */
/*         Compute PA (or P^{T}A)                                     */
/*         for some permutation matrix P.                             */
/*                                                                    */
/*   Subroutines Used:                                                */
/*                                                                    */
/*         who0  ---- PICL                                            */
/*         send0 ---- PICL                                            */
/*         recv0 ---- PICL                                            */
/*                                                                    */
/*         malloc_v1 ---- PRDLA [tl_etc.c]                            */
/*         free_v1   ---- PRDLA [tl_etc.c]                            */
/*                                                                    */

#include <stdio.h>
#include <math.h>
#include <cube.h>
#include "DLA_head.h"

int mt_rpm(A,ipvt,f,m,n,p)
  int f,m,n,p,*ipvt;
  double *A[];
  {
   int i,j,k,ip,kp,kmp,imp,mp,nproc,myid,host;
   double *t,temp;

   who0(&nproc,&myid,&host);   /* Obtain cube info.                  */
```

```
          i1+=p;
          }

  /* Send local elements to processor l.                */

     if (np1>0 && mp>0 && p>1)
        send0(&t[0],np1*mp*sizeof(double),myid,l);

     if (l<nmp) np1--;

     }

  l++;

  }

free_v1(t);            /* Free allocated memory for t.  */

}
```

```
        k1=(myid+k)%p;

        if (k1<mmp) mp1++;

     /* Receive elements of A on processor k1.         */

        if (np>0 && mp1>0 && p>1)
           recv0(&t[0],np*mp1*sizeof(double),k1);

        j1=k1;
        for (j=0;j<mp1;j++) {
            j2=j;
            for (i=0;i<np;i++) {
                B[i][j1]=t[j2];
                j2+=mp1;
                }
            j1+=p;
            }

        if (k1<mmp) mp1--;

        }

   }

else {

   if (l<nmp) np1++;

   i1=l;
   j1=0;
   for (i=0;i<np1;i++) {
       for (j=0;j<mp;j++) {
           t[j1]=A[j][i1];
           j1++;
           }
```

```
sync0();

mp1=m/p;
mmp=m%p;

mp=mp1;
if (myid<mmp) mp++;             /* The # of local rows of A. */

np1=n/p;
nmp=n%p;

np=np1;
if (myid<nmp) np++;             /* The # of local rows of B. */

malloc_v1(&t,(mp1+1)*(np1+1));  /* Allocate memory for t.    */

l=0;

while (l<p) {

  if (myid==l) {

  /* Transpose elements of A on processor l.            */

     i1=l;
     for (i=0;i<np;i++) {
         j1=l;
         for (j=0;j<mp;j++) {
             B[i][j1]=A[j][i1];
             j1+=p;
             }
         i1+=p;
         }

  /* Transpose elements of A on other processors.       */

     for (k=1;k<p;k++) {
```

## B.8 mt_trp

```
/*                                                                    */
/*   File Name:                                                       */
/*                                                                    */
/*         mt_trp.c                                                   */
/*                                                                    */
/*   Function:                                                        */
/*                                                                    */
/*         Compute A^{T} => B.                                        */
/*                                                                    */
/*   Subroutines Used:                                                */
/*                                                                    */
/*         who0  ---- PICL                                            */
/*         sync0 ---- PICL                                            */
/*         send0 ---- PICL                                            */
/*         recv0 ---- PICL                                            */
/*                                                                    */
/*         malloc_v1 ---- PRDLA [tl_etc.c]                            */
/*         free_v1   ---- PRDLA [tl_etc.c]                            */
/*                                                                    */

#include <stdio.h>
#include <math.h>
#include <cube.h>
#include "DLA_head.h"

int mt_trp(A,B,m,n,p)
  int m,n,p;
  double *A[],*B[];
  {
   int i,i1,j,j1,j2,k,k1,l,mp,mp1,mmp,np,np1,nmp,nproc,myid,host;
   double *t;

   who0(&nproc,&myid,&host);       /* Obtain cube info.          */
```

```
            for (l2=0;l2<k;l2++) {
                temp+=A[i][l2]*t[j2];
                j2++;
                }

            C[i][j1]=temp;

            j1+=p;

            }

        }

 /* Send columns of B on processor l1 to 'right'. */

    if (l<p-1 && p>1)
       send0(&t[0],np*k*sizeof(double),l1,right);

    N0:       ;

    }

free_v1(t);       /* Free allocated memory for t. */

}
```

```
            for (l=0;l<k;l++) {
                temp+=A[i][l]*t[j2];
                j2++;
                }

            C[i][j1]=temp;

            j1+=p;

            }

        }

/* Compute elements of C using nonlocal columns of B.           */

   for (l=1;l<p;l++) {

        l1=(p+myid-l)%p;

        np=np1;
        if (l1<nmp) np++;

        if (np==0) goto N0;

    /* Receive columns of B from processor l1.       */

        if (p>1)
           recv0(&t[0],np*k*sizeof(double),l1);

        for (i=0;i<mp;i++) {

            j1=l1;
            j2=0;

            for (j=0;j<np;j++) {

                temp=0.0e0;
```

```
   mp=m/p;
   if (myid<m%p) mp++;       /* The # of local rows of A.       */

   np1=n/p;
   nmp=n%p;

   np=np1;
   if (myid<nmp) np++;       /* The # of local rows of B.       */

   right=(myid+1)%p;         /* The id of next right processor. */

   malloc_v1(&t,(np1+1)*k);  /* Allocate memory for t.          */

   j1=0;
   for (i=0;i<np;i++)
       for (j=0;j<k;j++) {
           t[j1]=B[i][j];
           j1++;
           }

/* Send columns of B to 'right'.                                 */

   if (np>0 && p>1)
      send0(&t[0],np*k*sizeof(double),myid,right);

/* Compute elements of C using local columns of B.               */

   for (i=0;i<mp;i++) {

       j1=myid;
       j2=0;

       for (j=0;j<np;j++) {

           temp=0.0e0;
```

## B.7 mo_prd

```
/*                                                                    */
/*   File Name:                                                       */
/*                                                                    */
/*         mo_prd.c                                                   */
/*                                                                    */
/*   Function:                                                        */
/*                                                                    */
/*         Compute AB^{T} => C.                                       */
/*                                                                    */
/*   Subroutines Used:                                                */
/*                                                                    */
/*         who0  ---- PICL                                            */
/*         send0 ---- PICL                                            */
/*         recv0 ---- PICL                                            */
/*                                                                    */
/*         malloc_v1 ---- PRDLA [tl_etc.c]                            */
/*         free_v1   ---- PRDLA [tl_etc.c]                            */
/*                                                                    */

#include <stdio.h>
#include <math.h>
#include <cube.h>
#include "DLA_head.h"

int mo_prd(A,B,C,m,k,n,p)
  int m,k,n,p;
  double *A[],*B[],*C[];
  {
   int i,i1,j,j1,j2,l,l1,l2,nproc,myid,host,mp,np,np1,
       nmp,right;
   double temp,*t;

   who0(&nproc,&myid,&host);     /* Obtain cube info.              */
```

```
                temp+=A[i][j1]*t[l3];
                j1+=p;
                l3++;
                }

            C[i][j]+=temp;

            }

        }

 /* Send rows of B on processor l1 to 'right'. */

    if (l<p-1 && p>1)
       send0(&t[0],kp*n*sizeof(double),l1,right);

    N0:      ;

    }

free_v1(t);    /* Free allocated memory for t. */

}
```

```
            temp+=A[i][j1]*t[l3];
            j1+=p;
            l3++;
            }

        C[i][j]=temp;

        }

    }

/* Compute partial sum of C using nonlocal rows of B.          */

   for (l=1;l<p;l++) {

       l1=(p+myid-l)%p;

       kp=kp1;
       if (l1<kmp) kp++;

       if (kp==0) goto N0;

    /* Receive rows of B from processor l1.       */

       if (p>1)
          recv0(&t[0],kp*n*sizeof(double),l1);

       for (i=0;i<mp;i++) {

           l3=0;

           for (j=0;j<n;j++) {

               temp=0.0e0;

               j1=l1;
               for (l2=0;l2<kp;l2++) {
```

```
   mp=m/p;
   if (myid<m%p) mp++;       /* The # of local rows of A.       */

   kp1=k/p;
   kmp=k%p;

   kp=kp1;
   if (myid<kmp) kp++;       /* The # of local rows of B.       */

   right=(myid+1)%p;         /* The id of next right processor. */

   malloc_v1(&t,(kp1+1)*n);  /* Allocate memory for t.          */

   i1=0;
   for (j=0;j<n;j++)
       for (i=0;i<kp;i++) {
           t[i1]=B[i][j];
           i1++;
           }

/* Send rows of B to 'right'.                                    */

   if (kp>0 && p>1)
      send0(&t[0],kp*n*sizeof(double),myid,right);

