High Performance Linear Algebra II

M. D. Jones, Ph.D.

Center for Computational Research
University at Buffalo
State University of New York

High Performance Computing I, 2013

Dense & Parallel

In this topic we will still discuss **dense** (as opposed to **sparse**) matrices, but now focus on parallel execution.

- Best Performance (and scalability) most-often comes down to making best use of (local) L3 BLAS
- Optimized BLAS are generally crucial for achieving performance gains

BLAS Recapitulated

BLAS are critical:

- Foundation for all (computational) linear algebra
- Parallel versions still call serial versions on each processor
- Performance goes up with increasing ratio of floating point operations to memory references
- Well-tuned BLAS takes maximum advantage of memory hierarchy

BLAS Level	Calculation	Memory Refs.	Flop Count	Ratio(Flop/Mem)
1	DDOT	$3N$	$2N$	2/3
2	DGEMV	N^2	$2N^2$	2
3	DGEMM	$4N^2$	$2N^3$	N/2

Simple Memory Model

Let us assume just two levels of memory - slow and fast

$$
\begin{aligned}
m &= \#\text{ memory elements (words) moved between fast and slow memory,} \\
\tau_m &= \text{time per slow memory operation,} \\
f_{op} &= \text{number of floating-point operations,} \\
\tau_f &= \text{time per floating-point operation,}
\end{aligned}
$$

and $q = f_{op}/m$. Minimum time is just $f_{op} * \tau_f$, but more realistically:

$$\text{time} \simeq f_{op} * \tau_f + m * \tau_m = f_{op}\tau_f \left[1 + \tau_m/(q\tau_f)\right].$$

key observation - we need 'optimal reuse' of data = larger q (certainly $\tau_m \gg \tau_f$).

Simple Matrix Multiplication

Everyone has probably written this out at least once ...

```
for(i=0;i<N;i++) {
   for(j=0;j<N;j++) {
      C[i][j] = 0.0;
      for(k=0;k<N;k++) {
         C[i][j] += A[i][k]*B[k][j];
      }
   }
}
```

in order to multiply two matrices, **A** and **B**, and store the result in a third, **C**.

- Can use a temporary scalar to avoid dereferencing in innermost loop
- $\mathcal{O}(N^3)$ multiply-add operations

Partitioning

In this case, **partitioning** is also known as **block matrix multiplication**:

- Divide into s^2 submatrices, $N/s \times N/s$ elements in each submatrix
- Let $m = N/s$, and

```
for(p=0; p<s; p++) {
   for(q=0; q<s; q++) {
      C_{p,q} = 0.0;
      for(r=0; r<m; r++) {
         C_{p,q} += A_{p,r}*B{r,q};
      }
   }
}
```

where $A_{p,r}$ are themselves matrices.

Recursive/Divide & Conquer

Let N be a power of 2 (not strictly necessary) and divide **A** and **B** into 4 submatrices, delimited by:

$$\mathbf{A} = \begin{pmatrix} A_{pp} & A_{pq} \\ A_{qp} & A_{qq} \end{pmatrix}.$$

Then the solution requires 8 pairs of submatrix multiplications, which can be done recursively ...

```
matmultR(A, B, s) {
 if (s==1) {
    C = A*B; /* termination condition */
 }
 else {
   s = s/2;
   P0 = matmultR(A_{pp},B_{pp},s);
   P1 = matmultR(A_{pq},B_{qp},s);
   P2 = matmultR(A_{pp},B_{pq},s);
   P3 = matmultR(A_{pq},B_{qq},s);
   P4 = matmultR(A_{qp},B_{pp},s);
   P5 = matmultR(A_{qq},B_{qp},s);
   P6 = matmultR(A_{qp},B_{pq},s);
   P7 = matmultR(A_{qq},B_{qq},s);
   C_{pp} = P0+P1;
   C_{pq} = P2+P3;
   C_{qp} = P4+P5;
   C_{qq} = P6+P7;
 }
 return(C);
}
```

- Well suited for SMP with cache hierarchy
- Size of data continually reduced and localized
- Can be highly performing when making maximum reuse of data within cache memory hierarchy
- More generally we want to address cases for which matrix will not fit within memory of a single machine, message passing/distributed model is required

