Matrix-Vector Operations

January 21, 2014 9:05 PM

Note on storing large arrays: In this section we will look at matrix-vector operations and often will use very large arrays.
If you allocate an array using something like:

```
double a[100000];
```

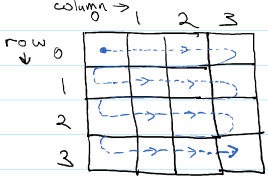
the array will be stored in something called the stack. This is also where the program keeps track of function calls and other code details. Space on the stack is limited so if you allocate a very large array in this manner you may get a stack "overflow", typically signalled by "seg fault 11".
If you dynamically allocate memory (through "new" command in c++ or "calloc"/"malloc" in c) the array will be stored in something called the heap, which does not have space limitations.
Things in the stack can often be accessed faster, so there are cons/benifits to both cases. However, if you arrange the order in which you access elements on the stack, you can often speed up your code by 20-30% due to caching effects. To understand this, and to understand how to send/recv multidimensional arrays using MPI we need to understand how multidimensional arrays are stored in memory.

C-style 2D arrays

consider a static array declared via:

```
double array[4][4];
```

we think of this as a matrix



Row-major order (C-style)

but device memory is addressed in a single linear sequence (i.e. 1D). Therefore, to store this 2D array in memory it must 1st be converted into a 1D sequence:

$$\texttt{array[i][j]} \rightarrow *(\&\texttt{array[0][0]} + i*4 + j)$$

- value stored at address →
- address of beginning of the array
- i times size of 1 row
- column number

This means array[i][j] ≡ array[0][i*4+j]
As there is no bounds checking, you can use either interchangably in code.
The compiler actually changes all multidimensional arrays into 1D arrays at compile time which is why in C you need to pass functions the number of columns in order for the function to decode what you mean by a[i][j].

In order to use external libraries like MPI (and others like the blas & lapack) we need to ensure dynamically allocated multidimensional arrays break into 1D storage in a manner consistent with a static 2D array.

There are several ways to dynamically allocate multidimensional arrays. Many are <u>not</u> consistent with static storage. We will do this so that the <u>entire</u> array is stored in one contiguous block of memory (<u>not</u> what is done in the text).

eg

```
1  int rows = 5, cols = 3;
2  double** A;
3  A = (double**) calloc(rows,sizeof(double*));
4  A[0] = (double*) calloc(rows*cols, sizeof(double));
5  for (int i=1; i<rows; i++)
6  {
7    A[i] = A[i-1]+ cols;
8  }
```

make pointer A[0] point to a 1D array with size rows*cols (i.e. the entire matrix)

make A[i] point to the position in our big 1D array where row i starts

Created in this way, we can use A as if it had been created as a fixed array using

```
double A[5][3];
```

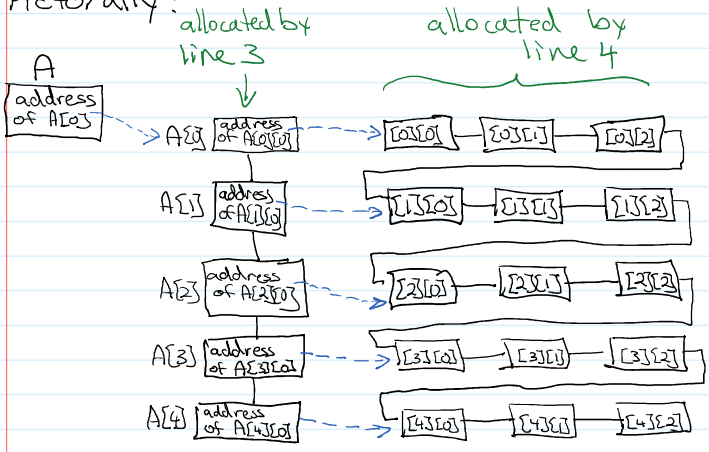
The only difference is that it does not know how big the array is through type information.

Note that:

1. Each A[i] is a pointer that points to A[i][0]
2. A is a pointer that points to A[0], which is itself a pointer pointing to A[0][0]

As such, A is pointer to an array of <u>pointers</u>, each of which points to the beginning of a new row.

Pictorally:



Note that the actual array is stored sequentially, just like a fixed array.

To deallocate this array, we <u>must do it in</u> reverse order from the way we created it with

reverse order from the "way we created it with the malloc statements:

eg
```
free(A[0]);   ← deallocates actual array
free(A);      ← deallocates array of row pointers
```

If we attempted to deallocate in the opposite order, we would have lost what A[0] pointed to (the actual array), and would have a big memory leak.

C++ 2D arrays

In order for us to use MPI and other external libraries, we need the actual data in our arrays to be stored in memory in the same way as a static multidimensional C array.
We can still do this with C++ vectors. To illustrate,

eg.

```
int rows = 5, cols = 3;
std::vector<double *> A;
A.reserve(rows);  A.resize(rows);
std::vector<double> Amem;
Amem.reserve(rows*cols);  Amem.resize(rows*cols);
A[0] = &(Amem[0]);
for (int i=1; i<rows; i++){
  A[i] = A[i-1]+ cols;
}
```

This sets up something analagous to what we did in C. i.e. we can use A[i][j] ha as if we declared the array statically as
double A[5][3];

However, this is obviously less than ideal and makes bounds checking very messy. The best way to avoid these problems is to use the boost libraries.

Boost

- boost is a set of libraries that are similar to, and often expand on, the STL. they are close enough to the standard that
- different parts of boost have already been incorporated into C++ standards in the past
- to use the boost libraries, we must first install them on our machine.
  On LINUX or WSL (windows bash shell) we install from a standard repository using:

  `sudo apt-get install libboost-all-dev`
  (libboost-all-dev: library to install)

  on the command line. You will be prompted for a password and then it will install boost plus anything else you may need to use it on your machine.
  On a MAC, you first need to install

something like MacPorts (www.macports.org). Go to their website, get an appropriate .pkg and then double-click to install. Then install boost using
sudo port install boost-devel

On linux/wsl you should not need to do anything else. In some systems you may have to link when you compile using something like -lboost_xx where xx is one of the boost sublibrary names.

Boost Example

std::vector is great for 1-dimensional arrays but does not naturally handle multi-dimensional arrays well. The Boost.Multi_Array library provides std::vector-like functionality to multi-dimensional arrays.

eg.

```
#include "boost/multi_array.hpp" <- from boost library
#include <cassert>

int
main () {
  // Create a 3D array that is 3 x 4 x 2
  typedef boost::multi_array<double, 3> array_type;
  array_type A(boost::extents[3][4][2]);

  // Assign values to the elements
  typedef array_type::index index;
  int values = 0;
  for(index i = 0; i != 3; ++i)
    for(index j = 0; j != 4; ++j)
      for(index k = 0; k != 2; ++k)
        A[i][j][k] = values++;

  // Verify values
  int verify = 0;
  for(index i = 0; i != 3; ++i)
    for(index j = 0; j != 4; ++j)
      for(index k = 0; k != 2; ++k)
        assert(A[i][j][k] == verify++);

  return 0;
}
```

element type
number of indices
<- this defines a type of array that has 3 indices
this declares an array of A with size 3 in 1st dimension, 4 in 2nd dimension, 2 in 3rd dimension
index for our array type
<- we can access elements just like a simple array

Note: The [] operator does bounds checking here. To disable range-checking (for performance reasons after code has been thoroughly tested), define BOOST_DISABLE_ASSERTS prior to the multi_array.hpp include.