/* Compute partial sum of C using local rows of B.               */

   for (i=0;i<mp;i++) {

       l3=0;

       for (j=0;j<n;j++) {

           temp=0.0e0;

           j1=myid;
           for (l2=0;l2<kp;l2++) {
```

## B.6 mm_prd

```
/*                                                                    */
/*   File Name:                                                       */
/*                                                                    */
/*         mm_prd.c                                                   */
/*                                                                    */
/*   Function:                                                        */
/*                                                                    */
/*         Compute AB => C.                                           */
/*                                                                    */
/*   Subroutines Used:                                                */
/*                                                                    */
/*         who0  ---- PICL                                            */
/*         send0 ---- PICL                                            */
/*         recv0 ---- PICL                                            */
/*                                                                    */
/*         malloc_v1 ---- PRDLA [tl_etc.c]                            */
/*         free_v1   ---- PRDLA [tl_etc.c]                            */
/*                                                                    */

#include <stdio.h>
#include <math.h>
#include <cube.h>
#include "DLA_head.h"

int mm_prd(A,B,C,m,k,n,p)
  int m,k,n,p;
  double *A[],*B[],*C[];
  {
   int i,i1,j,j1,l,l1,l2,l3,nproc,myid,host,mp,kp,kp1,kmp,right;
   double temp,*t;

   who0(&nproc,&myid,&host);  /* Obtain cube info.                */
```

```
  /* Compute local solutions.                                  */

     b[i]=(sum[0]+psum[i])/L[i1][i];

     for (k=0;k<p-1;k++) {
         i2=i-k-1;
         if (i2>=0)
            sum[k]=sum[k+1]-L[i1][i2]*b[i]+psum[i2];
         }

  /* Send local solutions to 'left'.                           */

     if (i>0 && p>1)
        send0(&sum[0],p*sizeof(double),myid,left);

  /* Update right hand side elements.                          */

     for (k=0;k<i+1-p;k++)
         psum[k]-=L[i1][k]*b[i];

     }

  i--;

  }

root=0;

if (p>1) {              /* Obtain all components of b.     */
   gsum0(&b[0],n,5,n,root);
   bcast0(&b[0],n*sizeof(double),n,root);
   }

free_v1(sum);           /* Free allocated memory for sum.  */
free_v1(psum);          /* Free allocated memory for psum. */

}
```

```
   who0(&nproc,&myid,&host);  /* Obtain cube info.           */

   malloc_v1(&sum,p);     /* Allocate memory for sum.        */
   malloc_v1(&psum,n);    /* Allocate memory for psum.       */

   left=(p+myid-1)%p;     /* The id of next left processor.  */
   right=(myid+1)%p;      /* The id of next right processor. */

/* Initial sum, psum and b.                                   */

   for (k=0;k<p;k++) sum[k]=0.0e0;
   for (k=0;k<n;k++) psum[k]=0.0e0;

   if (myid!=(n-1)%p) goto L1;

   for (k=0;k<p-1;k++) {
       k1=n-k-1;
       sum[k]=b[k1];
       }

   for (k=0;k<n-p+1;k++) psum[k]=b[k];

   L1: for (k=0;k<n;k++) b[k]=0.0e0;

   i=n-1;               /* The current working row.          */

   while (i>=0) {

     i1=i/p;            /* The row # of i on local array.    */

     if (i%p==myid) {

     /* Receive local solutions.                              */

        if (i<n-1 && p>1)
           recv0(&sum[0],p*sizeof(double),right);
```

## B.5 lf_ltr

```
/*                                                                    */
/*   File Name:                                                       */
/*                                                                    */
/*         lf_ltr.c                                                   */
/*                                                                    */
/*   Function:                                                        */
/*                                                                    */
/*         Solve a lower trangular left system xL = b.                */
/*                                                                    */
/*   Subroutines Used:                                                */
/*                                                                    */
/*         who0    ---- PICL                                          */
/*         send0   ---- PICL                                          */
/*         recv0   ---- PICL                                          */
/*         gsum0   ---- PICL                                          */
/*         bcast0  ---- PICL                                          */
/*                                                                    */
/*         malloc_v1 ---- PRDLA [tl_etc.c]                            */
/*         free_v1   ---- PRDLA [tl_etc.c]                            */
/*                                                                    */

#include <stdio.h>
#include <math.h>
#include <cube.h>
#include "DLA_head.h"

int lf_ltr(L,b,n,p)
  int n,p;
  double *L[],*b;
  {
   int i,i1,i2,k,k1,nproc,myid,host,left,right,root;
   double *sum,*psum;
```

```
  }
free_v1(xsub);    /* Free allocated memory for xsub.      */
}
```

```
            j1++;
            }

        B[i1][j]-=temp;
        B[i1][j]/=L[i1][i];

        for (k=0;k<p-1;k++) {
            xsub[j1]=xsub[j1-1];
            j1--;
            }

        xsub[j1]=B[i1][j];

        j1+=p;

        }

/* Send local solutions to 'right'.                    */

   if (i<n-1 && p>1)
      send0(&xsub[0],r*p*sizeof(double),i,right);

/* Update right hand side elements.                    */

   j1=0;
   for (j=0;j<r;j++)
       for (k=0;k<p;k++) {
           k1=i-k;
           for (l=i1+1;l<np;l++)
               if (k1>=0) B[l][j]-=L[l][k1]*xsub[j1];
           j1++;
           }

   }

i++;
```

```
np=n/p;
if (myid<n%p) np++;    /* The # of local rows of L.       */

right=(myid+1)%p;      /* The id of next right processor. */

malloc_v1(&xsub,r*p);  /* Allocate memory for xsub.       */

j1=0;
for (j=0;j<r;j++)
    for (k=0;k<p;k++) {
        xsub[j1]=0.0e0;
        j1++;
        }

i=0;                   /* The current working row.        */

while (i<n) {

  i1=i/p;              /* The row # of i on local array.  */

  if (i%p==myid) {     /* The kth row is on this processor. */

  /* Receive local solutions.                              */

     if (i>0 && p>1)
        recv0(&xsub[0],r*p*sizeof(double),i-1);

  /* Compute local solutions.                              */

     j1=0;

     for (j=0;j<r;j++) {

         temp=0.0e0;
         for (k=0;k<p-1;k++) {
             k1=i-k-1;
             if (k1>=0) temp+=L[i1][k1]*xsub[j1];
```

## B.4 rt_ltr

```
/*                                                                    */
/*   File Name:                                                       */
/*                                                                    */
/*         rt_ltr.c                                                   */
/*                                                                    */
/*   Function:                                                        */
/*                                                                    */
/*         Solve a lower trangular right system LX = B.               */
/*                                                                    */
/*   Subroutines Used:                                                */
/*                                                                    */
/*         who0  ---- PICL                                            */
/*         send0 ---- PICL                                            */
/*         recv0 ---- PICL                                            */
/*                                                                    */
/*         malloc_v1 ---- PRDLA [tl_etc.c]                            */
/*         free_v1   ---- PRDLA [tl_etc.c]                            */
/*                                                                    */

#include <stdio.h>
#include <math.h>
#include <cube.h>
#include "DLA_head.h"

int rt_ltr(L,B,n,r,p)
  int n,r,p;
  double *L[],*B[];
  {
   int i,i1,j,j1,k,k1,l,np,nproc,myid,host,right;
   double temp,*xsub;

   who0(&nproc,&myid,&host);  /* Obtain cube info.           */
```

```
          }
      }
  k++;
  }
free_v1(t);          /* Free allocated memory for t.        */
}
```

```
else {

/* Receive the current Householder vector.                */

   if (p>1)
      recv0(&t[k],(n-k)*sizeof(double),k);

/* Send the Householder vector to next right neighbor.    */

   if (right!=k%p && p>1)
      send0(&t[k],(n-k)*sizeof(double),k,right);

/* Update local rows of A.                                */

   for (i1=i;i1<mp;i1++) {

       temp=0.0e0;
       for (j=k;j<n;j++)
           temp+=t[j]*A[i1][j];

       temp/=t[k];
       for (j=k;j<n;j++)
           A[i1][j]-=temp*t[j];

       }

/* Update local rows of B.                                */

   for (i1=0;i1<rp;i1++) {

       temp=0.0e0;
       for (j=k;j<n;j++)
           temp+=t[j]*B[i1][j];
       temp/=t[k];

       for (j=k;j<n;j++)
           B[i1][j]-=temp*t[j];
```

```
   if (p>1)
      send0(&t[k],(n-k)*sizeof(double),k,right);

/* Update local rows of A.                                    */

   for (i1=i;i1<mp;i1++) {

       temp=0.0e0;
       for (j=k;j<n;j++)
           temp+=t[j]*A[i1][j];
       temp/=t[k];

       for (j=k;j<n;j++)
           A[i1][j]-=temp*t[j];