Cannon's Algorithm

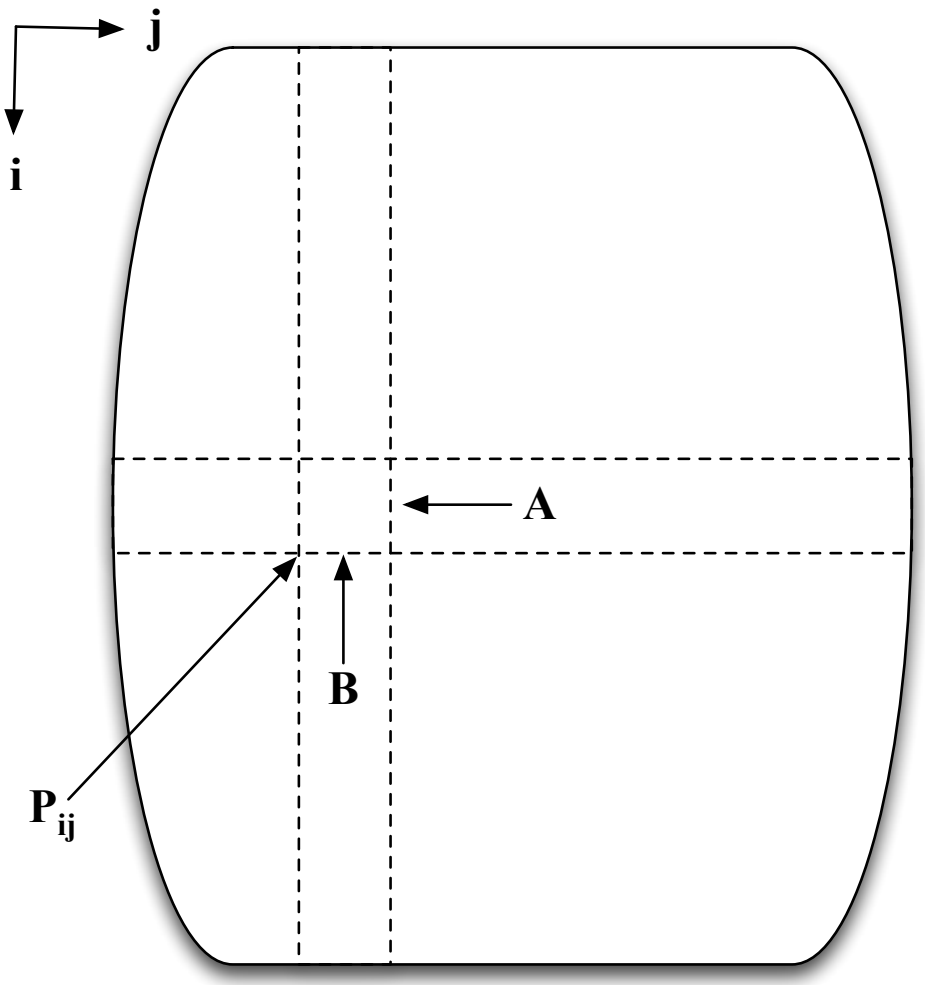
Cannon's algorithm:

- Uses a mesh of processors with a torus topology (periodic boundary conditions
- Elements are shifted to an "aligned" position: Row i of **A** is shifted i places to the left, while row j of **B** is shifted j spots upward. This puts $A_{i,j+i}$ and $B_{i+j,j}$ in processor $P_{i,j}$ (namely we have appropriate submatrices/elements to multiply in $P_{i,j}$)
- Usually submatrices are used, but we can use elements to simplify the notation a bit

After alignment, Cannon's algorithm proceeds as follows:

- Each process $P_{i,j}$ multiplies its elements
- Row i of **A** is shifted one place left, and column j of **B** is shifted one place up (brings together adjacent elements of **A** and **B** needed for summing results)
- Each process $P_{i,j}$ multiplies its elements and adds to accumulating sum
- Preceding two steps repeated until results obtained ($N - 1$ shifts)

Cannon Illustrated



Cannon's algorithm showing flow of data for process P_{ij}.

For s submatrices ($m = N/s$) the communication time for Cannon's algorithm is given by

$$\tau_{\text{comm}} = 4(s-1)(\tau_{\text{lat}} + m^2\tau_{\text{dat}}),$$

where τ_{lat} is the **latency** and τ_{dat} is the time required to send one value (word). The computation time in Cannon's algorithm:

$$\tau_{\text{comp}} = 2sm^3 = 2m^2N,$$

or $\mathcal{O}(m^2N)$.
Cannon's algorithm is also known as the **ScaLAPACK outer product** algorithm (can you see another numerical library coming?) ...