We can also grab a part, or slice, of our array:

```
// myarray = 2 x 3 x 4
//
// array_view dims:
// [base,stride,bound)
// [0,1,2), 1, [0,2,4)
//
```

```
typedef boost::multi_array_types::index_range range;
array_type::index_gen indices;
array_type::array_view<2>::type myview =
    myarray[ indices[range(0,2)][1][range(0,4,2)] ];

for (array_type::index i = 0; i != 2; ++i)
  for (array_type::index j = 0; j != 2; ++j)
    assert(myview[i][j] == myarray[i][1][j*2]);
```

myview is now a 2d array consisting of 0:1 of 1st dim of myarray and 0,2 of 3rd dim all in [1] of 2nd dimension. so myview is 2x2

just to be clear which elements of myview match those of myarray

From https://www.boost.org/doc/libs/1_69_0/libs/multi_array/doc/user.html

We can also resize the array using the resize() function:

eg

```
typedef boost::multi_array<int, 3> array_type;
array_type::extent_gen extents;
array_type A(extents[3][3][3]);
A[0][0][0] = 4;
A[2][2][2] = 5;
A.resize(extents[2][3][4]);
assert(A[0][0][0] == 4);
// A[2][2][2] is no longer valid.
```

← define 3d array type
← define member variable to hold extents of array
← define our array A as 3x3x3
} set some elements
← resize the array to 2x3x4

From <http://www.boost.org/doc/libs/1_65_1/libs/multi_array/doc/user.html>

Note: this changes the length in each dimension, does not change # of dimensions

Array access and Performance

Consider a multidimensional array A with say 4-indices. Many algorithms will involve nested loops over the entries of such an array.
Does the loop order matter?
In many cases, not for getting the correct answer, but it can make a big difference to performance.
Why? Because when you retrieve an entry of an array from main memory, typically a block of nearby entries are loaded into cache at the same time.

eg.

row \ col	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

A

memory order

If we ask for A[0][3] there is a good chance we will "pre-fetch" A[1][0] at the same time as it is stored next in memory

⇒
```
for (int i= 0; i<3; ++i)
    for (int j=0; j<3; ++j)
        A[i][j] = atan(M_PI*i/3, M_PI*j/3);
```

will be faster than

```
for (int j= 0; j<3; ++j)
    for (int i=0; i<3; ++i)
        A[i][j] = atan(M_PI*i/3, M_PI*j/3);
```

Compiler Optimizations

There are multiple sources of error in Scientific Computing. Two main ones are:

1) finite precision arithmetic - over/underflow
   - roundoff errors
2) truncation or discretization errors
   - arise from fact that our models & algorithms usually only try to approximate the solution

In most cases, errors from the 2nd source dominate. We can then usually relax some of the constraints usually setup to minimize roundoff errors, such as strictly following IEEE rules on arithmetic to speed things up. This is done via compiler flags

e.g.

```
g++ -O3 program.cpp -o prog
```

or

```
mpiCC -O3 program.cpp -o prog
```

-O3 means apply agressive optimization to code

In many cases, this can substantially speed up run times.

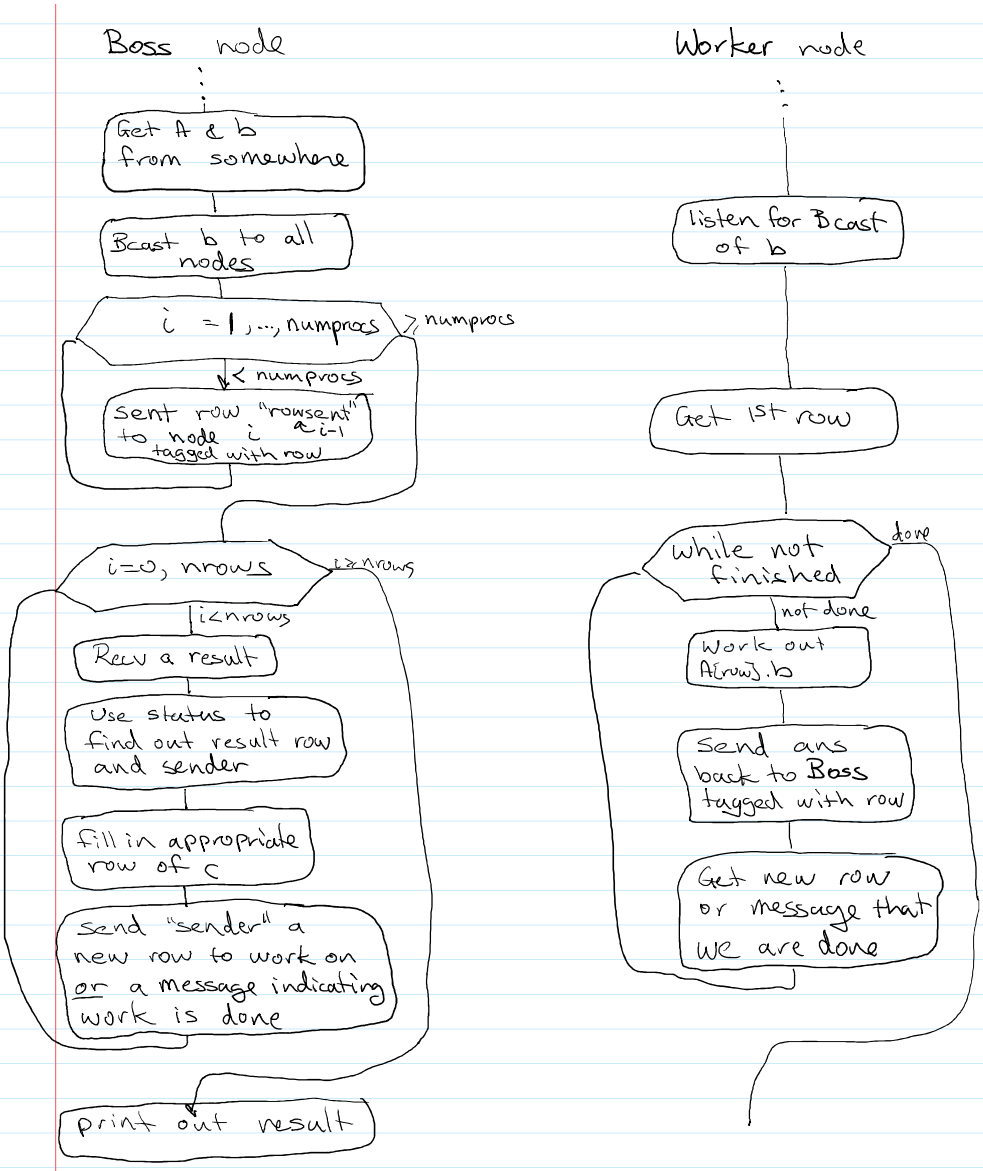
Matrix Operations in Parallel

eg. Let's write some code to do $A.b = c$ where A is a matrix and b and c are vectors. We will use a Boss-Worker model where workers do all computation while Boss directs the work:

To do this, we divide the task of $A.b = c$ into $A[i].b = c[i]$, $i = 0, .. ncols-1$