       }

   for (j=k+1;j<n;j++) A[i][j]=t[j];

/* Update local rows of B.                                    */

   for (i1=0;i1<rp;i1++) {

       temp=0.0e0;
       for (j=k;j<n;j++)
           temp+=t[j]*B[i1][j];
       temp/=t[k];

       for (j=k;j<n;j++)
           B[i1][j]-=temp*t[j];

       }

   i++;

   }
```

```
mp=m/p;
if (myid<m%p) mp++;    /* The # of local rows of A.         */

rp=r/p;
if (myid<r%p) rp++;    /* The # of local rows of B.         */

right=(myid+1)%p;      /* The id of next right processor.   */

malloc_v1(&t,n);       /* Allocate memory for t.            */

tol=1.0e-16;           /* A small tolerance.                */

i=0;                   /* The current local row.            */
k=0;                   /* The current working row.          */

while (k<m) {

  if (k==i*p+myid) {   /* The ith local row is the kth row. */

  /* Compute the Householder vector.                         */

     temp=0.0e0;
     for (j=k;j<n;j++)
         temp+=A[i][j]*A[i][j];
     temp=sqrt(temp);

     if (temp<tol) {
        printf("ill-conditioned system!\n");
        }

     temp=sign(A[i][k])/temp;
     for (j=k;j<n;j++)
         t[j]=A[i][j]*temp;
     t[k]+=1.0e0;

  /* Send the Householder vector to next right neighbor.    */
```

## B.3 lq_fac

```
/*                                                              */
/*   File Name:                                                 */
/*                                                              */
/*         lq_fac.c                                             */
/*                                                              */
/*   Function:                                                  */
/*                                                              */
/*         Compute L and Q such that A = LQ.                    */
/*                                                              */
/*   Subroutines Used:                                          */
/*                                                              */
/*         who0  ---- PICL                                      */
/*         send0 ---- PICL                                      */
/*         recv0 ---- PICL                                      */
/*                                                              */
/*         malloc_v1 ---- PRDLA [tl_etc.c]                      */
/*         free_v1   ---- PRDLA [tl_etc.c]                      */
/*                                                              */

#include <stdio.h>
#include <math.h>
#include <cube.h>
#include "DLA_head.h"

int lq_fac(A,B,m,n,r,p)
  int m,n,r,p;
  double *A[],*B[];
  {
   int i,i1,j,k,mp,rp,nproc,myid,host,right;
   double temp,tol,*t;

   who0(&nproc,&myid,&host);  /* Obtain cube info. */
```

```
    }
  k++;
  }
free_v1(t);          /* Free allocated memory for t.          */
}
```

```
   if (p>1)
      recv0(&t[k],(n-k+1)*sizeof(double),k);

/* Send the current working row  and the index of         */
/* pivoting column to 'right'.                            */

   if (right!=k%p && p>1)
      send0(&t[k],(n-k+1)*sizeof(double),k,right);

/* Column pivoting.                                       */

   l=(int)t[n];
   ipvt[k]=l;

   if (l!=k)
      for (i1=0;i1<np;i1++) {
          temp=A[i1][l];
          A[i1][l]=A[i1][k];
          A[i1][k]=temp;
          }

/* Test if the pivot is too small.                        */

   if (fabs(t[k])<tol) {
      printf("ill-conditioned system!\n");
      break;
      }

/* Update local rows of A.                                */

   for (i1=i;i1<np;i1++) {
       temp=A[i1][k]/t[k];
       for (j=k+1;j<n;j++)
           A[i1][j]-=temp*t[j];
       A[i1][k]=temp;
       }
```

```
            A[i1][k]=temp;
            }

   /* Send the current working row  and the index of          */
   /* pivoting column to 'right'.                             */

      for (j=k;j<n;j++) t[j]=A[i][j];

      t[n]=(double)l;

      if (p>1)
         send0(&t[k],(n-k+1)*sizeof(double),k,right);

   /* Test if the pivot is too small.                         */

      if (kmax<tol) {
         printf("ill-conditioned system!\n");
         break;
         }

   /* Update local rows of A.                                 */

      for (i1=i+1;i1<np;i1++) {
          temp=A[i1][k]/t[k];
          for (j=k+1;j<n;j++)
              A[i1][j]-=temp*t[j];
          A[i1][k]=temp;
          }

      i++;

      }

 else {       /* The kth row is not on this processor.       */

   /* Receive the current working row and the index           */
   /* pivoting column.                                        */
```

```
np=n/p;
if (myid<n%p) np++;    /* The # of local rows of A.         */

right=(myid+1)%p;      /* The id of next right processor.   */

malloc_v1(&t,n+1);     /* Allocate memory for t.            */

tol=1.0e-16;           /* A small tolerance.                */

for (i=0;i<n;i++) ipvt[i]=i;    /* Initial pivoting vector. */

i=0;                   /* The current local row.            */
k=0;                   /* The current working row.          */

while (k<n) {

  if (k%p==myid) {     /* The kth row is on this processor. */

  /* Column pivoting.                                        */

     kmax=fabs(A[i][k]);
     l=k;

     for (j=k+1;j<n;j++) {
         if (fabs(A[i][j])>kmax) {
            kmax=fabs(A[i][j]);
            l=j;
            }
         }

     ipvt[k]=l;

     if (l!=k)
        for (i1=0;i1<np;i1++) {
            temp=A[i1][l];
            A[i1][l]=A[i1][k];
```

## B.2 lu_fac

```
/*                                                              */
/*   File Name:                                                 */
/*                                                              */
/*         lu_fac.c                                             */
/*                                                              */
/*   Function:                                                  */
/*                                                              */
/*         Compute L and U such that AQ = LU                    */
/*         for some pivoting matrix Q.                          */
/*                                                              */
/*   Subroutines Used:                                          */
/*                                                              */
/*         who0  ---- PICL                                      */
/*         send0 ---- PICL                                      */
/*         recv0 ---- PICL                                      */
/*                                                              */
/*         malloc_v1 ---- PRDLA [tl_etc.c]                      */
/*         free_v1   ---- PRDLA [tl_etc.c]                      */
/*                                                              */

#include <stdio.h>
#include <math.h>
#include <cube.h>
#include "DLA_head.h"

int lu_fac(A,ipvt,n,p)
  int n,p,*ipvt;
  double *A[];
  {
   int i,i1,j,k,l,np,nproc,myid,host,right;
   double temp,tol,kmax,*t;

   who0(&nproc,&myid,&host);  /* Obtain cube info.             */
```

```
         send0(&t[k],(n-k)*sizeof(double),k,right);

   /* Test if the diagonal element is too small.          */

      if (t[k]<tol) {

         printf("ill-conditioned system!\n");
         break;

         }

   /* Update local rows of A.                             */

      for (i1=i;i1<np;i1++) {
          temp=t[r];
          for (j=r;j<n;j++)
              A[i1][j]-=temp*t[j];
          r+=p;
          }

      }

   k++;

   }

free_v1(t);           /* Free allocated memory for t.     */

}
```

```
   temp=sqrt(A[i][k]);

   for (j=k;j<n;j++) {
       A[i][j]/=temp;
       t[j]=A[i][j];
       }

/* Send the row to next right neighbor.                */

   if (p>1 && k<n-1)
      send0(&t[k],(n-k)*sizeof(double),k,right);

/* Update local rows of A.                             */

   r+=p;
   for (i1=i+1;i1<np;i1++) {
       temp=t[r];
       for (j=r;j<n;j++)
           A[i1][j]-=temp*t[j];
       r+=p;
       }

   i++;

   }

else {   /* The ith local row is not the working row.  */

/* Receive the current working row.                    */

   if (p>1)
      recv0(&t[k],(n-k)*sizeof(double),k);

/* Send the row to next right neighbor.                */

   if (p>1 && right!=k%p && k<beta)
```

```
np1=n/p;
nmp=n%p;

np=np1;
if (myid<nmp) np++;    /* The # of local rows of A.       */

malloc_v1(&t,n);       /* Allocate memory for t.          */

right=(myid+1)%p;      /* The id of next right processor. */

if (right<nmp) np1++;
beta=(np1-1)*p+right;  /* The last row of A on 'right'.   */

tol=1.0e-16;           /* A small tolerance.              */

i=0;                   /* The current local row.          */
k=0;                   /* The current working row.        */

while (i<np) {

   r=i*p+myid;   /* The actual row # of i.               */

   if (k==r) {   /* The ith local row is the working row. */

   /* Test if the diagonal element is too small.          */

      if (A[i][k]<tol) {

         if (p>1 && k<n-1)
            send0(&A[i][k],(n-k)*sizeof(double),k,right);

         printf("ill-conditioned system!\n");
         break;