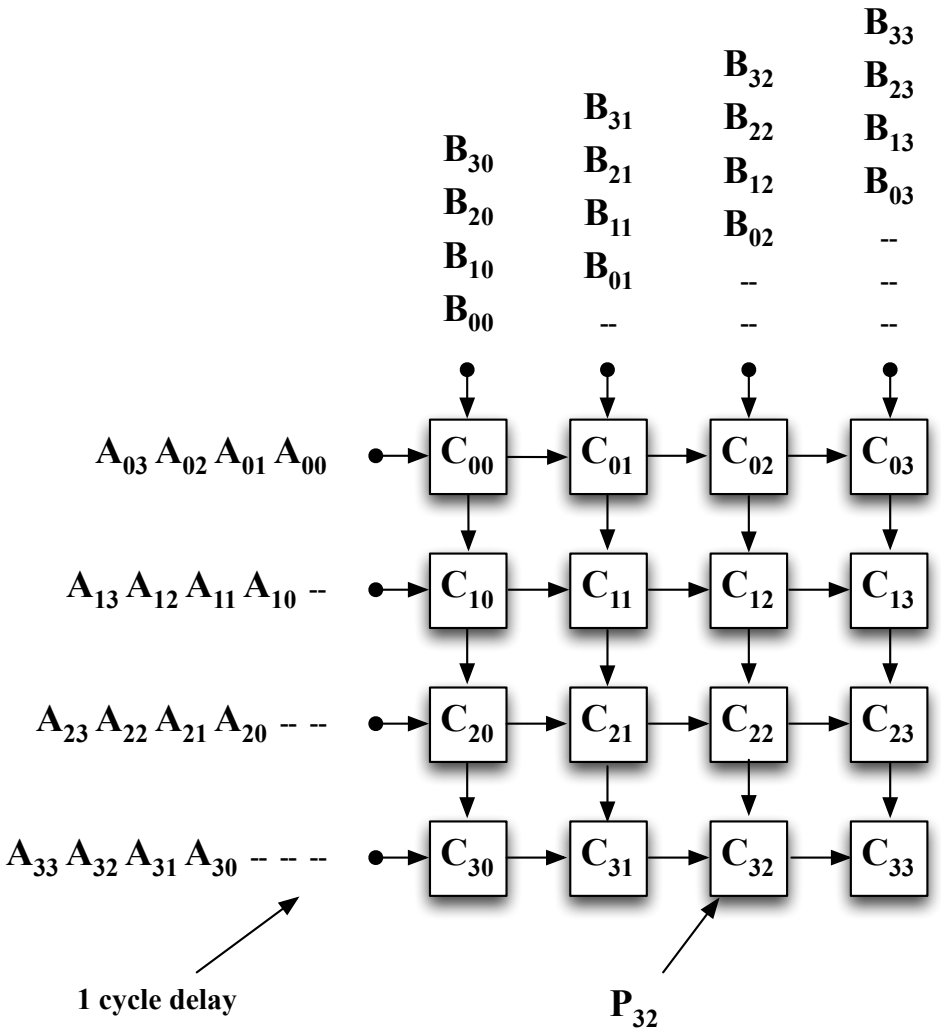
2D Pipeline

Another useful algorithm is the so-called **2D pipeline**, in which the data flows into a rectangular array of processors. Labeling the process grid using $(0, 0)$ as the top left corner, the data flows from the left and from the top, with process $P_{i,j}$:

1. `RECV`(A from $P_{i,j-1}$) { A data from left }
2. `RECV`(B from $P_{i-1,j}$) { B data from above }
3. Accumulate $C_{i,j}$
4. `SEND`(A to $P_{i,j+1}$) { A data to right }
5. `SEND`(B to $P_{i+1,j}$) { B data down }

(*illustrate with sketch*)

2D Pipeline Illustrated



2D pipeline (or systolic array) algorithm showing flow of data.

Strassen's Method

The techniques discussed thus far are fundamentally $\mathcal{O}(N^3)$, but there is a clever method due to Strassen[1] which is $\mathcal{O}(N^{2.81})$:

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \quad \mathbf{C} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

$$\begin{aligned} Q_1 &= (A_{11} + A_{22})(B_{11} + B_{22}), \\ Q_2 &= (A_{21} + A_{22})B_{11}, \\ Q_3 &= A_{11}(B_{12} - B_{22}), \\ Q_4 &= A_{22}(-B_{11} + B_{21}), \\ Q_5 &= (A_{11} + A_{12})B_{22}, \\ Q_6 &= (-A_{11} + A_{21})(B_{11} + B_{12}), \\ Q_7 &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

$$\begin{aligned} C_{11} &= Q_1 + Q_4 - Q_5 + Q_7 \\ C_{12} &= Q_3 + Q_5 \\ C_{21} &= Q_2 + Q_4 \\ C_{22} &= Q_1 + Q_3 - Q_2 + Q_6 \end{aligned}$$

Conventional matrix multiplication takes N^3 multiplies and $N^3 - N^2$ adds

[1] V. Strassen, "Gaussian Elimination Is Not Optimal," Numerische Mathematik, **13**, 353-356 (1969).

- If N is not a power of 2 can pad with zeros
- Reiterate until submatrices reduced to numbers or optimal matrix multiply size for particular processor (Strassen switches to standard matrix multiplication for small enough submatrices)
- Eliminates 1 matrix multiply, instead of $\mathcal{O}(N^{\log_2 8})$, $\mathcal{O}(N^{\log_2 7}) = N^{2.81}$
- Conventional matrix multiplication takes N^3 multiplies and $N^3 - N^2$ adds, Strassen is $7N^{\log_2 7} - 6N^2$
- Requires additional storage for intermediate matrices, can be less stable numerically ...

Sequential LU/Gaussian Elimination Code

LU code fragment:

```
for (k=1;k<N;k++) {
   for (i=k+1;i<=N;i++) {
       L(i,k) = A(i,k)/A(k,k)
   }
   for (j=k+1;j<=N;j++) {
      for (i=k+1;i<=N;i++) {
          A(i,j) = A(i,j) - L(i,k)*A(k,j)
      }
   }
}
```

Note that the inner loops (over i) have no dependencies, i.e. they can more easily be executed in parallel.

Parallel LU/Gaussian Elimination

One strategy for parallelizing the Gaussian eleimination part of LU decomposition is to do so over the middle loop (j) in the algorithm. Decomposing the columns, and using a message passing implementation, we might have something like the following:

```
do k=1,N-1
   if(``column k is mine'') then     ! column-wise decomposition
        do i=k+1,N
           L(i,k) = A(i,k)/A(k,k)
        end do
        BCAST( L(k+1:N,k),root=''owner of k'' )
   else
        RECV( L(k+1:N,k) )
   end if
   do j=k+1,N (``modulo I own column j'') ! column-wise decomposition
      do i=k+1,N
         A(i,j) = A(i,j) - L(i,k)*A(k,j)
      end do
   end do
end do
```

Parallel LU Efficiency

If we do a (not overly crude) analysis of the preceding parallel algorithm, for each column k we have:

- Broadcat $N - k$ values, let's say taking time c_b
- Compute $(N - k)^2$ multiply-adds, time c_{fma}
-

$$\tau_k \simeq c_b(N - k) + c_{fma}\frac{(N - k)^2}{N_p},$$

Summing over k we find:

$$\tau(N_p) \simeq \frac{c_b}{2}N^2 + \frac{c_{fma}}{3}\frac{N^3}{N_p}, \tag{1}$$

Parallel LU Speedup

Now, looking at the parallel speedup factor we have:

$$\begin{aligned} S(N_p) &= \tau/\tau(N_p) \\ &\simeq \frac{c_{fma}N^3/3}{c_b N^2/2 + c_{fma}N^3/(3N_p)} \\ &= \left(\frac{3}{2N}\frac{c_b}{c_{fma}} + \frac{1}{N_p}\right)^{-1}. \end{aligned}$$

- See the ratio of communication cost to computation? It never goes away
- Minimizing communication costs is again the key to improving the efficiency ...