($A[i]$: ith row of ncolsxncols matrix; $c[i]$: ith row of ncolsx1 matrix)

and have different processors work on each of these tasks, until done.




```
//////////////////////////////////////////
// Matrix-vector multiplication code Ab=c //
//////////////////////////////////////////

// Note that I will index arrays from 0 to n-1.
// Here workers do all the work and boss just handles collating results
// and sending info about A.

// include, definitions, globals etc here
#include <iostream>
#include "boost/multi_array.hpp" <- use boost arrays
#include "mpi.h"

… mpi stuff declared here

int main(int argc, char** argv)
{// initialize MPI
  MPI_stuff the_mpi(argc, argv);

  // determine/distribute size of arrays here to all nodes
  …

  // Allocate and initialize A and b here on the Boss node and space for c
```

```
  // Workers need space for b and a row of A, Arow
  // Assume A will have rows 0,nrows-1 and columns 0,ncols-1, so b is 0,ncols-1
  // so c must be 0,nrows-1.  Note declarations need to be outside of an if block to
  // avoid going out of scope.  Boss should initialize A and only Boss needs this whole
  // matrix.  Workers need only space for a row of A which we will call Arow
  …

  // send b to every worker process, note b is a std::vector so b and &b[0] not same
  MPI_Bcast(&b[0],ncols, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  MPI_Status status;
  // Boss part
  if (the_mpi.MyID == 0) {
    // send one row to each worker tagged with row number, assume nprocs<nrows
    int rowsent=0;
    for (int i=1; i< the_mpi.NProcs; i++) {
      MPI_Send(&A[rowsent][0], ncols, MPI_DOUBLE,i,rowsent+1,MPI_COMM_WORLD);
      rowsent++;
    }

    for (int i=0; i<nrows; i++) {
      double ans;
      MPI_Recv(&ans, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      int sender = status.MPI_SOURCE;
      int anstype = status.MPI_TAG;
      c[anstype-1] = ans;
      if (rowsent < nrows) {
        MPI_Send(&A[rowsent][0],ncols,MPI_DOUBLE,sender,rowsent+1,MPI_COMM_WORLD);
        rowsent++;
      }
      else       // tell sender no more work to do via a 0 TAG
        MPI_Send(MPI_BOTTOM,0,MPI_DOUBLE,sender,0,MPI_COMM_WORLD);
    }
  }
  // Worker part: compute dot products of Arow.b until done message received
  else {
    // Get a row of A
    MPI_Recv(&Arow[0],ncols,MPI_DOUBLE,0,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
    while(status.MPI_TAG != 0) {
      int crow = status.MPI_TAG;  // which row did we get

      // work out Arow.b
      double ans=0.0;
      for (int i=0; i< ncols; i++)
        ans+=Arow[i]*b[i];

      // Send answer of Arow.b back to boss and get another row to work on
      MPI_Send(&ans,1,MPI_DOUBLE, 0, crow, MPI_COMM_WORLD);
      MPI_Recv(&Arow[0],ncols,MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    }
  }

  // output c here on Boss node
  …

  // in C you should free space and MPI_Finalize here
}
```

want to reserve TAG=0

together, these mean we take next message in queue regardless of source or tag

who sent (new!)

which answer //row number+1

// send new row to who we got result from

address of "bottom of memory", just a placeholder

no actual buffer is sent, just TAG & other status stuff

keep working until we get a 0 tag

actual row of A dotted with b

Note: 1. Here we only need the full matrix A on BOSS node.
It would be tempting to declare A inside an if block
(e.g. if (MyID==0)
└ declare & allocate A)

but then A would immediatly go out of scope. In this case create the pointers to A on all and only allocate space on BOSS

In C:

```
double ** A;
if (the_MPI.MyID ==0) {
  A = (double **) calloc(nrows, sizeof(double*)
  A[0] = (double *) calloc(nrows*ncols, sizeof(double);
  … etc
}
```

In C++:

```
typedef boost::multi_array<double, 2> A_type;
A_type A;
if (the_mpi.MyID == 0) {
  // Set size of A
  A.resize(boost::extents[nrows][ncols]);

  … etc
}
```

and similar with std::vector on 1D arrays

This code is an example of Dynamic Load Balancing in the sense that if one node is bogged down doing something else, other nodes which finish faster will get more work.

This seems like a brilliant idea! What could go wrong? Let's analyze to see:

#flops = n rows · n columns = n^2 (multiplication + add) for square matrix A
(multiplication + add) = 1 flop

compute time = $n^2 T_{flop}$ in serial
(T_{flop} ← time required to compute 1 flop)

If we do this on p processors using above algorithm

doubles sent/recv. $\approx (n+1)\cdot n = n^2+n$
(send a row of A; send back answer; n rows)

communication time $\approx (n^2+n) T_{comm}$
(T_{comm} ← time to send one double)

compute time spread over p processors so in ideal case

compute time = $\frac{n^2 T_{flop}}{p}$ so

total runtime on p processors $\sim \frac{n^2}{p} T_{flop} + (n^2+n) T_{comm}$

Now T_{flop} is of order 1 clock cycle ~ 0.5 ns for a 2 GHz core

A Gigabit ethernet connection takes ~ 64 bits x 1ns = 64 ns to transmit a double

so $T_{comm} > T_{flop}$.

S(p) = speedup ≡ $\frac{\text{runtime on 1 processor}}{}$

runtime on p processors

An "ideal" result would be

$S(p) \approx a\,p$ i.e. running on p processors is p times faster if $a=1$

$a=1$ is called linear

$a>1$ is called "superlinear" (sadly not a mathematically sensible statement)

↑ possible due to caching effects when program able to keep smaller data associated with breaking a task up in cache rather than main memory

$S<1$ called "slowdown", sadly this is easy to achieve.

here $$S(p) \sim \left[\frac{\left[\frac{n^2 T_{flop}}{p} + (n^2+n)\,T_{comm}\right]}{n^2\,T_{flop}}\right]^{-1}$$

$$\sim \left[\frac{1}{p} + \frac{T_{comm}}{T_{flop}}\right]^{-1} < 1$$

$\Rightarrow$ The program will likely run slower in parallel.

eg. Matrix - Matrix multiplication

Let's generalize the above code to do Matrix-Matrix multiplication

$A \cdot B = C$ (all nxn matrices)

```
////////////////////////////////////////////////////
// changes for Matrix-Matrix multiplication code //
////////////////////////////////////////////////////
{

  // changes to BOSS part: you figure it out

  //changes to Worker part...
    // worker receive B, then compute rows of C until done message received
    MPI_Bcast(B,Bcols*Brows, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Recv(&Arow[0], Acols, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    while (status.MPI_TAG != 0) {
      int crow = status.MPI_TAG;

      // Work out Arow.B = Crow
      for (i=0; i< Bcols; i++) {
        Crow[i]=0.0;
        for (j=0; j< Acols; j++)
          Crow[i] += Arow[j]*B[j][i];
      }
      MPI_Send(Crow,Bcols,MPI_DOUBLE, 0,crow,MPI_COMM_WORLD);
      MPI_Recv(&Arow[0], Acols, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    }

    // blah, blah, blah...
```

all of B

assumes B stored as a contiguous block of memory

Now lets analyze this code:

flops required for $A.B = n^3 \cdot n = n^4$

(n^3: $A.b(i)$; n: n columns in B)

compute time using p processors $\sim \frac{n^3}{p} T_{flop}$

doubles sent/recv $= (n + n) \cdot n = 2n^2$

(n of A: send row; n: $A.b(row)$ send back; n: rows)

(note BCast of B is $O(n^2)$ operations too)

total runtime on p processors $\sim \frac{n^3}{p} T_{flop} + 2n^2 T_{comm}$

So now speedup is

$$S(p) = \left[\frac{\frac{n^3}{p} T_{flop} + 2n^2 T_{comm}}{n^3 T_{flop}} \right]^{-1} = \left[\frac{1}{p} + \frac{2}{n} \frac{T_{comm}}{T_{flop}} \right]^{-1}$$

$\frac{2}{n} \frac{T_{comm}}{T_{flop}} \to 0$ as $n \to \infty$

So here $S(p) \to p$ as $n \to \infty$ (fixed p).
(NB: n may need to be very large in order to overcome $T_{comm} > T_{flop}$).