         }

   /* Compute the corresponding row of R.                 */
```

## B.1 ch_fac

```
/*                                                                    */
/*   File Name:                                                       */
/*                                                                    */
/*         ch_fac.c                                                   */
/*                                                                    */
/*   Function:                                                        */
/*                                                                    */
/*         Compute R such that A = R^{T}R.                            */
/*                                                                    */
/*   Subroutines Used:                                                */
/*                                                                    */
/*         who0  ---- PICL                                            */
/*         send0 ---- PICL                                            */
/*         recv0 ---- PICL                                            */
/*                                                                    */
/*         malloc_v1 ---- PRDLA [tl_etc.c]                            */
/*         free_v1   ---- PRDLA [tl_etc.c]                            */
/*                                                                    */

#include <stdio.h>
#include <math.h>
#include <cube.h>
#include "DLA_head.h"

int ch_fac(A,n,p)
  int n,p;
  double *A[];
  {
   int i,i1,j,k,r,np,np1,nmp,nproc,myid,host,right,beta;
   double temp,tol,*t;

   who0(&nproc,&myid,&host);  /* Obtain cube info.          */
```

# Appendix B

# Major Source Code

## A.9 mt_rpm

**Algorithm 9** { **mt_rpm** $(A, ipvt, f, m, n, p)$ }