Parallel LU Efficiency

The efficiency is given by:

$$\begin{aligned} \mathcal{E}(N_p) &= S(N_p)/N_p, \\ &\simeq \left(\frac{3N_p}{2N}\frac{c_b}{c_{fma}} + 1\right)^{-1}, \\ &= 1 - \frac{3N_p}{2N}\frac{c_b}{c_{fma}} \end{aligned}$$

- Note that this algorithm is **scalable** (in time) by our previous definition, for a given fixed ratio of N_p/N
- Note that it is not scalable in terms of memory, since the memory requirements grow steadily as $\mathcal{O}(N^2)$

Improving Parallel LU

The most obvious way to improve the efficiency of our parallel Gaussian elimination is to overlap computation with the necessary communication. Consider the revised algorithm ...

```
! 1st processor computes L(2:N,1) and BCAST ( L(2:N,1) )
do k=1,N-1
   if(``column k is not mine'') then          ! post RECV for multipliers
      RECV( L(k+1:N,k) from BCAST( next_L )
   end if
   if(``column k+1 is mine'') then            ! compute next set of multipliers
      do i=k+1,N                              !  and eliminate for column k+1
         A(i,k+1) = A(i,k+1) - L(i,k)*A(k,k+1)
         next_L(i,k+1) = A(i,k+1)/A(k+1,k+1)
      end do
      BCAST( next_L(k+2:N,k+1),root=''owner of k+1'' )
   end if
   do j=k+2,N (``modulo I own column j'')     ! perform eliminations
      do i=k+1,N
         A(i,j) = A(i,j) - L(i,k)*A(k,j)
      end do
   end do
end do
```

ScaLAPACK

Scalable **LAPACK** library (LAPACK is the general purpose Linear Algebra PACKage), contains LAPACK-style driver routines formulated for distributed memory parallel processing:

- An exceptional example of an MPI library:
  1. Portable - based on optimized low-level routines
  2. User friendly - parallel versions of the common LAPACK routines have similar syntax, names just prefixed with a 'P'.
  3. User of the library need never write parallel processing code - it's all under the covers.
  4. Efficient.
- Source code and documentation available from NETLIB:
  `http://www.netlib.org/scalapack`

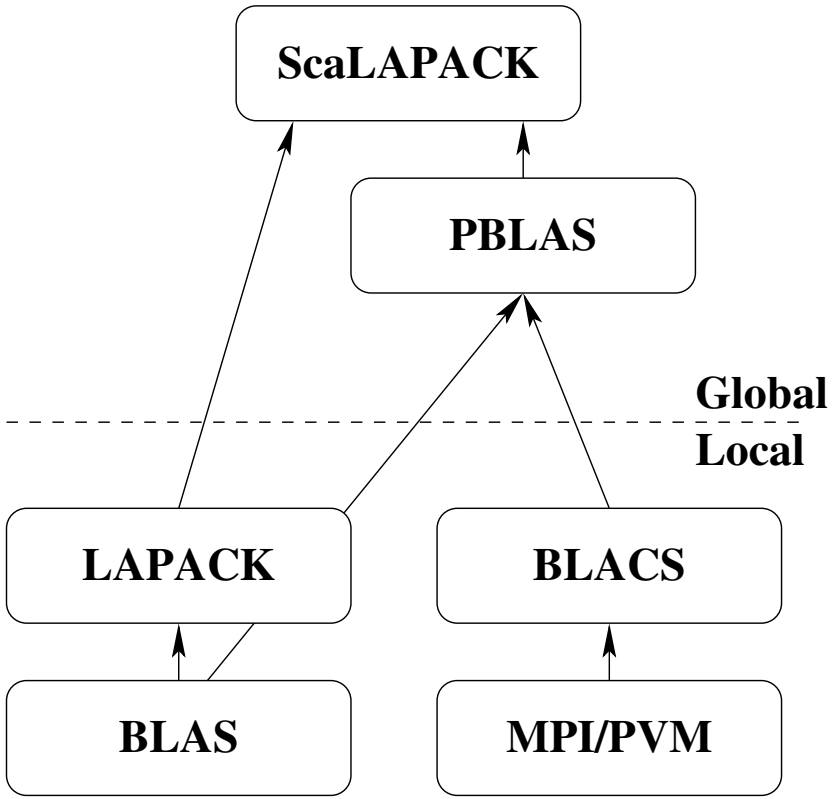
Under the ScaLAPACK Hood

Uses a **block-cyclic** decomposition of matrices, into $P_{row} \times P_{col}$ 2D representation. If we now consider a $NB \times NB$ sub-block (starting at index ib, ending at index end):

```
do ib=1,N,NB
   end=MIN(N,ib+NB-1)
   do i=ib,N
      (a) Find pivot row, k, BCAST column
      (b) Swap rows k and i in block column, BCAST row k
      (c) A(i+1:NB,i)=A(i+1:N,i)/A(i,i)
      (d) A(i+1:N,i+1:end)=A(i+1:N,i+1:end)-A(i+1:N,i)*A(i,i+1:end)
   end do
   (e) BCAST all swap information to right and left
   (f) Apply all row swaps to other columns
   (g) BCAST row k to right
   (h) A(ib:end,end+1:N)=LL/A(ib:end,end+1:N)
   (i) BCAST A(ib:end,end+1:N) down
   (j) BCAST A(end+1:N,ib:end) right
   (k) Eliminate A(end+1:N,end+1:N)
```

this is our old friend LU decomposition, available as the ScaLAPACK `PGETRF` routine.