Timing

The routine MPI_Wtime(), returns a double precision (sycronized) time in seconds

e.g.
```
begintime = MPI_Wtime();
        ≋
endtime = MPI_Wtime();
```
to time stuff in between

To evalute parallel performance it is often worth timing the computational part seperate from the communication part, (e.g. by putting a t_{com} around MPI calls)

In addition to speedup, it is also interesting to consider

$$\text{efficiency} = \frac{\text{runtime on 1 processor}}{p * (\text{runtime on } p \text{ processors})}$$

efficiency $= 1 \Rightarrow$ "perfect" parallelism

Other considerations:

Typically

$T_{message} = t_s + k\, t_c$ (k: number of units of data in message)

Typically

$$T_{message} = t_s + k\,t_c$$

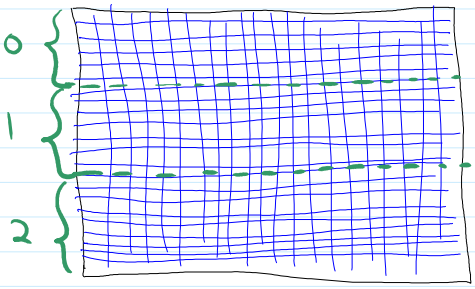
- $T_{message}$: time to send one message
- t_s: set-up or "latency" time
- k: number of units of data in message
- t_c: time to communicate one unit of data

- t_s & t_c vary enormously among different hardware/implementations
- typically you cannot ignore t_s (i.e. $t_s >> t_c$) so send data in largest possible chunks

These considerations suggest dividing up Matrix operations differently for systems where dynamic load balancing is not necessary (still do static load balancing)
e.g. SharcNet gives n-processor MPI jobs exclusive use of n-processors so dynamic balancing is not really necessary.

Consider an array in C/C++ declared as a contiguous block of memory:

A



Then do matrix operations on blocks.

MPI has some useful commands for doing this automatically:

MPI_Scatter (*void* *send_buffer, *int* send_count, *MPI_DATATYPE* *send_type*, *void* *recv_buffer, *int* recv_count, *MPI_DATATYPE* recv_type, *int* root, *MPI_Comm* comm)
Sends data from one processor to all other processes in a group
The process with rank ***root*** distributes the contents of ***send_buffer*** among the processes. The contents of send_buffer are split into ***p*** segments (***p*** being the number of processors involved) each consisting of ***send_count*** elements. The first segment goes to process 0, the second to process 1 ,etc. Note that ***p* send_count*** elements after the address send_buffer are sent. It is up to you to ensure that this number of elements is available. The send arguments are significant only on process root, in the sense that they are essentially ignored on the other processors.

MPI_Gather (*void* *send_buffer, *int* send_count, *MPI_DATATYPE* *send_type*, *void* *recv_buffer, *int* recv_count, *MPI_DATATYPE* recv_type, *int* root, *MPI_Comm* comm)
Gathers together values from a group of tasks
Each process in comm sends the contents of ***send_buffer*** to the process with rank ***root***. The process root concatenates the received data in the process rank order in ***recv_buffer***. The receive arguments are significant only on the process with rank root. The argument ***recv_count*** indicates the number of items received from each process - not the total number received.

MPI_Allgather (*void* *send_buffer, *int* send_count, *MPI_DATATYPE* send_type, *void* *recv_buffer, *int* recv_count, *MPI_Datatype* recv_type, *MPI_Comm* comm)
Gathers data from all processes and distribute it to all
MPI_Allgather gathers the contents of each ***send_buffer*** on each process. Its *effect* is the same as if there were a sequence of ***p*** calls to MPI_Gather, each of which has a different process acting as a ***root.***

Useful variants of these commands are MPI_Scatterv and MPI_Gatherv which allow different sized segments sent to different processors.

(B.T.W. a great resource to get more info on a specific MPI command, often including an example, can be found at: www.open-mpi.org/doc/v1.5/

There are other sites, but I find the descriptions here to be fairly complete, without being overly verbose.)

Before building more efficient Matrix operations in parallel, let's step back and consider how to do this in serial.

BLAS & LAPACK

Let's discuss the Basic Linear Algebra Subprograms first. These are standards over which all supercomputers (and regular computers) are tested. There are several implementations tuned for specific architectures.

There are 3 levels of routines:

BLAS1: vector operations (eg. dot products, norms, $\vec{y} \leftarrow \alpha \vec{x} + \vec{y}$ (scalar)

BLAS2: matrix-vector operations, eg. $\vec{y} \leftarrow \alpha A \vec{x} + \beta \vec{y}$ (matrix; scalar)

also solve $T\vec{x} = \vec{y}$ for T triangular

BLAS3: matrix-matrix operations eg. $C \leftarrow \alpha AB + AC$ (matrices; scalar)

Routine to do this is dgemm

BLAS Nomenclature

How did I know to look for something as weirdly named as "dgemm()"? The BLAS have a (mostly) systematic naming convention.

- For almost all BLAS, the first letter of the routine is one from the set {S,D,C,Z}. Those indicate single precision, double precision, complex, double complex, resp. Exception: *i*amax(n,x,incx)* where * = one of S,D,C,Z. Since P573.
- BLAS2/BLAS3 have second and third letters giving the type of matrix involved:
  - GE = general matrix
  - GB = general banded matrix
  - GP = general "packed" matrix
  - HE = general Hermitian matrix
  - SY = general symmetric matrix
  - TR = triangular matrix
  - ...
- BLAS2/BLAS3 have fourth and fifth letters giving the type of operation.
  - MV = matrix-vector product.
  - SV = solve (for triangular matrices only).

- MM = matrix-matrix product.
- R* = low rank update (R1, R2).

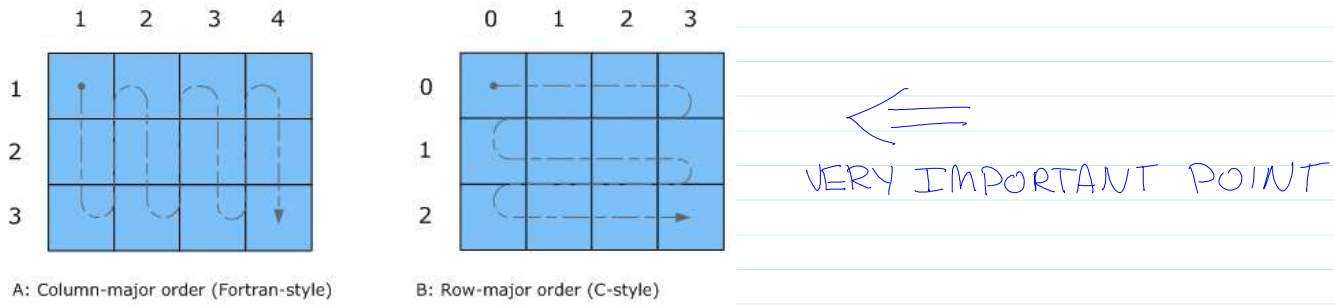
- Vector operations will have an "increment" associated with each vector (normally you would pick one, but you could do a vector operation on an column of a 2d array by picking an increment equal to the number of columns in a row. For example, the daxpy call is *daxpy(n, alpha, x, incx, y, incy)*. This is a BLAS1 routine and the name means: D for double precision and the a scalar "a"* (vector "x") +(the "p" is for +) (vector "y")

How do you find out what BLAS are exist? Use the man pages, or just plain old Google. You can also use the BLAS Quick Reference Card (Posted on OWL) although learning to read the card requires more than a little preliminary practice.
Part of the naming obscurity comes from a limitation of Fortran compilers when the BLAS were first proposed. An identifier in Fortran could have 6-8 characters, so the only safely portable way to proceed was to limit names to 6 characters. The latest standards have increased that limit up to 64 characters, which makes a major difference in readability. A function named *matrix_matrix_multiply()* is readable and easy to figure out; the name *dgemm()* is not.