```
i = 0  (= m − 1 if f = 0)
while i < m ∧ i ≥ 0
   if ipvt(i) = i, goto N0:
   k = ipvt(i)
   imp = i%p, kmp = k%p
   ip = i/p, kp = k/p
   if imp = myid
      if kmp = myid
         t(0 : n − 1) = A(ip, 0 : n − 1)
         A(ip, 0 : n − 1) = A(kp, 0 : n − 1)
         A(kp, 0 : n − 1) = t(0 : n − 1)
      else
         send(A(ip, 0 : n − 1), kmp)
         receive(A(ip, 0 : n − 1), kmp)
      end
   else
      if kmp = myid
         send(A(kp, 0 : n − 1), imp)
         receive(A(kp, 0 : n − 1), imp)
      end
   end
   N0: i = i + 1  (= i − 1 if f = 0)
end
```

Figure A.9: Matrix row permutation.

## A.8 mt_trp

**Algorithm 8** { **mt_trp** $(A, B, m, n, p)$ }

$mp = m/p$, $np = n/p$ { assuming $p|m$ and $p|n$. }
$l = 0$
**while** $l < p$
  $l1 = (mp - 1)p + l$
  $l2 = (np - 1)p + l$
  **if** $myid = l$
    $B(0 : np - 1, l : p : l1) = (A(0 : mp - 1, l : p : l2))^T$
    **for** $k = 1 : p - 1$ **do**
      $k1 = (l + k)\%p$
      $k2 = (mp - 1)p + k1$
      **receive**$(T(0 : mp - 1, 0 : np - 1),\ k1)$
      $B(0 : np - 1, k1 : p : k2) = (T(0 : mp - 1, 0 : np - 1))^T$
    **end**
  **else**
    **send**$(A(0 : mp - 1, l : p : l2), l)$
  **end**
  $l = l + 1$
**end**

Figure A.8: Matrix transposition.

## A.7 mo_prd

**Algorithm 7** { **mo_prd** $(A, B, C, m, k, n, p)$ }

$mp = m/p$, $np = n/p$ { assuming $p|m$ and $p|n$. }
$l1 = myid$, $l2 = (np-1)p + myid$
$C(0:mp-1, l1:p:l2) = A(0:mp-1, 0:k-1)(B(0:np-1, 0:k-1))^T$
**send**$(B(0:np-1, 0:k-1),\ right)$
**for** $l = 1:p-1$ **do**
    **receive**$(B(0:np-1, 0:k-1),\ left)$
    **send**$(B(0:np-1, 0:k-1),\ right)$, **if** $l \neq p-1$
    $l1 = (p + myid - l)\%p$
    $l2 = (np-1)p + l1$
    $C(0:mp-1, l1:p:l2) =$
        $A(0:mp-1, 0:k-1)(B(0:np-1, 0:k-1))^T$
**end**

Figure A.7: Matrix matrix outer product.

## A.6 mm_prd

**Algorithm 6** { **mm_prd** $(A, B, C, m, k, n, p)$ }

$mp = m/p$, $kp = k/p$ { assuming $p|m$ and $p|k$. }
$l1 = myid$, $l2 = (kp-1)p + myid$
$C(0:mp-1, 0:n-1) = A(0:mp-1, l1:p:l2)B(0:kp-1, 0:n-1)$
**send**$(B(0:kp-1, 0:n-1),\ right)$
**for** $l = 1:p-1$ **do**
    **receive**$(B(0:kp-1, 0:n-1),\ left)$
    **send**$(B(0:kp-1, 0:n-1),\ right)$, **if** $l \neq p-1$
    $l1 = (p + myid - l)\%p$
    $l2 = (kp-1)p + l1$
    $C(0:mp-1, 0:n-1) = C(0:mp-1, 0:n-1)$
        $+A(0:mp-1, l1:p:l2)B(0:kp-1, 0:n-1)$
**end**

Figure A.6: Matrix matrix product.

## A.5 lf_ltr

**Algorithm 5** { lf_ltr $(L, b, n, p)$ }

**if** $myid = (n-1)\%p$
  $sum(0:p-2) = b(n-1:n-p+1)$
  $psum(0:n-p) = b(0:n-p)$
**end**
$i = n-1$
**while** $i \geq 0$
  $i1 = i/p$
  **if** $i\%p = myid$
    **receive**$(sum(0:p-1),\ right)$, **if** $i < n-1$
    $b(i) = (sum(0) + psum(i))/L(i1, i)$
    $sum(0:p-2) = sum(1:p-1)$
      $-L(i1, i-1:i-p+1) * b(i) + psum(i-1:i-p+1)$
    **send**$(sum(0:p-1),\ left)$, **if** $i > 0$
    $psum(0:i-p) = psum(0:i-p) - L(i1, 0:i-p) * b(i)$
  **end**
  $i = i-1$
**end**
Globally combine $b$ to get each processor a copy of entire $b$.

Figure A.5: Lower triangular left system solution.

## A.4 rt_ltr

**Algorithm 4** { **rt_ltr** $(L, B, n, r, p)$ }

$i = 0$, $np = n/p$ { assuming $p|n$. }
**while** $i < n$
  $i1 = i/p$
  **if** $i\%p = myid$
    **receive**$(xsub(0 : p-1, 0 : r-1),\ left)$, **if** $i > 0$
    **for** $j = 0 : r-1$ **do**
      $temp = L(i1, i-1 : i-p+1)xsub(0 : p-2, j)$
      $B(i1, j) = (B(i1, j) - temp)/L(i1, i)$
      $xsub(1 : p-1, j) = xsub(0 : p-2, j)$
      $xsub(0, j) = B(i1, j)$
    **end**
    **send**$(xsub(0 : p-1, 0 : r-1),\ right)$, **if** $i < n-1$
    **for** $j = 0 : r-1$ **do**
      $B(i1+1 : np-1, j) = B(i1+1 : np-1, j)$
        $-L(i1+1 : np-1, i : i-p+1) * xsub(0 : p-1, j)$
    **end**
  **end**
  $i = i + 1$
**end**

Figure A.4: Lower triangular right system solution.

## A.3 lq_fac

**Algorithm 3** { **lq_fac** $(A, B, m, n, r, p)$ }

$i = 0$, $k = 0$
**while** $k < m$
  **if** $myid = k\%p$
    $t(k : n-1) =$ **house**$(A(i, k : n-1))$
    **send**$(t(k : n-1),\ right)$ **if** $necessary$
    Update local rows of both $A$ and $B$ with $t(k : n-1)$.
    $i = i + 1$
  **else**
    **receive**$(t(k : n-1),\ left)$
    **send**$(t(k : n-1),\ right)$ **if** $necessary$
    Update local rows of both $A$ and $B$ with $t(k : n-1)$.
  **end**
  $k = k + 1$
**end**

Figure A.3: LQ factorization.

## A.2 lu_fac

**Algorithm 2** { **lu_fac** $(A, ipvt, n, p)$ }

$i = 0$, $k = 0$
**while** $k < n$
  **if** $myid = k\%p$
    Permute columns and determine the rank.
    **send**$(A(i, k : n-1),\ right)$ **if** $necessary$
    Update local rows of $A$ with $A(i, k : n-1)$.
    $i = i + 1$
  **else**
    **receive**$(t(k : n-1),\ left)$
    **send**$(t(k : n-1),\ right)$ **if** $necessary$
    Permute columns and determine the rank.
    Update local rows of $A$ with $t(k : n-1)$.
  **end**
  $k = k + 1$
**end**

Figure A.2: LU factorization.

## A.1 ch_fac

**Algorithm 1** { **ch_fac** $(A, n, p)$ }

$k = 0$, $i = 0$, $np = n/p$ { assuming $p|n$. }
**while** $i < np$
  **if** $i * p + myid = k$
    $A(i, k : n-1) = A(i, k : n-1)$ / $\sqrt{A(i,k)}$
    **send**$(A(i, k : n-1),\ right)$ **if** $necessary$
    Update local rows of $A$ with $A(i, k : n-1)$.
    $i = i + 1$
  **else**
    **receive**$(t(k : n-1),\ left)$
    **send**$(t(k : n-1),\ right)$ **if** $necessary$
    Update local rows of $A$ with $t(k : n-1)$.
  **end**
  $k = k + 1$
**end**

Figure A.1: Cholesky factorization.

# Abbreviations

- **send** — send message.
- **receive** — receive message.
- **house** — Householder vector.
- *myid* — processor id $(0, \ldots, p-1)$.
- *left* — adjacent left processor.
- *right* — adjacent right processor.

# Appendix A

# Major Algorithms

## 3.31 vs_prd

Given a real vector $v$ and a real scalar $s$, this subroutine computes $sv$. The result overwrites $v$.

### Syntax

**vs_prd** $(v, s, n, p)$

### Arguments

| | |
|---|---|
| $v$ | A real array of size $n$. |
| $s$ | A real scalar. |
| $n$ | An integer. An array dimension. |
| $p$ | An integer. The number of processors used. |

### Notes

Array $v$ is not distributed.

## 3.30 vs_add

Given a real vector $v$ and a real scalar $s$, this subroutine adds $s$ to all elements of $v$. The result overwrites $v$.

**Syntax**

**vs_add** $(v, s, n, p)$

**Arguments**

| | |
|---|---|
| $v$ | A real array of size $n$. |
| $s$ | A real scalar. |
| $n$ | An integer. An array dimension. |
| $p$ | An integer. The number of processors used. |

**Notes**

Array $v$ is not distributed.

## 3.29 vs_asg

Given a real vector $v$ and a real scalar $s$, this subroutine sets all elements of $v$ to $s$.

**Syntax**

**vs_asg** $(v, s, n, p)$

**Arguments**

| | |
|---|---|
| $v$ | A real array of size $n$. |
| $s$ | A real scalar. |
| $n$ | An integer. An array dimension. |
| $p$ | An integer. The number of processors used. |

**Notes**

Array $v$ is not distributed.

## 3.28 vv_prd

Given real vectors $v$ and $u$, this subroutine computes $v^T u$. The result is sent to $t$.

**Syntax**

**vv_prd** $(v, u, \&t, n, p)$

**Arguments**

| | |
|---|---|
| $v$ | A real array of size $n$. |
| $u$ | A real array of size $n$. |
| $t$ | A real scalar. |
| $n$ | An integer. An array dimension. |
| $p$ | An integer. The number of processors used. |

**Notes**

Arrays $v$ and $u$ are not distributed.

## 3.27 vv_sbt

Given real vectors $v$ and $u$, this subroutine computes $v - u$. The result overwrites $v$.

**Syntax**

**vv_sbt** $(v, u, n, p)$

**Arguments**

| | |
|---|---|
| $v$ | A real array of size $n$. |
| $u$ | A real array of size $n$. |
| $n$ | An integer. An array dimension. |
| $p$ | An integer. The number of processors used. |

**Notes**

Arrays $v$ and $u$ are not distributed.

## 3.26 vv_add

Given real vectors $v$ and $u$, this subroutine computes $v + u$. The result overwrites $v$.

**Syntax**

**vv_add** $(v, u, n, p)$

**Arguments**

| | |
|---|---|
| $v$ | A real array of size $n$. |
| $u$ | A real array of size $n$. |
| $n$ | An integer. An array dimension. |
| $p$ | An integer. The number of processors used. |

**Notes**

Arrays $v$ and $u$ are not distributed.

## 3.25 vv_asg

Given a real vector $u$, this subroutine sets $v$ to $u$.

**Syntax**

**vv_asg** $(v, u, n, p)$

**Arguments**

| | |
|---|---|
| $v$ | A real array of size $n$. |
| $u$ | A real array of size $n$. |
| $n$ | An integer. An array dimension. |
| $p$ | An integer. The number of processors used. |

**Notes**

Arrays $u$ and $v$ are not distributed.

## 3.24 mt_cpm

Given a real matrix $A$ and a permutation vector $ipvt$, this subroutine permutes columns of $A$ according to the value of $ipvt$. The result overwrites $A$.

**Syntax**

**mt_cpm** $(A, ipvt, f, m, n, p)$

**Arguments**

| | |
|---|---|
| $A$ | A 2-dimensional $m \times n$ real array. |
| $ipvt$ | An integer array of size $n$. |
| $f$ | An integer. If $f = 1$, forward permutation, else backward. |
| $m$ | An integer. An array dimension. |
| $n$ | An integer. An array dimension. |
| $p$ | An integer. The number of processors used. |

**Notes**

Array $A$ is distributed over processors in the row wrap fashion.

Each processor must have a copy of entire vector $ipvt$.

## 3.23 mt_rpm

Given a real matrix $A$ and a permutation vector $ipvt$, this subroutine permutes rows of $A$ according to the value of $ipvt$. The result overwrites $A$.

**Syntax**

**mt_rpm** $(A, ipvt, f, m, n, p)$

**Arguments**

| | |
|---|---|
| $A$ | A 2-dimensional $m \times n$ real array. |
| $ipvt$ | An integer array of size $m$. |
| $f$ | An integer. If $f = 1$, forward permutation, else backward. |
| $m$ | An integer. An array dimension. |
| $n$ | An integer. An array dimension. |
| $p$ | An integer. The number of processors used. |

**Notes**

Array $A$ is distributed over processors in the row wrap fashion.

Each processor must have a copy of entire vector $ipvt$.

## 3.22 mt_trp

Given a real matrix $A$, this subroutine computes the transpose of $A$. The result is sent to $B$.

**Syntax**

**mt_trp** $(A, B, m, n, p)$

**Arguments**

| | |
|---|---|
| $A$ | A 2-dimensional $m \times n$ real array. |
| $B$ | A 2-dimensional $n \times m$ real array. |
| $m$ | An integer. An array dimension. |
| $n$ | An integer. An array dimension. |
| $p$ | An integer. The number of processors used. |

**Notes**

Arrays $A$ and $B$ are distributed over processors in the row wrap fashion.

$A$ and $B$ must be distinct arrays.

## 3.21 fb_nom

Given a real matrix $A$, this subroutine returns the Frobenius norm of $A$.

**Syntax**

$fnorm \leftarrow$ **fb_nom** $(A, m, n, p)$

**Arguments**

| | |
|---|---|
| $A$ | A 2-dimensional $m \times n$ real array. |
| $m$ | An integer. An array dimension. |
| $n$ | An integer. An array dimension. |
| $p$ | An integer. The number of processors used. |

**Notes**

Array $A$ is distributed over processors in the row wrap fashion.

## 3.20 um_asg

Given a real matrix $B$, this subroutine sets $A$ to the upper triangular part of $B$.

**Syntax**

**um_asg** $(A, B, n, p)$

**Arguments**

| | |
|---|---|
| $A$ | A 2-dimensional $n \times n$ real array. |
| $B$ | A 2-dimensional $n \times n$ real array. |
| $n$ | An integer. An array dimension. |
| $p$ | An integer. The number of processors used. |

**Notes**

Array $A$ and $B$ are distributed over processors in the row wrap fashion.

## 3.19 lm_asg

Given a real matrix $B$, this subroutine sets $A$ to the lower triangular part of $B$.

**Syntax**

**lm_asg** $(A, B, n, p)$

**Arguments**

| | |
|---|---|
| $A$ | A 2-dimensional $n \times n$ real array. |
| $B$ | A 2-dimensional $n \times n$ real array. |
| $n$ | An integer. An array dimension. |
| $p$ | An integer. The number of processors used. |

**Notes**

Array $A$ and $B$ are distributed over processors in the row wrap fashion.

## 3.18 dm_asg

Given a real vector $d$, this subroutine sets $A$ to a diagonal matrix with $d$ as its diagonal vector.

**Syntax**

**dm_asg** $(A, d, n, p)$

**Arguments**

| | |
|---|---|
| $A$ | A 2-dimensional $n \times n$ real array. |
| $d$ | A real array of size $n$. |
| $n$ | An integer. An array dimension. |