ScaLAPACK Schematic



General schematic of the ScaLAPACK library.

ScaLAPACK Efficiency

Variable	Description
$C_f N^3$	Total Number FP Operations
$C_v N^2/\sqrt{N_p}$	Total Number Data Communicated
$C_m N/NB$	Total Number of Messages
τ_f	Time per FP Operation
τ_v	Time per Data Item Communicated
τ_m	Time per Message

With these quantities, the Time for the ScaLAPACK drivers is given by:

$$\tau(N, N_p) = \frac{C_f N^3}{N_p}\tau_f + \frac{C_v N^2}{\sqrt{N_p}}\tau_v + \frac{C_m N}{NB}\tau_m,$$

and the scaling:

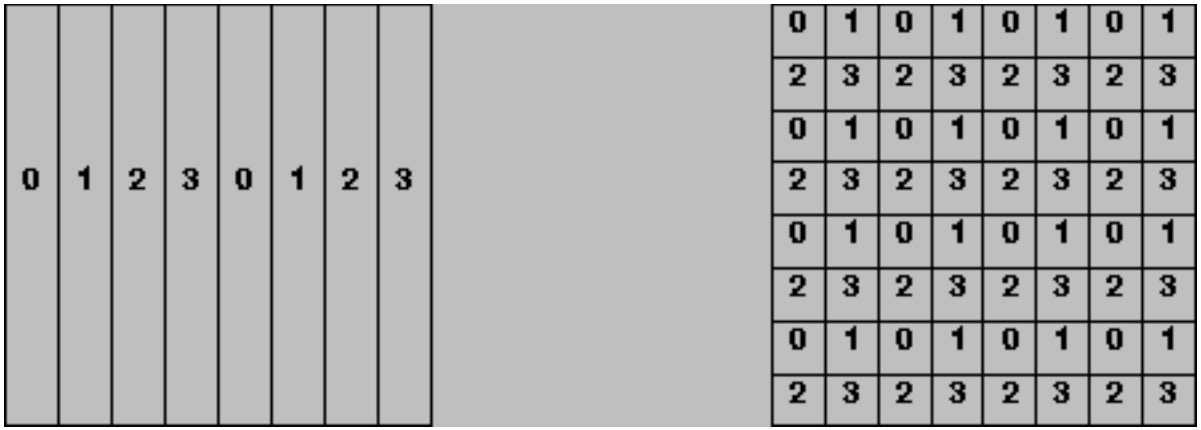
$$\mathcal{E}(N, N_p) \simeq \left(1 + \frac{1}{NB}\frac{C_m\tau_m}{C_f\tau_f}\frac{N_p}{N^2} + \frac{C_v\tau_v}{C_f\tau_f}\frac{\sqrt{N_p}}{N}\right)^{-1}.$$

Note that:

- Values of C_f, C_v, and C_m depend on driver routine (and underlying algorithm)
- Scalable for constant N^2/N_p ...
- **Small** problems, dominated by τ_m/τ_f, the ratio of latency to time per FP operation (see why latency matters?)
- **Medium** problems, significantly impacted by ratio of network bandwidth to FP operation rate, τ_f/τ_v
- **Large** problems, node Flop/s ($1/\tau_f$) dominates

Block-Cyclic Decomposition

ScaLAPACK uses a block-cyclic decomposition:



On the left, a 4 processor column-wise decomposition, to the right the block-cyclic version using $P_r = P_q = 2$, each square is *NB*x*NB* (2x2 in this example). The advantage of block-cyclic is that it allows levels 2 and 3 BLAS operations on subvectors and submatrices within each processor.