BLAS Calling Conventions
When using the BLAS, beware that

- The BLAS have to be callable from both Fortran C/C++. That means all arguments passed in are **addresses**. So if the length of the vector is n, from C you need to pass the argument &n. Of course, from a Fortran program it is called with argument n; the default in Fortran is to pass the address, not a copy of the actual datum. C-interface versions of the BLAS can make life easier when writing from C/C++ (generally called CBLAS). The problem is that the interface is nonstandard, so code using it may not port to another platform. Furthermore, your library routine may be used by a code that itself calls libraries that use the standard BLAS, which can cause trouble for linkers.
- 2D arrays are presumed to be in **column-major** order; that is, in the computer's memory, consecutive locations correspond to going down one column after another. Again, see the note about the C-interface versions - some assume a row-by-row layout others the usual column major order, which makes using a C/C++ interface non-portable.
- One option is to store your data in Fortran style, that is, column-major rather than row-major order.

With row-major order, adopted in C, the last array index changes most quickly and the first one changes most slowly when traversing the memory segment where the array is stored. With Fortran-style column-major order, the last index changes most slowly whereas the first index changes most quickly (as illustrated by the figure below for a two-dimensional array).



A: Column-major order (Fortran-style) B: Row-major order (C-style)

For example, if a two-dimensional matrix A of size mxn is stored densely in a one-dimensional array B, you can access a matrix element like this:
A[i][j] = B[i*n+j] in C (i=0, ... , m-1, j=0, ... , -1)
A(i,j) = B((j-1)*m+i) in Fortran (i=1, ... , m, j=1, ... , n).

- When calling LAPACK or BLAS routines from C, be aware that because the Fortran language is case-insensitive, the routine names can be both upper-case or lower-case, with or without the trailing underscore. For example, the following names are equivalent:
  - LAPACK: dgetrf, DGETRF, dgetrf_, and DGETRF_
  - BLAS: dgemm, DGEMM, dgemm_, and DGEMM_
- The **leading dimension** of each 2D array needs to be passed in, usually in an integer variable **lda**. Mapping *matrix coordinates* (i,j) into singly-indexed *computer memory coordinates* requires the array's declared leading dimension, as well as the sizes of the matrix stored in the 2D array.
- When calling Fortran-based BLAS from a C routine, the invoked BLAS function must be declared as "extern". For example,

```
extern void dgemv_(char *trans, int *m, int *n,
                   double *alpha, double *a, int *lda, double *x, int *incx,
                   double *beta,  double *y, int *incy );
```

- When calling from a C++ function, you will need to subvert the name-mangling that C++ does. This can be done using a header file that says to use the C-calling convention for the loader:

```
extern "C" {
extern void dgemv_(char *trans, int *m, int *n,
                   double *alpha, double *a, int *lda, double *x, int *incx,
                   double *beta,  double *y, int *incy ); };
```

  Inside the braces, put all the BLAS (and C and Fortran) functions you will be calling.
- Usually a header file is provided along with the BLAS that does the above declarations for you, and you just have to

#include it.

- Linking in the BLAS library requires using something like **-lblas** on the link line of compilation.
- Since the BLAS are compiled using **-dalign**, you will need to do the same for your functions/routines in any language.
- Using C/C++ as the main program and calling the standard BLAS (i.e., the Fortran convention ones) some Fortran libraries need to also be linked in. Look for a library like **-lftn** or something similar if necessary.
- When calling the standard BLAS from C/C++, most machines require appending an underscore to the name of the BLAS routine. So a call is to daxpy_() instead of daxpy(). IBM machines do not follow this "underscore convention", which is irksome for mixed-language codes. When the underscore convention is followed, calling the BLAS matrix-vector product function looks like

```
double alpha = 1.0, beta = 0.0;
int incx = 1, incy = 1;
int tda = 100;
double A[10000]; // for a 100x100 matrix.  or a 10x1000 matrix. or
                 // any matrix with fewer than 10001 entries
char trans = 'N';
 .
 .
 .
dgemv_(&trans,&n,&n,&alpha,A,&tda,x,&incx,&beta,y,&incy);
```

Again, note that the values are passed in as addresses when calling the Fortran BLAS from C.

Adapted From <http://www.cs.indiana.edu/classes/p573/notes/arch/09_blas2.html>
and <http://software.intel.com/sites/products/documentation/hpc/mkl/mkl_userguide_lnx/GUID-ABCC618B-43C4-4DCD-ADA2-6F061B5116CD.htm>

There are multiple implementations of the BLAS and in particular

1) intel MKL library includes a version that when used with the intel library can be very fast (on machines with intel chips, they have got into trouble for attempting to slow AMD chips)
2) AMD has a library called BLIS and libFLAME with similar functionality.
3) OS implementations which are easy to install/use but may not be as fast. We will use these as they are easiest to setup and don't cost $.

There are C implementations of the BLAS but we will stick with the standard FORTRAN versions as this leads to more portable code and LAPACK is still primarily only fully implemented in FORTRAN.

On WSL/LINUX we can add BLAS & LAPACK (which uses BLAS) via

```
sudo apt-get install liblapack-dev
```

Using MacPorts I would guess the analagous instruction would be

```
sudo port install lapack-devel
```

As LAPACK uses blas, this should install both.

Here is an example that deals with the row-major versus column major ordering in 4 different ways:
DGEMM does the operation

$C \leftarrow \alpha\, op(A) \cdot op(B) + \beta C$ (A, B, C: matrices)

note, C is overwritten & comes out in column-major form

DGEMM does the operation

$C \leftarrow \alpha\, op(A) \cdot op(B) + \beta C$ (matrices)

note, C is overwritten & comes out in column-major form

$op(A) = A$ or $op(A) = A^T$ (transpose) or $op(A) = A^+$ (adjoint)

A good way to figure out the function arguments is from the netlib lapack website:
http://www.netlib.org/lapack/explore-html/index.html

then search for dgemm (take 1st option, not the actual .f file)

eg.

```
#include <stdio.h>

// compile with
// gcc Blas_dgemm.c -lblas

// dgemm_ is a symbol in the BLAS library files
extern int dgemm_(char*,char*,int*,int*,int*,double*,double*,int*,double*,
int*,double*,  double*, int*);
int main(void)
{
  // Method I
  // store as normal C-style array in row-major form
  // a is 2x3, b is 3x2, c is 2x2 and storage is row-major here
  double A2[2][3];
  double B2[3][2];
  double C2[2][2];

  A2[0][0] = 0.11; A2[0][1] = 0.12; A2[0][2] = 0.13;
  A2[1][0] = 0.21; A2[1][1] = 0.22; A2[1][2] = 0.23;
  B2[0][0] = 1011; B2[0][1] = 1012;
  B2[1][0] = 1021; B2[1][1] = 1022;
  B2[2][0] = 1031; B2[2][1] = 1032;
  C2[0][0] = 0.00; C2[0][1] = 0.00;
  C2[1][0] = 0.00; C2[1][1] = 0.00;

  //We want A.B but we have A and B in row-major so need to "transpose" to
  // get in col-major form
  char transa='T', transb='T';  // op(A) = A, similar for B
  double alpha=1.0, beta=0.0;
  int m=2, n=2, k=3;   // m=rows of op(A), n= cols of op(B), k= cols of op(A)
                       // and rows of op(B)
  int lda=3, ldb=2, ldc=2; // leading dimensions of A, B, C as declared is
                           // fastest changing index, ie. last index in C-style

  /* Compute C = alpha*op(A)*op(B) + beta*C  using DGEEM and C is overwritten */
  dgemm_(&transa, &transb, &m, &n, &k, &alpha, &A2[0][0], &lda,
    &B2[0][0], &ldb, &beta, &C2[0][0], &ldc);

  // Note that we get c back in col-major order so output transpose
  printf ("[ %g, %g\n", C2[0][0], C2[1][0]);
  printf (" %g, %g ]\n", C2[0][1], C2[1][1]);

  /*************************************************************************/
  // Method II
  // store as 2D array in col-major form
  // a is 2x3, b is 3x2, c is 2x2 and storage is row-major here but
  // let's interpret first index as column and 2nd as row
  double A3[3][2];
  double B3[2][3];
  double C3[2][2];
```