| $p$ | An integer. The number of processors used. |

**Notes**

Array $A$ is distributed over processors in the row wrap fashion.

Each processor must have a copy of entire vector $d$.

## 3.17 ms_prd

Given a real matrix $A$ and a real scalar $s$, this subroutine computes $sA$. The result overwrites $A$.

**Syntax**

**ms_prd** $(A, s, m, n, p)$

**Arguments**

| | |
|---|---|
| $A$ | A 2-dimensional $m \times n$ real array. |
| $s$ | A real scalar. |
| $m$ | An integer. An array dimension. |
| $n$ | An integer. An array dimension. |
| $p$ | An integer. The number of processors used. |

**Notes**

Array $A$ is distributed over processors in the row wrap fashion.

## 3.16 ms_add

Given a real matrix $A$ and a real scalar $s$, this subroutine adds $s$ to all elements of $A$.

**Syntax**

**ms_add** $(A, s, m, n, p)$

**Arguments**

| | |
|---|---|
| $A$ | A 2-dimensional $m \times n$ real array. |
| $s$ | A real scalar. |
| $m$ | An integer. An array dimension. |
| $n$ | An integer. An array dimension. |
| $p$ | An integer. The number of processors used. |

**Notes**

Array $A$ is distributed over processors in the row wrap fashion.

## 3.15 ms_asg

Given a real matrix $A$ and a real scalar $s$, this subroutine sets all elements of $A$ to $s$.

**Syntax**

**ms_asg** $(A, s, m, n, p)$

**Arguments**

| | |
|---|---|
| $A$ | A 2-dimensional $m \times n$ real array. |
| $s$ | A real scalar. |
| $m$ | An integer. An array dimension. |
| $n$ | An integer. An array dimension. |
| $p$ | An integer. The number of processors used. |

**Notes**

Array $A$ is distributed over processors in the row wrap fashion.

## 3.14 vm_prd

Given a real matrix $A$ and a real vector $b$, this subroutine computes $b^T A$. Upon completion, the result of $b^T A$ is sent to $c^T$.

**Syntax**

**vm_prd** $(b, A, c, m, n, p)$

**Arguments**

| | |
|---|---|
| $A$ | A 2-dimensional $m \times n$ real array. |
| $b$ | A real array of size $m$. |
| $c$ | A real array of size $n$. |
| $m$ | An integer. An array dimension. |
| $n$ | An integer. An array dimension. |
| $p$ | An integer. The number of processors used. |

**Notes**

Array $A$ is distributed over processors in the row wrap fashion.

Each processor must have a copy of entire vector $b$.

Each processor obtains a copy of entire vector $c$.

## 3.13 mv_prd

Given a real matrix $A$ and a real vector $b$, this subroutine computes $Ab$. Upon completion, the result of $Ab$ is sent to $c$.

**Syntax**

**mv_prd** $(A, b, c, m, n, p)$

**Arguments**

| | |
|---|---|
| $A$ | A 2-dimensional $m \times n$ real array. |
| $b$ | A real array of size $n$. |
| $c$ | A real array of size $n$. |
| $m$ | An integer. An array dimension. |
| $n$ | An integer. An array dimension. |
| $p$ | An integer. The number of processors used. |

**Notes**

Array $A$ is distributed over processors in the row wrap fashion.

Each processor must have a copy of entire vector $b$.

Each processor obtains a copy of entire vector $c$.

## 3.12 mo_prd

Given real matrices $A$ and $B$, this subroutine computes $AB^T$. Upon completion, the result of $AB^T$ is sent to $C$.

**Syntax**

**mo_prd** $(A, B, C, m, k, n, p)$

**Arguments**

| | |
|---|---|
| $A$ | A 2-dimensional $m \times k$ real array. |
| $B$ | A 2-dimensional $n \times k$ real array. |
| $C$ | A 2-dimensional $m \times n$ real array. |
| $m$ | An integer. An array dimension. |
| $k$ | An integer. An array dimension. |
| $n$ | An integer. An array dimension. |
| $p$ | An integer. The number of processors used. |

**Notes**

Arrays $A$, $B$ and $C$ are distributed over processors in the row wrap fashion.

The array for $B$ or $C$ must be distinct from that for $A$.

## 3.11 mm_prd

Given real matrices $A$ and $B$, this subroutine computes $AB$. Upon completion, the result of $AB$ is sent to $C$.

**Syntax**

**mm_prd** $(A, B, C, m, k, n, p)$

**Arguments**

| | |
|---|---|
| $A$ | A 2-dimensional $m \times k$ real array. |
| $B$ | A 2-dimensional $k \times n$ real array. |
| $C$ | A 2-dimensional $m \times n$ real array. |
| $m$ | An integer. An array dimension. |
| $k$ | An integer. An array dimension. |
| $n$ | An integer. An array dimension. |
| $p$ | An integer. The number of processors used. |

**Notes**

Arrays $A$, $B$ and $C$ are distributed over processors in the row wrap fashion.

The array for $B$ or $C$ must be distinct from that for $A$.

## 3.10 mm_sbt

Given real matrices $A$ and $B$, this subroutine computes $A - B$. Upon completion, $A$ is overwritten by $A - B$.

**Syntax**

**mm_sbt** $(A, B, m, n, p)$

**Arguments**

| | |
|---|---|
| $A$ | A 2-dimensional $m \times n$ real array. |
| $B$ | A 2-dimensional $m \times n$ real array. |
| $m$ | An integer. An array dimension. |
| $n$ | An integer. An array dimension. |
| $p$ | An integer. The number of processors used. |

**Notes**

Arrays $A$ and $B$ are distributed over processors in the row wrap fashion.

## 3.9 mm_add

Given real matrices $A$ and $B$, this subroutine computes $A + B$. Upon completion, $A$ is overwritten by $A + B$.

**Syntax**

**mm_add** $(A, B, m, n, p)$

**Arguments**

| | |
|---|---|
| $A$ | A 2-dimensional $m \times n$ real array. |
| $B$ | A 2-dimensional $m \times n$ real array. |
| $m$ | An integer. An array dimension. |
| $n$ | An integer. An array dimension. |
| $p$ | An integer. The number of processors used. |

**Notes**

Arrays $A$ and $B$ are distributed over processors in the row wrap fashion.

## 3.8 mm_asg

Given real matrices $A$ and $B$, this subroutine sets $A$ to $B$.

**Syntax**

**mm_asg** $(A, B, m, n, p)$

**Arguments**

| | |
|---|---|
| $A$ | A 2-dimensional $m \times n$ real array. |
| $B$ | A 2-dimensional $m \times n$ real array. |
| $m$ | An integer. An array dimension. |
| $n$ | An integer. An array dimension. |
| $p$ | An integer. The number of processors used. |

**Notes**

Arrays $A$ and $B$ are distributed over processors in the row wrap fashion.

## 3.7 lf_utr

Given a real matrix $U$, this subroutine computes the solution of the upper triangular system $x^T U = b^T$. Upon completion, $b$ is overwritten by $x$.

**Syntax**

**lf_utr** $(U, b, n, p)$

**Arguments**

| | |
|---|---|
| $U$ | A 2-dimensional $n \times n$ real array. Contains the upper triangular matrix. |
| $b$ | A real array of size $n$. Upon completion, the array is overwritten by $x$. |
| $n$ | An integer. The array dimension. |
| $p$ | An integer. The number of processors used. |

**Notes**

Array $U$ is distributed over processors in the row wrap fashion.

Vector $b$ must be stored initially on the processor containing the first row of $U$.

Upon completion, each processor obtains a copy of entire vector $x$.

## 3.6 lf_ltr

Given a real matrix $L$, this subroutine computes the solution of the lower triangular system $x^T L = b^T$. Upon completion, $b$ is overwritten by $x$.

**Syntax**

**lf_ltr** $(L, b, n, p)$

**Arguments**

| | |
|---|---|
| $L$ | A 2-dimensional $n \times n$ real array. Contains the lower triangular matrix. |
| $b$ | A real array of size $n$. Upon completion, the array is overwritten by $x$. |
| $n$ | An integer. The array dimension. |
| $p$ | An integer. The number of processors used. |

**Notes**

Array $L$ is distributed over processors in the row wrap fashion.

Vector $b$ must be stored initially on the processor containing the last row of $L$.

Upon completion, each processor obtains a copy of entire vector $x$.

## 3.5 rt_utr

Given a real matrix $U$, this subroutine computes the solution of the upper triangular system $UX = B$ with the right hand side $B$. Upon completion, $B$ is overwritten by $X$.

**Syntax**

**rt_utr** $(U, B, n, r, p)$

**Arguments**

| | |
|---|---|
| $U$ | A 2-dimensional $n \times n$ real array. Contains the upper triangular matrix. |
| $B$ | A 2-dimensional $n \times r$ real array. Upon completion, the array is overwritten by $X$. |
| $n$ | An integer. An array dimension. |
| $r$ | An integer. An array dimension. |
| $p$ | An integer. The number of processors used. |

**Notes**

Arrays $U$ and $B$ are distributed over processors in the row wrap fashion.

$U$ and $B$ must distinct arrays.

## 3.4 rt_ltr

Given a real matrix $L$, this subroutine computes the solution of the lower triangular system $LX = B$ with the right hand side $B$. Upon completion, $B$ is overwritten by $X$.

**Syntax**

**rt_ltr** $(L, B, n, r, p)$

**Arguments**

| | |
|---|---|
| $L$ | A 2-dimensional $n \times n$ real array. Contains the lower triangular matrix. |
| $B$ | A 2-dimensional $n \times r$ real array. Upon completion, the array is overwritten by $X$. |
| $n$ | An integer. An array dimension. |
| $r$ | An integer. An array dimension. |
| $p$ | An integer. The number of processors used. |

**Notes**

Arrays $L$ and $B$ are distributed over processors in the row wrap fashion.

$L$ and $B$ must be distinct arrays.

## 3.3 lq_fac

Given a real matrix $A$, this subroutine computes an lower triangular matrix $L$ and an orthogonal matrix $Q$ such that $A = LQ$. Upon completion, the lower triangular part of $A$ is overwritten by $L$. Given a real matrix $B$ of proper dimensions, this subroutine also computes $BQ^T$, which overwrites $B$ upon completion.

**Syntax**

**lq_fac** $(A, B, m, n, r, p)$

**Arguments**

| | |
|---|---|
| $A$ | A 2-dimensional $m \times n$ real array ($m \leq n$). Contains the matrix to be factored. Upon completion, the lower triangular part of $A$ is overwritten by $L$. |
| $B$ | A 2-dimensional $r \times n$ real array. Upon completion, the array is overwritten by $BQ^T$. |
| $m$ | An integer. An array dimension. |
| $n$ | An integer. An array dimension. |
| $r$ | An integer. An array dimension. |
| $p$ | An integer. The number of processors used. |

**Notes**

Arrays $A$ and $B$ are distributed over processors in the row wrap fashion.

$A$ and $B$ must be distinct arrays.

## 3.2 lu_fac

Given a real matrix $A$, this subroutine computes the LU factorization of $A$ with partial column pivoting. Upon completion, the upper triangular part of $A$ is overwritten by $U$, and the lower triangular part by $L$.

**Syntax**

**lu_fac** $(A, ipvt, n, p)$

**Arguments**

$A$ A 2-dimensional $n \times n$ real array. Contains the matrix to be factored. Upon completion, the upper triangular part of $A$ is overwritten by $U$, and the lower triangular part by $L$.

*ipvt* An integer array of size $n$. The permutation vector for the column pivoting.

$n$ An integer. The array dimension.

$p$ An integer. The number of processors used.

**Notes**

Array $A$ is distributed over processors in the row wrap fashion.

Each processor has a copy of entire array *ipvt*.

## 3.1 ch_fac

Given a real matrix $A$, this subroutine computes an upper triangular matrix $R$ such that $A = R^T R$. Upon completion, the upper triangular part of $A$ is overwritten by $R$.

**Syntax**

**ch_fac** $(A, n, p)$

**Arguments**

| | |
|---|---|
| $A$ | A 2-dimensional $n \times n$ real array. Contains the matrix to be factored. Upon completion, the upper triangular part of $A$ is overwritten by the upper triangular factor. |
| $n$ | An integer. The array dimension. |
| $p$ | An integer. The number of processors used. |

**Notes**

Array $A$ is distributed over processors in the row wrap fashion.

- **vs** — vector scalar.

**SUFFIX:**

- **fac** — factorization.
- **ltr** — lower triangular.
- **utr** — upper triangular.
- **asg** — assignment.
- **add** — addition.
- **sbt** — subtraction.
- **prd** — product.
- **trp** — transposition.
- **rpm** — row permutation.
- **cpm** — column permutation.
- **nom** — norm.

# Abbreviations

**SUBROUTINE NAME ::= PREFIX_SUFFIX**

**PREFIX:**

- **ch** — Cholesky.
- **fb** — Frobenius.
- **lu** — LU.
- **lq** — LQ.
- **rt** — right.
- **lf** — left.
- **mt** — matrix.
- **dm** — diagonal matrix.
- **lm** — lower triangular matrix.
- **um** — upper triangular matrix.
- **mm** — matrix matrix.
- **mv** — matrix vector.
- **vm** — vector matrix.
- **ms** — matrix scalar.
- **vv** — vector vector.

# Chapter 3

# Subroutine Descriptions