Sample ScaLAPACK Program

"Simplest" program to solve $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ using `PDGESV` (LU), available at www.netlib.org/scalapack/examples/example1.f

```
1         PROGRAM EXAMPLE1
2   *
3   *     Example Program solving Ax=b via ScaLAPACK routine PDGESV
4   *
5   *     .. Parameters ..
6         INTEGER            DLEN_, IA, JA, IB, JB, M, N, MB, NB, RSRC,
7        $                   CSRC, MXLLDA, MXLLDB, NRHS, NBRHS, NOUT,
8        $                   MXLOCR, MXLOCC, MXRHSC
9         PARAMETER          ( DLEN_ = 9, IA = 1, JA = 1, IB = 1, JB = 1,
10       $                   M = 9, N = 9, MB = 2, NB = 2, RSRC = 0,
11       $                   CSRC = 0, MXLLDA = 5, MXLLDB = 5, NRHS = 1,
12       $                   NBRHS = 1, NOUT = 6, MXLOCR = 5, MXLOCC = 4,
13       $                   MXRHSC = 1 )
14        DOUBLE PRECISION   ONE
15        PARAMETER          ( ONE = 1.0D+0 )
16  *     ..
17  *     .. Local Scalars ..
18        INTEGER            ICTXT, INFO, MYCOL, MYROW, NPCOL, NPROW
19        DOUBLE PRECISION   ANORM, BNORM, EPS, RESID, XNORM
```

```
20  *      ..
21  *      .. Local Arrays ..
22        INTEGER            DESCA( DLEN_ ), DESCB( DLEN_ ),
23       $                   IPIV( MXLOCR+NB )
24        DOUBLE PRECISION   A( MXLLDA, MXLOCC ), A0( MXLLDA, MXLOCC ),
25       $                   B( MXLLDB, MXRHSC ), B0( MXLLDB, MXRHSC ),
26       $                   WORK( MXLOCR )
27  *      ..
28  *      .. External Functions ..
29        DOUBLE PRECISION   PDLAMCH, PDLANGE
30        EXTERNAL           PDLAMCH, PDLANGE
31  *      ..
32  *      .. External Subroutines ..
33        EXTERNAL           BLACS_EXIT, BLACS_GRIDEXIT, BLACS_GRIDINFO,
34       $                   DESCINIT, MATINIT, PDGEMM, PDGESV, PDLACPY,
35       $                   SL_INIT
36  *      ..
37  *      .. Intrinsic Functions ..
38        INTRINSIC          DBLE
39  *      ..
40  *      .. Data statements ..
41        DATA               NPROW / 2 / , NPCOL / 3 /
42  *      ..
43  *      .. Executable Statements ..
44  *
45  *      INITIALIZE THE PROCESS GRID
46  *
47        CALL SL_INIT( ICTXT, NPROW, NPCOL )
48        CALL BLACS_GRIDINFO( ICTXT, NPROW, NPCOL, MYROW, MYCOL )
```

```
49  *
50  *     If I'm not in the process grid, go to the end of the program
51  *
52        IF( MYROW.EQ.-1 )
53       $   GO TO 10
54  *
55  *     DISTRIBUTE THE MATRIX ON THE PROCESS GRID
56  *     Initialize the array descriptors for the matrices A and B
57  *
58        CALL DESCINIT( DESCA, M, N, MB, NB, RSRC, CSRC, ICTXT, MXLLDA,
59       $               INFO )
60        CALL DESCINIT( DESCB, N, NRHS, NB, NBRHS, RSRC, CSRC, ICTXT,
61       $               MXLLDB, INFO )
62  *
63  *     Generate matrices A and B and distribute to the process grid
64  *
65        CALL MATINIT( A, DESCA, B, DESCB )
66  *
67  *     Make a copy of A and B for checking purposes
68  *
69        CALL PDLACPY( 'All', N, N, A, 1, 1, DESCA, A0, 1, 1, DESCA )
70        CALL PDLACPY( 'All', N, NRHS, B, 1, 1, DESCB, B0, 1, 1, DESCB )
71  *
72  *     CALL THE SCALAPACK ROUTINE
73  *     Solve the linear system A * X = B
74  *
75        CALL PDGESV( N, NRHS, A, IA, JA, DESCA, IPIV, B, IB, JB, DESCB,
76       $             INFO )
```

```
*
      IF( MYROW.EQ.0 .AND. MYCOL.EQ.0 ) THEN
         WRITE( NOUT, FMT = 9999 )
         WRITE( NOUT, FMT = 9998 )M, N, NB
         WRITE( NOUT, FMT = 9997 )NPROW*NPCOL, NPROW, NPCOL
         WRITE( NOUT, FMT = 9996 )INFO
      END IF
*
*     Compute residual ||A * X  - B|| / ( ||X|| * ||A|| * eps * N )
*
      EPS = PDLAMCH( ICTXT, 'Epsilon' )
      ANORM = PDLANGE( 'I', N, N, A, 1, 1, DESCA, WORK )
      BNORM = PDLANGE( 'I', N, NRHS, B, 1, 1, DESCB, WORK )
      CALL PDGEMM( 'N', 'N', N, NRHS, N, ONE, A0, 1, 1, DESCA, B, 1, 1,
     $             DESCB, -ONE, B0, 1, 1, DESCB )
      XNORM = PDLANGE( 'I', N, NRHS, B0, 1, 1, DESCB, WORK )
      RESID = XNORM / ( ANORM*BNORM*EPS*DBLE( N ) )
*
      IF( MYROW.EQ.0 .AND. MYCOL.EQ.0 ) THEN
         IF( RESID.LT.10.0D+0 ) THEN
            WRITE( NOUT, FMT = 9995 )
            WRITE( NOUT, FMT = 9993 )RESID
         ELSE
            WRITE( NOUT, FMT = 9994 )
            WRITE( NOUT, FMT = 9993 )RESID
         END IF
      END IF
```

```
104 *
105 *     RELEASE THE PROCESS GRID
106 *     Free the BLACS context
107 *
108       CALL BLACS_GRIDEXIT( ICTXT )
109    10 CONTINUE
110 *
111 *     Exit the BLACS
112 *
113       CALL BLACS_EXIT( 0 )
114 *
115  9999 FORMAT( / 'ScaLAPACK Example Program #1  -  May 1, 1997' )
116  9998 FORMAT( / 'Solving Ax=b where A is a ', I3, ' by ', I3,
117     $      ' matrix with a block size of ', I3 )
118  9997 FORMAT( 'Running on ', I3, ' processes, where the process grid',
119     $      ' is ', I3, ' by ', I3 )
120  9996 FORMAT( / 'INFO code returned by PDGESV = ', I3 )
121  9995 FORMAT( /
122     $   'According to the normalized residual the solution is correct.'
123     $       )
124  9994 FORMAT( /
125     $ 'According to the normalized residual the solution is incorrect.'
126     $       )
127  9993 FORMAT( / '||A*x - b|| / ( ||x||*||A||*eps*N ) = ', 1P, E16.8 )
128       STOP
129       END
```

Compiling and running on UB/CCR (use the Intel MKL link advisor to get the linking right):

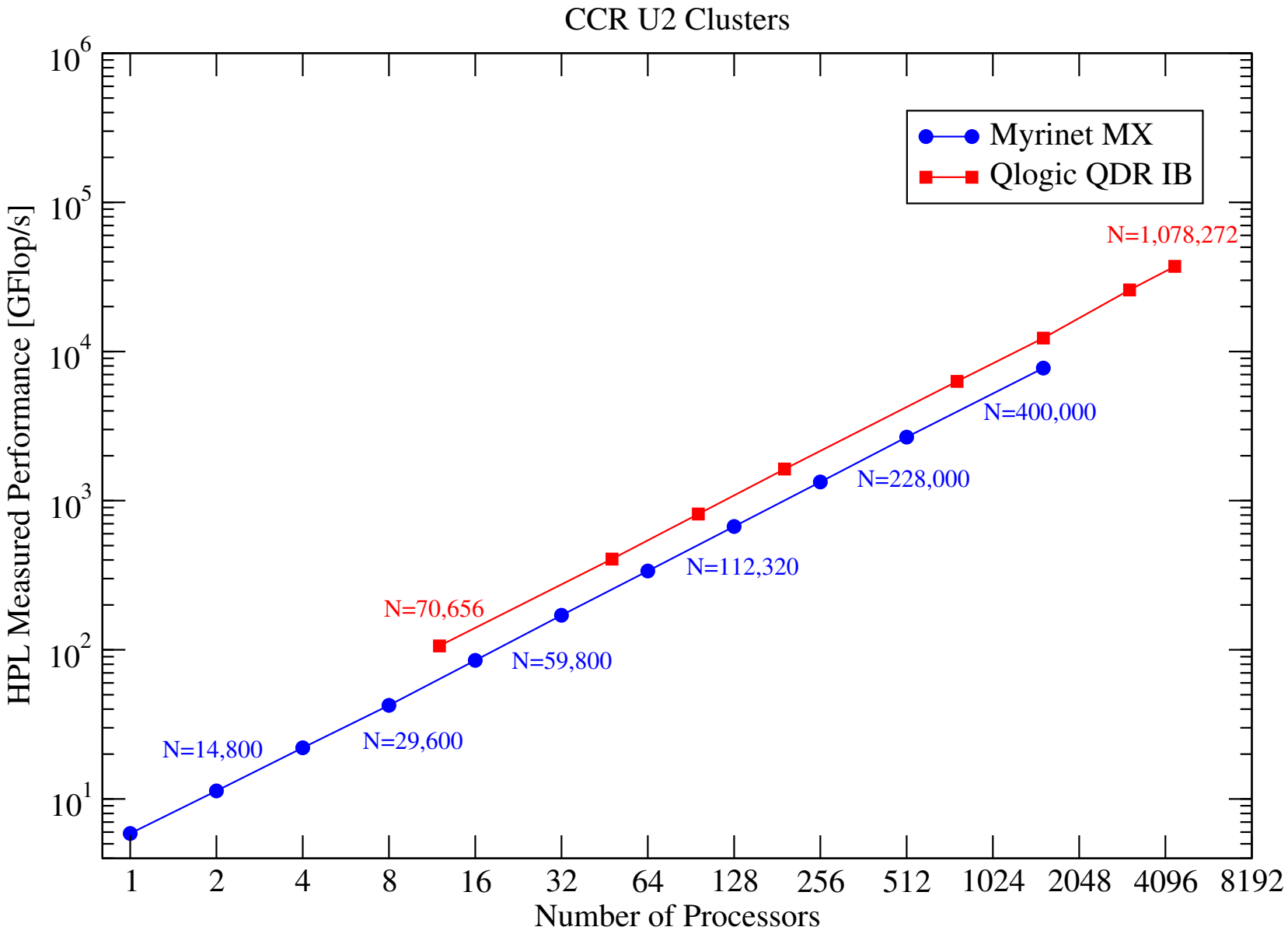
```
[rush:~/d_scalapack]$ module list
Currently Loaded Modulefiles:
  1) null              2) modules           3) use.own           4) intel-mpi/4.1.1
  5) intel/13.1        6) mkl/11.1
[rush:~/d_scalapack]$ mpiifort -o example1 example1.f -L$MKLROOT/lib/intel64 -lmkl_scalapack_lp64
       -lmkl_intel_lp64 -lmkl_core -lmkl_sequential -lmkl_blacs_intelmpi_lp64 -lpthread -lm
[rush:~/d_scalapack]$ ldd ./example1
    linux-vdso.so.1 =>  (0x00007fffa39ff000)
libmkl_scalapack_lp64.so => /util/intel/composer_xe_2013/mkl/lib/intel64/libmkl_scalapack_lp64.so (0x00002b42c0c0c000)
libmkl_intel_lp64.so => /util/intel/composer_xe_2013/mkl/lib/intel64/libmkl_intel_lp64.so (0x00002b42c14bb000)
libmkl_core.so => /util/intel/composer_xe_2013/mkl/lib/intel64/libmkl_core.so (0x00002b42c1bcf000)
libmkl_sequential.so => /util/intel/composer_xe_2013/mkl/lib/intel64/libmkl_sequential.so (0x00002b42c2e34000)
libmkl_blacs_intelmpi_lp64.so =>
/util/intel/composer_xe_2013/mkl/lib/intel64/libmkl_blacs_intelmpi_lp64.so (0x00002b42c34e3000)
libpthread.so.0 => /lib64/libpthread.so.0 (0x0000003e1ca00000)
libm.so.6 => /lib64/libm.so.6 (0x0000003e1c200000)
libmpigf.so.4 => /util/intel/impi/4.1.1.036/intel64/lib/libmpigf.so.4 (0x00002b42c3736000)
libmpi.so.4 => /util/intel/impi/4.1.1.036/intel64/lib/libmpi.so.4 (0x00002b42c3966000)
libdl.so.2 => /lib64/libdl.so.2 (0x0000003e1c600000)
librt.so.1 => /lib64/librt.so.1 (0x0000003e1d200000)
libc.so.6 => /lib64/libc.so.6 (0x0000003e1be00000)
libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x0000003e1e600000)
/lib64/ld-linux-x86-64.so.2 (0x0000003e1ba00000)
```

```
1  [rush:~/d_scalapack]$ mpirun -np 6 ./example1
2
3  ScaLAPACK Example Program #1  -  May 1, 1997
4
5  Solving Ax=b where A is a   9 by   9 matrix with a block size of   2
6  Running on   6 processes, where the process grid is   2 by   3
7
8  INFO code returned by PDGESV =   0
9
10 According to the normalized residual the solution is correct.
11
12 ||A*x - b|| / ( ||x||*||A||*eps*N ) =   0.00000000E+00
13 [rush:~/d_scalapack]$ mpirun -np 2 ./example1
14 application called MPI_Abort(MPI_COMM_WORLD, 1) - process 0
15 application called MPI_Abort(MPI_COMM_WORLD, 1) - process 1
```