```
/* Send partial solution to host.                                    */

   send0(&b[0],np*sizeof(double),myid,host);

/* Free allocated memory.                                            */

   free_m1(A,np);
   free_m1(B,np);
   free_v1(b);

/* Disable PICL communication.                                       */

   close0();

   }
```

```
   int i,i1,j,p,np,nproc,myid,host,bytes;
   double *A[n],*B[n],*b;

   open0(&p,&myid,&host);     /* Enable PICL communication.  */

   np=n/p;
   if (myid<n%p) np++;        /* # of local rows.            */

/* Allocate memory for A, B, and b.                          */

   malloc_m1(A,np,n);
   malloc_m1(B,np,1);
   malloc_v1(&b,n);

   bytes=n*sizeof(double);

/* Receive A from host.                                      */

   for (i=0;i<np;i++)
       recv0(&A[i][0],bytes,i);

/* Node 0 receives b from host.                              */

   if (myid==0)
      recv0(&b[0],bytes,n);

/* Solve Ax=b.                                               */

   ch_fac(A,n,p);                  /* Obtain R st. A=R^{T}R. */

   lf_utr(A,b,n,p);                /* Solve R^{T}y=b.        */

   for (i=0;i<np;i++) B[i][0]=b[i*p+myid];

   rt_utr(A,B,n,1,p);              /* Solve Rx=y.            */

   for (i=0;i<np;i++) b[i]=B[i][0];
```

## 2.6 An Example Node Program

```
/*                                                              */
/*   File Name:                                                 */
/*                                                              */
/*         node.c                                               */
/*                                                              */
/*   Function:                                                  */
/*                                                              */
/*         Node program for solving Ax=b,                       */
/*         where A is symmetric positive definite.              */
/*                                                              */
/*   Subroutines Used:                                          */
/*                                                              */
/*         open0  ---- PICL                                     */
/*         close0 ---- PICL                                     */
/*         send0  ---- PICL                                     */
/*         recv0  ---- PICL                                     */
/*                                                              */
/*         ch_fac ---- PRDLA                                    */
/*         rt_utr ---- PRDLA                                    */
/*         lf_utr ---- PRDLA                                    */
/*                                                              */
/*         malloc_m1 ---- PRDLA [tl_etc.c]                      */
/*         malloc_v1 ---- PRDLA [tl_etc.c]                      */
/*         free_m1   ---- PRDLA [tl_etc.c]                      */
/*         free_v1   ---- PRDLA [tl_etc.c]                      */
/*                                                              */

#include <stdio.h>
#include <cube.h>
#define n 400

main()
  {
```

```
        if (i<nmp) np++;

        recv0(buf,np*sizeof(double),i);    /* From node i.    */

        for (j=0;j<np;j++)
            data[j*p+i]=buf[j];    /* Transform wrap mapping. */

        }

/* Output solution to the datafile.                           */

   fs=fopen("x.data","w");

   for (j=0;j<n;j++)
       fprintf(fs,"%+e \n",data[j]);

   fclose(fs);

/* Release nodes and disable PICL communication.              */

   close0(1);

   }
```

```
   load0("nodeprogram",-1);     /* Load node program.  */

   bytes=n*sizeof(double);

/* Read A from the datafile and distribute it to nodes.      */

   fs=fopen("A.data","r");

   for (i=0;i<n;i++) {

       for (j=0;j<n;j++)
           fscanf(fs,"%lf",&data[j]);  /* Read ith row of A. */

       send0(data,bytes,i/p,i%p);      /* Send to node i%p.  */

       }

   fclose(fs);

/* Read b from the datafile and send it to node 0.           */

   fs=fopen("b.data","r");

   for (j=0;j<n;j++)
       fscanf(fs,"%lf",&data[j]);         /* Read b.         */

   send0(data,bytes,n,0);                 /* Send to node 0. */

   fclose(fs);

/* Wait for the solution from nodes.                         */

   np=n/p;
   nmp=n%p;

   for (i=0;i<p;i++) {
```

## 2.5 An Example Host Program

```
/*                                                              */
/*   File Name:                                                 */
/*                                                              */
/*         host.c                                               */
/*                                                              */
/*   Function:                                                  */
/*                                                              */
/*         Host program for solving Ax=b,                       */
/*         where A is symmetric positive definite.              */
/*                                                              */
/*   Subroutines Used:                                          */
/*                                                              */
/*         open0  ---- PICL                                     */
/*         load0  ---- PICL                                     */
/*         close0 ---- PICL                                     */
/*         send0  ---- PICL                                     */
/*         recv0  ---- PICL                                     */
/*                                                              */
/*                                                              */

#include <stdio.h>
#include <cube.h>
#define n 400

main()
  {
   FILE *fs;
   int i,j,p,np,nmp,myid,host,bytes;
   double data[n],buf[n];

   p=16;
   open0(&p,&myid,&host);       /* Allocate 16 nodes.  */
```

where **hostlib.a** is the PICL library for the host program. The node program needs to be compiled with PRDLA subroutine files, the PICL library, as well as the iPSC/860 math library (for the case that PRDLA1 is used). For the node program, the iPSC/860 C compiler is required:

```
icc -o nodeprogram node.c ch_fac.c rt_utr.c lf_utr.c tl_etc.c
    nodelib.a -lkmath -lm -node
```

where **nodelib.a** is the PICL library for the node program, and **kmath** is the iPSC/860 math library. The PRDLA file **tl_etc.c** also is included since it contains subroutines necessary for **ch_fac**, **rt_utr** and **lf_utr**.

Using PICL functions, the work to allocate, load and release nodes can also be done in the host program. So, to run the program on the cube, we only need to run the executable **hostprogram** obtained after compiling the host program. The node program **nodeprogram** will be loaded onto the cube by the **hostprogram**.

where $p$ is the number of node processors used.

## 2.3 Data Access

To use PRDLA subroutines for a special application, the input data must be arranged appropriately. For example, for the linear system in last section, matrix $A$ and vector $b$ usually are stored in a data file on the front end, or generated somehow by programs. In any case, matrix $A$ must be distributed over node processors according to the row wrap fashion, i.e., if $p$ processors are used, submatrix $A(i : p : n-1, 0 : n-1)$ needs to be stored on node processor $i$ for all $i = 0, \ldots, p-1$. Vector $b$, for the particular case, needs to be send to node processor 0 as required by the subroutine **lf_utr**.

Suppose that matrix $A$ and vector $b$ are stored previously in a data file on the front end. Then, a host program needs to be written to read matrix $A$ and vector $b$ from the data file, distribute rows of $A$ over node processors, send $b$ to node processor 0, and receive results from node processors, etc. The node program is responsible to read corresponding rows of $A$ and vector $b$ from the host, call PRDLA subroutines to solve the linear system, and then send the solution $x$ back to the host.

## 2.4 Compiling and Linking

The host and node programs can be edited on either the front end or other workstations. Suppose that a host program **host.c** and a node program **node.c** are written for the linear system discussed in last sections. Then the host program can be compiled using the standard C compiler as follows:

```
cc -o hostprogram host.c hostlib.a -host
```

## 2.1 Programming with PRDLA

Typically, to implement an application on the hypercube, two seperate programs need to be written, a host program and a node program. The host program is run on the front end, serving as an interface between users and the cube. The node program is run on the cube, and contains major work for the application. PRDLA subroutines are used for the node program.

Programs with PRDLA subroutines must be compiled along with PICL because all communication functions used in PRDLA are standard PICL functions. If PRDLA1 is to be used, the iPSC/860 math library must also be linked with the programs.

## 2.2 An Example Application

PRDLA subroutines can be used for various high level applications. For example, suppose that a linear system is given as follows:

$$Ax = b$$

where $A$ is an $n$ by $n$ real symmetric positive definite matrix, and $b$ is an $n$ dimensional vector. The system can be solved with Cholesky factorization followed by two triangular system solves, which can be done by making a sequence of PRDLA subroutine calls:

**ch_fac** $(A, n, p)$ { Obtain $R$ such that $A = R^T R$. }
**lf_utr** $(A, b, n, p)$ { Solve $R^T y = b$. }
**rt_utr** $(A, b, n, 1, p)$ { Solve $Rx = y$. }

# Chapter 2

# Using the System

- G. A. Geist and C. H. Romine [1988]. *LU Factorization Algorithms on Distributed Memory Multiprocessor Architectures.* SIAM Journal on Scientific and Statistical Computing 9, 639-649.

- G. H. Golub and C. F. Van Loan [1989]. *Matrix Computations.* The Johns Hopkins Univ. Press, Baltimore, MD.

- G. Li and T. Coleman [1988]. *A Parallel Triangular Solver for a Distributed-Memory Multiprocessor.* SIAM Journal on Scientific and Statistical Computing 9, 485-502.

- G. Li and T. Coleman [1989]. *A New Method for Solving Triangular Systems on Distributed-Memory Message-Passing Multiprocessors.* SIAM Journal on Scientific and Statistical Computing 10, 382-396.

- The Intel Corporation [1991]. *The iPSC/860 Basic Math Library Users' Guide.*

- The Intel Corporation [1991]. *The iPSC/2 and iPSC/860 Users' Guide.*

Instrumented Communication Library, developed at the Oak Ridge National Laboratory. Subroutines in PRDLA0 are all written in C without using any machine specific subroutines. In PRDLA1, the system provided level-1 BLAS subroutines are used. Therefore, the performance of PRDLA1 is much higher. The average double precision megaflop rate is over 100 mflops for matrices of sizes around 2000 by 2000.

PRDLA is not totally user transparent. As a matter of fact, users are encouraged to know about the implementation of all subroutines. So, the system is not documented as a library. Rather, it is left as a collection of independent subroutines. Users are free to look at any subroutine, to include it in their own programs, or to modify it for a special usage. With this practice, using the system is also a learning or training process for parallel and distributed programming.

## 1.5 Online Performance Test

Online performance test programs are available for both PRDLA0 and PRDLA1. A group of subroutines involving message passing can be tested easily using the performance test programs. The performance results include the running time, megaflop rate, and error estimate. The subroutines are tested for randomly generated matrices of given sizes with a given number of processors.

## 1.6 Further Reading


- G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley [1991] *A Users' Guide to PICL: A Portable Instrumented Communication Library.* ORNL/TM-11616, Oak Ridge National Laboratory, Oak Ridge, TN.

- **Special Matrix Operation:** matrix transposition, matrix permutation, and diagonal/triangular matrix generation.

- **Vector-Vector Operation:** vector-vector assignment, addition, subtraction and product.

- **Vector-Scalar Operation:** vector-scalar assignment, addition and product.

## 1.3 Data Mapping and Message Passing

The row distribution scheme is used for the data mapping in PRDLA. That is, all matrices, either input or output to a subroutine, are stored over processors row by row in wrap fashion. With this type of data mapping, simple algorithms can be implemented. However, for some computations, the performance may be far from optimal: i.e., when the dimension is very large. In that case, a mixed row and column distributed system might be necessary. Such a system is not available yet.

In general, most subroutines in PRDLA assume a ring of processors with message passing conducted only between neighboring processors. The pipeline message passing technique is used to overlap the communication overhead with computation.

## 1.4 Software Support

Currently, there are two versions of PRDLA, namely, PRDLA0 and PRDLA1. Both versions use standard communication subroutines of PICL, the Portable

## 1.1 PRDLA

PRDLA is a small software system written in C and developed on the Intel iPSC/860 hypercube. The system contains a group of independent subroutines for distributed matrix computation, which cover a subset of basic matrix computation functions such as matrix factorization, triangular system solve, matrix-matrix product, matrix transposition, and so on. The system is intended to be used for both research and teaching for basic parallel and distributed computation.

## 1.2 The System Functions

The subroutines can be divided into the following groups in terms of their functions:

- **Matrix Factorization:** Cholesky, LU and LQ factorizations.
- **Triangular System Solution:** lower/upper triangular right/left system solves.
- **Matrix-Matrix Operation:** matrix-matrix assignment, addition, subtraction and product.
- **Matrix-Vector Operation:** matrix-vector and vector-matrix product.
- **Matrix-Scalar Operation:** matrix-scalar assignment, addition and product.

# Chapter 1

# Overview

# Preface

The purpose of this system is to provide an easy-to-use set of basic parallel matrix manipulation subroutines, in C, for use on the Intel iPSC/860 hypercube. Most of the subroutines exhibit good performance, but not necessarily optimal. Users are welcome to use these subroutines as building blocks in their own codes, to modify if the situation warrants this, or just to study the implementations to help guide their own programming.

The subroutines all operate under the assumption that matrices are distributed by row in a natural wrap mapping. That is, if we assume $p$ processors, row 1 is mapped to processor 0, row 2 is mapped to processor 1, row $p$ is mapped to processor $p-1$, row $p+1$ is mapped to processor 0, etc. If this is deemed too restrictive, the user is welcome to use these codes as a starting point and develop their own codes (perhaps under less restrictive assumptions).

The codes are not directly portable to other computer systems. However, the codes do use PICL (the portable instrumented communication library, developed at Oak Ridge National Laboratory); therefore, provided PICL is available the codes are easily ported.

# List of Figures

# Contents

******

# A PARALLEL ROW DISTRIBUTED LINEAR ALGEBRA SYSTEM

*PRDLA USERS' GUIDE*

**Zhijun Wu and Thomas Coleman**

**February, 1992**

**Advanced Computing Research Institute**

**Cornell University**

******