ScaLAPACK Driver Illustrated - HPL

- HPL (High Performance Linpack) benchmark still used to rank the Top500 list (`www.top500.org`)
- Super-sized version of previous example code, uses `PDGESV` to solve randomly seeded linear system, check results to ensure that results are accurate
- Following plot shows results for UB/CCR's old Myrinet nodes (1536 processors total, circa 2005) and Qlogic QDR Infiniband (4416 cores total, circa 2011) - note scalability as the problem size is increased (essentially to maintain a fixed amount of memory used per node)

Linpack (HPL) Benchmark



Jacobi Iteration

Our method for "improving" on solutions to systems of linear equations can be formalized into a solution technique in its own right (Jacobi did it first):

$$x_i^{(k)} = \frac{1}{A_{i,i}} \left(b_i - \sum_{j \neq i} A_{i,j} x_j^{(k-1)} \right),$$

- Particularly useful in solution of (O|P)DEs using finite differences
- Advantage of small memory requirements (especially for sparse systems)
- Disadvantage of convergence problems (slowly, or not)
- Initial guess - often take $\mathbf{x}^0 = \mathbf{b}$

Measuring convergence for Jacobi Iteration can be tricky, particularly in parallel. At first glance, you might be tempted by:

$$\left| x_i^{(k+1)} - x_i^k \right| < \epsilon, \qquad \forall i,$$

but that says little about solution accuracy.

$$\left| \sum_j A_{i,j} x_j^k - b_i \right| < \epsilon$$

is a better form, and can reuse values already computed in the previous iteration.

Sequential

```
do i=1,N
   x(i)=b(i)
end do
do iter=1,MAXITER
   do i=1,N
      sum = 0.0
      do j=1,N
         if (i /= j) sum = sum + A(i,j)*x(j)
      end do
      new_x(i) = (b(i)-sum)/A(i,i)
   end do
   do i=1,N
      x(i) = new_x(i)
   end do
end do
```

Block Parallel

Have each process accountable for solving a particular **block** of unknowns, and simply use an **Allgather** or **Allgatherv** (for an unequal distribution of work on the processes). The resulting computation time is

$$\tau_{\text{comp}} = \frac{N}{N_p}(2N + 4)N_{\text{iter}}$$

with a time for communication

$$\tau_{\text{comm}} = (N_p\tau_{lat} + N\tau_{dat})N_{\text{iter}},$$

and the speedup factor:

$$S(N_p) = \frac{N(2N+4)}{N(2N+4)/N_p + N_p\tau_{lat} + N\tau_{dat}} == \frac{N_p}{1 + \tau_{\text{comm}}/\tau_{\text{comp}}}.$$

Note that this is scalable, but only if the ratio N/N_p is maintained with increasing N_p.

Gauss-Seidel Relaxation

Technique for convergence acceleration, usually converged faster than Jacobi,

$$x_i^{(k)} = \frac{1}{A_{i,i}} \left(b_i - \sum_{j=1}^{i-1} A_{i,j} x_j^{(k)} - \sum_{j=i+1}^{N} A_{i,j} x_j^{(k-1)} \right),$$

making use of just-updated solution. More difficult to parallelize, but can be done through particular patterning (say using a **checkerboard** allocation) in which regions can be computed simultaneously.

Overrelaxation

Another improved convergence technique in which factor $(1-\omega)x_i$ is added:

$$x_i^{(k)} = \frac{\omega}{A_{i,i}}\left(b_i - \sum_{j\neq i} A_{i,j}x_j^{(k-1)}\right) + (1-\omega)x_i^{(k-1)},$$

where $0 < \omega < 1$. For the Gauss-Seidel method, this is slightly modified:

$$x_i^{(k)} = \frac{\omega}{A_{i,i}}\left(b_i - \sum_{j=1}^{i-1} A_{i,j}x_j^{(k)} - \sum_{j=i+1}^{N} A_{i,j}x_j^{(k-1)}\right) + (1-\omega)x_i^{k-1},$$

with $0 < \omega \leq 2$.

Multigrid Method

Makes use of successively finer grids to improve solution:

- Coarse starting grid quickly take distant effects into account
- Initial values of finer grids available through interpolation from existing grid values
- Of course, we can even have regions of variable grid densities using so-called **adaptive** grid methods