C needs this linking explicitly

need to explicitly declare a prototype for blas functions and declare them as extern. Also note the '_' at the end of the name

```
A3[0][0] = 0.11; A3[1][0] = 0.12; A3[2][0] = 0.13;
A3[0][1] = 0.21; A3[1][1] = 0.22; A3[2][1] = 0.23;
B3[0][0] = 1011; B3[1][0] = 1012;
B3[0][1] = 1021; B3[1][1] = 1022;
B3[0][2] = 1031; B3[1][2] = 1032;
C3[0][0] = 0.00; C3[1][0] = 0.00;
C3[0][1] = 0.00; C3[1][1] = 0.00;

//We want A.B but we have A and B in col-major so no "transpose" needed
transa='N'; transb='N';
alpha=1.0; beta=0.0;
m=2; n=2; k=3;   // m=rows of op(A), n= cols of op(B), k= cols of op(A)
                 // and rows of op(B)
lda=2; ldb=3; ldc=2;  // leading dimensions of A, B, C as declared where
          // "leading" means fastest changing which is actually the last in C

/* Compute C = alpha*op(A)*op(B) + beta*C  using DGEEM and C is overwritten */
dgemm_(&transa, &transb, &m, &n, &k, &alpha, &A3[0][0], &lda,
  &B3[0][0], &ldb, &beta, &C3[0][0], &ldc);

// Note that we get c back in col-major order so output with row-column switch
printf ("[ %g, %g\n", C3[0][0], C3[1][0]);
printf (" %g, %g ]\n", C3[0][1], C3[1][1]);

/*********************************************************************/
//Method III
// store as linear array in row-major form
// a is 2x3, b is 3x2, c is 2x2 and storage is row-major here
// so in col-major order we have a as 3x2, b as 2x3, and c as 2x2
double a[] = { 0.11, 0.12, 0.13, 0.21, 0.22, 0.23 };
double b[] = { 1011, 1012, 1021, 1022, 1031, 1032 };
double c[] = { 0.00, 0.00, 0.00, 0.00 };

//We want A.B but need to "transpose" to col-major order (not real transpose)
// to get actual A and actual B
transa='T'; transb='T';  // op(A) = A transpose, similar for B
alpha=1.0; beta=0.0;
m=2; n=2; k=3;   // m=rows of op(A), n= cols of op(B), k= cols of op(A)
                 // and rows of op(B)
lda=3; ldb=2; ldc=2;  // leading dimensions of A, B, C as declared

/* Compute C = alpha*op(A)*op(B) + beta*C  using DGEEM and C is overwritten */
dgemm_(&transa, &transb, &m, &n, &k, &alpha, &a[0], &lda,
  &b[0], &ldb, &beta, &c[0], &ldc);

// Note that we get the transpose of c back again due to col-major order
printf ("[ %g, %g\n", c[0], c[2]);
printf (" %g, %g ]\n", c[1], c[3]);

/*********************************************************************/
// Method IV
// store as linear array in col-major form
// a is 2x3, b is 3x2, c is 2x2 and storage is col-major here
double A[] = { 0.11, 0.21, 0.12, 0.22, 0.13, 0.23 };
double B[] = { 1011, 1021, 1031, 1012, 1022, 1032 };
double C[] = { 0.00, 0.00, 0.00, 0.00 };

//We want A.B which are already in col-major form so don't transpose
transa='N'; transb='N';  // op(A) = A, similar for B
alpha=1.0; beta=0.0;
m=2; n=2; k=3;   // m=rows of op(A), n= cols of op(B), k= cols of op(A)
                 // and rows of op(B)
lda=2; ldb=3; ldc=2;  // leading dimensions of A, B, C as declared
```

```
  /* Compute C = alpha*op(A)*op(B) + beta*C  using DGEEM and C is overwritten */
  dgemm_(&transa, &transb, &m, &n, &k, &alpha, &A[0], &lda,
    &B[0], &ldb, &beta, &C[0], &ldc);

  // Note that we get c back in col-major order
  printf ("[ %g, %g\n", C[0], C[2]);
  printf (" %g, %g ]\n", C[1], C[3]);

 return 0;
}
```

This is similar, but not identical in C++. We only illustrate the 1st two methods in the C++ example, using the boost multi-arrays:

```
#include <stdio.h>
#include "boost/multi_array.hpp"

// compile with
// g++ Blas_dgemm.cpp -lblas

// dgemm_ is a symbol in the BLAS library files
extern "C" {
  extern int dgemm_(char*,char*,int*,int*,int*,double*,double*,int*,double*,
int*,double*,  double*, int*);
}

int main (void)
{
  // Method I
  // store as normal C-style array in row-major form
  // a is 2x3, b is 3x2, c is 2x2 and storage is row-major here
  boost::multi_array<double,2> A2(boost::extents[2][3]);
  boost::multi_array<double,2> B2(boost::extents[3][2]);
  boost::multi_array<double,2> C2(boost::extents[2][2]);

  A2[0][0] = 0.11; A2[0][1] = 0.12; A2[0][2] = 0.13;
  A2[1][0] = 0.21; A2[1][1] = 0.22; A2[1][2] = 0.23;
  B2[0][0] = 1011; B2[0][1] = 1012;
  B2[1][0] = 1021; B2[1][1] = 1022;
  B2[2][0] = 1031; B2[2][1] = 1032;
  C2[0][0] = 0.00; C2[0][1] = 0.00;
  C2[1][0] = 0.00; C2[1][1] = 0.00;

  //We want A.B but we have A and B in row-major so need to "transpose" to
  // get in col-major form
  char transa='T', transb='T';  // op(A) = A, similar for B
  double alpha=1.0, beta=0.0;
  int m=2, n=2, k=3;   // m=rows of op(A), n= cols of op(B), k= cols of op(A)
                       // and rows of op(B)
  int lda=3, ldb=2, ldc=2; // leading dimensions of A, B, C as declared is
                           // fastest changing index, ie. last index in C-style

  /* Compute C = alpha*op(A)*op(B) + beta*C  using DGEEM and C is overwritten */
  dgemm_(&transa, &transb, &m, &n, &k, &alpha, &A2[0][0], &lda,
    &B2[0][0], &ldb, &beta, &C2[0][0], &ldc);

  // Note that we get c back in col-major order so output transpose
  printf ("[ %g, %g\n", C2[0][0], C2[1][0]);
  printf (" %g, %g ]\n", C2[0][1], C2[1][1]);

  /*****************************************************************/
  // Method II
```

```
   printf (" %g, %g ]\n", C2[0][1], C2[1][1]);

   /*************************************************************************/
   // Method II
   // store as 2D array in col-major form
   // a is 2x3, b is 3x2, c is 2x2 and storage is col-major here
   boost::multi_array<double,2> A3(boost::extents[2][3],boost::fortran_storage_order());
   boost::multi_array<double,2> B3(boost::extents[3][2],boost::fortran_storage_order());
   boost::multi_array<double,2> C3(boost::extents[2][2],boost::fortran_storage_order());

   A3[0][0] = 0.11; A3[0][1] = 0.12; A3[0][2] = 0.13;
   A3[1][0] = 0.21; A3[1][1] = 0.22; A3[1][2] = 0.23;
   B3[0][0] = 1011; B3[0][1] = 1012;
   B3[1][0] = 1021; B3[1][1] = 1022;
   B3[2][0] = 1031; B3[2][1] = 1032;
   C3[0][0] = 0.00; C3[0][1] = 0.00;
   C3[1][0] = 0.00; C3[1][1] = 0.00;

   //We want A.B but we have A and B in col-major so no "transpose" needed
   transa='N'; transb='N';
   alpha=1.0; beta=0.0;
   m=2; n=2; k=3;    // m=rows of op(A), n= cols of op(B), k= cols of op(A)
                     // and rows of op(B)
   lda=2; ldb=3; ldc=2;  // leading dimensions of A, B, C as declared where
            // "leading" means fastest changing which is actually the last in C

   /* Compute C = alpha*op(A)*op(B) + beta*C  using DGEEM and C is overwritten */
   dgemm_(&transa, &transb, &m, &n, &k, &alpha, &A3[0][0], &lda,
     &B3[0][0], &ldb, &beta, &C3[0][0], &ldc);

   // Note that we get c back in col-major order but we have stored in fortran form
   // so no output transpose is necessary here
   printf ("[ %g, %g\n", C3[0][0], C3[0][1]);
   printf (" %g, %g ]\n", C3[1][0], C3[1][1]);

  return 0;
 }
```

option to store in col-major form

The Intel MKL library is very well documented. Let's look at some examples:

Example "Calling a Complex BLAS Level 1 Function from C"

The example below illustrates a call from a C program to the complex BLAS Level 1 function zdotc(). This function computes the dot product of two double-precision complex vectors.
In this example, the complex dot product is returned in the structure c.

```
#include "mkl.h"
#define N 5
int main()
{
int n = N, inca = 1, incb = 1, i;
MKL_Complex16 a[N], b[N], c;
for( i = 0; i < n; i++ ){
a[i].real = (double)i; a[i].imag = (double)i * 2.0;
b[i].real = (double)(n - i); b[i].imag = (double)i * 2.0;
}
zdotc( &c, &n, a, &inca, b, &incb );
printf( "The complex dot product is: ( %6.2f, %6.2f)\n", c.real, c.imag );
return 0;
}
```

need this include for intel library

note this is an array of double complex type

Example "Calling a Complex BLAS Level 1 Function from C++"

Below is the C++ implementation:

```
#include <complex>
```

This example illustrates the power of using C++ for scientific computing.

Example "Calling a Complex BLAS Level 1 Function from C++"

Below is the C++ implementation:

This example illustrates the power of using C++ for scientific computing. Standard includes like <complex>

this allows much more portable code as you can use standard C++ complex type in code and MKL will still work

```
#include <complex>
#include <iostream>
#define MKL_Complex16 std::complex<double>
#include "mkl.h"

#define N 5

int main()
{
        int n, inca = 1, incb = 1, i;
        std::complex<double> a[N], b[N], c;
        n = N;

        for( i = 0; i < n; i++ ){
                a[i] = std::complex<double>(i,i*2.0);
                b[i] = std::complex<double>(n-i,i*2.0);
        }
        zdotc(&c, &n, a, &inca, b, &incb );
        std::cout << "The complex dot product is: " << c << std::endl;
        return 0;
}
```

Example "Using CBLAS Interface Instead of Calling BLAS Directly from C"

This example uses CBLAS:

```
#include <stdio.h>
#include "mkl.h"
typedef struct{ double re; double im; } complex16;
#define N 5
int main()
{
int n, inca = 1, incb = 1, i;
complex16 a[N], b[N], c;
n = N;
for( i = 0; i < n; i++ ){
a[i].re = (double)i; a[i].im = (double)i * 2.0;
b[i].re = (double)(n - i); b[i].im = (double)i * 2.0;
}
cblas_zdotc_sub(n, a, inca, b, incb, &c );
printf( "The complex dot product is: ( %6.2f, %6.2f)\n", c.re, c.im );
return 0;
}
```

From <http://software.intel.com/sites/products/documentation/hpc/mkl/mkl_userguide_lnx/GUID-A0908E50-19D7-44C1-A068-44036B466BC7.htm#XREF_EXAMPLE_6_3_USING_CBLAS>

Full docs can be found at:
http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation

LAPACK - standard Linear Algebra package
- while there are several implementations (versions are included in intel MKL & AMD libraries) the calls are very standard and almost always follow fortran implementation (though callable from C/C++ just like BLAS).

Linear Equations

Two types of driver routines are provided for solving systems of linear equations:

- a **simple** driver (name ending -SV), which solves the system $AX = B$ by factorizing A and overwriting B with the solution X;
- an **expert** driver (name ending -SVX), which can also perform the following functions (some of them optionally):
  - solve $A^T X = B$ or $A^H X = B$ (unless A is symmetric or Hermitian);
  - estimate the condition number of A, check for near-singularity, and check for pivot growth;
  - refine the solution and compute forward and backward error bounds;
  - equilibrate the system if A is poorly scaled.

  The expert driver requires roughly twice as much storage as the simple driver in order to perform these extra functions.

Both types of driver routines can handle multiple right hand sides (the columns of *B*).
Different driver routines are provided to take advantage of special properties or storage schemes of the matrix *A*, as shown in Table 2.2.
These driver routines cover all the functionality of the computational routines for linear systems, except matrix inversion. It is seldom necessary to compute the inverse of a matrix explicitly, and it is certainly not recommended as a means of solving linear systems.

Type of matrix	Operation	Single precision		Double precision	
and storage scheme		real	complex	real	complex
general	simple driver	SGESV	CGESV	DGESV	ZGESV
	expert driver	SGESVX	CGESVX	DGESVX	ZGESVX
general band	simple driver	SGBSV	CGBSV	DGBSV	ZGBSV
	expert driver	SGBSVX	CGBSVX	DGBSVX	ZGBSVX
general tridiagonal	simple driver	SGTSV	CGTSV	DGTSV	ZGTSV
	expert driver	SGTSVX	CGTSVX	DGTSVX	ZGTSVX
symmetric/Hermitian	simple driver	SPOSV	CPOSV	DPOSV	ZPOSV
positive definite	expert driver	SPOSVX	CPOSVX	DPOSVX	ZPOSVX
symmetric/Hermitian	simple driver	SPPSV	CPPSV	DPPSV	ZPPSV
positive definite (packed storage)	expert driver	SPPSVX	CPPSVX	DPPSVX	ZPPSVX
symmetric/Hermitian	simple driver	SPBSV	CPBSV	DPBSV	ZPBSV
positive definite band	expert driver	SPBSVX	CPBSVX	DPBSVX	ZPBSVX
symmetric/Hermitian	simple driver	SPTSV	CPTSV	DPTSV	ZPTSV
positive definite tridiagonal	expert driver	SPTSVX	CPTSVX	DPTSVX	ZPTSVX
symmetric/Hermitian	simple driver	SSYSV	CHESV	DSYSV	ZHESV
indefinite	expert driver	SSYSVX	CHESVX	DSYSVX	ZHESVX
complex symmetric	simple driver		CSYSV		ZSYSV
	expert driver		CSYSVX		ZSYSVX
symmetric/Hermitian	simple driver	SSPSV	CHPSV	DSPSV	ZHPSV
indefinite (packed storage)	expert driver	SSPSVX	CHPSVX	DSPSVX	ZHPSVX
complex symmetric	simple driver		CSPSV		ZSPSV
(packed storage)	expert driver		CSPSVX		ZSPSVX

Table 2.2: Driver routines for linear equations

Pasted from <http://www.netlib.org/lapack/lug/node26.html>

Linear Equations

We use the standard notation for a system of simultaneous linear equations:

$A x = b$ (2.4)

where *A* is the **coefficient matrix**, *b* is the **right hand side**, and *x* is the **solution**. In (2.4) *A* is assumed to be a square matrix of order *n*, but some of the individual routines allow *A* to be rectangular. If there are several right hand sides, we write

$A X = B$ (2.5)

where the columns of *B* are the individual right hand sides, and the columns of *X* are the corresponding solutions. The basic task is to compute *X*, given *A* and *B*.
If *A* is upper or lower triangular, (2.4) can be solved by a straightforward process of backward or forward substitution. Otherwise, the solution is obtained after first factorizing *A* as a product of triangular matrices (and possibly also a diagonal matrix or permutation matrix).
The form of the factorization depends on the properties of the matrix *A*. LAPACK provides routines for the following types of matrices, based on the stated factorizations:

- **general** matrices (*LU* factorization with partial pivoting):
  $A = PLU$
  where *P* is a permutation matrix, *L* is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$), and U is upper triangular (upper trapezoidal if $m < n$).
- **general band** matrices including **tridiagonal** matrices (*LU* factorization with partial pivoting): If *A* is *m*-by-*n* with *kl* subdiagonals and *ku* superdiagonals, the factorization is
  $A = LU$
  where *L* is a product of permutation and unit lower triangular matrices with *kl* subdiagonals, and *U* is upper triangular with *kl*+*ku* superdiagonals.
- **symmetric and Hermitian positive definite** matrices including **band** matrices (Cholesky factorization):

$A = U^T U$ or $A = L L^T$ (symmetric)
$A = U^H U$ or $A = L L^H$ (Hermitian)

$A = U^T U$ or $A = L L^T$ (symmetric)

$A = U^H U$ or $A = L L^H$ (Hermitian)

where *U* is an upper triangular matrix and *L* is lower triangular.

- **symmetric and Hermitian positive definite tridiagonal** matrices (*L D* L^T factorization):

$A = U D U^T$ $A = L D L^T$ (symmetric)

$A = U D U^H$ $A = L D L^H$ (Hermmitian)

where *U* is a unit upper bidiagonal matrix, *L* is unit lower bidiagonal, and *D* is diagonal.

- **symmetric and Hermitian indefinite** matrices (symmetric indefinite factorization):

$A = U D U^T$ $A = L D L^T$ (symmetric)

$A = U D U^H$ $A = L D L^H$

where *U* (or *L*) is a product of permutation and unit upper (lower) triangular matrices, and *D* is symmetric and block diagonal with diagonal blocks of order 1 or 2.

The factorization for a general tridiagonal matrix is like that for a general band matrix with *kl* = 1 and *ku* = 1. The factorization for a symmetric positive definite band matrix with *k* superdiagonals (or subdiagonals) has the same form as for a symmetric positive definite matrix, but the factor *U* (or *L*) is a band matrix with *k* superdiagonals (subdiagonals). Band matrices use a compact band storage scheme described in section 5.3.3. LAPACK routines are also provided for symmetric matrices (whether positive definite or indefinite) using **packed** storage, as described in section 5.3.2.

While the primary use of a matrix factorization is to solve a system of equations, other related tasks are provided as well. Wherever possible, LAPACK provides routines to perform each of these tasks for each type of matrix and storage scheme (see Tables 2.7 and 2.8). The following list relates the tasks to the last 3 characters of the name of the corresponding computational routine:

xyyTRF:
factorize (obviously not needed for triangular matrices);

xyyTRS:
use the factorization (or the matrix *A* itself if it is triangular) to solve (2.5) by forward or backward substitution;

xyyCON:
estimate the reciprocal of the condition number
$\kappa(A) = \|A\|.\|A^{-1}\|$
; Higham's modification [63] of Hager's method [59] is used to estimate $|A^{-1}|$, except for symmetric positive definite tridiagonal matrices for which it is computed directly with comparable efficiency [61];

xyyRFS:
compute bounds on the error in the computed solution (returned by the xyyTRS routine), and refine the solution to reduce the backward error (see below);

xyyTRI:
use the factorization (or the matrix *A* itself if it is triangular) to compute A^{-1} (not provided for band matrices, because the inverse does not in general preserve bandedness);

xyyEQU:
compute scaling factors to equilibrate *A* (not provided for tridiagonal, symmetric indefinite, or triangular matrices). These routines do not actually scale the matrices: auxiliary routines xLAQyy may be used for that purpose -- see the code of the driver routines xyySVX for sample usage.

Note that some of the above routines depend on the output of others:

xyyTRF:
may work on an equilibrated matrix produced by xyyEQU and xLAQyy, if yy is one of {GE, GB, PO, PP, PB};

xyyTRS:
requires the factorization returned by xyyTRF;

xyyCON:
requires the norm of the original matrix *A*, and the factorization returned by xyyTRF;

xyyRFS:
requires the original matrices *A* and *B*, the factorization returned by xyyTRF, and the solution *X* returned by xyyTRS;

xyyTRI:
requires the factorization returned by xyyTRF.

The RFS (``refine solution'') routines perform iterative refinement and compute backward and forward error bounds for the solution. Iterative refinement is done in the same precision as the input data. In particular, the residual is *not* computed with extra precision, as has been traditionally done. The benefit of this procedure is discussed in Section 4.4.

Type of matrix	Operation	Single precision		Double precision	
and storage scheme		real	complex	real	complex
general	factorize	SGETRF	CGETRF	DGETRF	ZGETRF
	solve using factorization	SGETRS	CGETRS	DGETRS	ZGETRS
	estimate condition number	SGECON	CGECON	DGECON	ZGECON
	error bounds for solution	SGERFS	CGERFS	DGERFS	ZGERFS

	invert using factorization	SGETRI	CGETRI	DGETRI	ZGETRI
	equilibrate	SGEEQU	CGEEQU	DGEEQU	ZGEEQU
general	factorize	SGBTRF	CGBTRF	DGBTRF	ZGBTRF
band	solve using factorization	SGBTRS	CGBTRS	DGBTRS	ZGBTRS
	estimate condition number	SGBCON	CGBCON	DGBCON	ZGBCON
	error bounds for solution	SGBRFS	CGBRFS	DGBRFS	ZGBRFS
	equilibrate	SGBEQU	CGBEQU	DGBEQU	ZGBEQU
general	factorize	SGTTRF	CGTTRF	DGTTRF	ZGTTRF
tridiagonal	solve using factorization	SGTTRS	CGTTRS	DGTTRS	ZGTTRS
	estimate condition number	SGTCON	CGTCON	DGTCON	ZGTCON
	error bounds for solution	SGTRFS	CGTRFS	DGTRFS	ZGTRFS
symmetric/Hermitian	factorize	SPOTRF	CPOTRF	DPOTRF	ZPOTRF
positive definite	solve using factorization	SPOTRS	CPOTRS	DPOTRS	ZPOTRS
	estimate condition number	SPOCON	CPOCON	DPOCON	ZPOCON
	error bounds for solution	SPORFS	CPORFS	DPORFS	ZPORFS
	invert using factorization	SPOTRI	CPOTRI	DPOTRI	ZPOTRI
	equilibrate	SPOEQU	CPOEQU	DPOEQU	ZPOEQU
symmetric/Hermitian	factorize	SPPTRF	CPPTRF	DPPTRF	ZPPTRF
positive definite	solve using factorization	SPPTRS	CPPTRS	DPPTRS	ZPPTRS
(packed storage)	estimate condition number	SPPCON	CPPCON	DPPCON	ZPPCON
	error bounds for solution	SPPRFS	CPPRFS	DPPRFS	ZPPRFS
	invert using factorization	SPPTRI	CPPTRI	DPPTRI	ZPPTRI
	equilibrate	SPPEQU	CPPEQU	DPPEQU	ZPPEQU
symmetric/Hermitian	factorize	SPBTRF	CPBTRF	DPBTRF	ZPBTRF
positive definite	solve using factorization	SPBTRS	CPBTRS	DPBTRS	ZPBTRS
band	estimate condition number	SPBCON	CPBCON	DPBCON	ZPBCON
	error bounds for solution	SPBRFS	CPBRFS	DPBRFS	ZPBRFS
	equilibrate	SPBEQU	CPBEQU	DPBEQU	ZPBEQU
symmetric/Hermitian	factorize	SPTTRF	CPTTRF	DPTTRF	ZPTTRF
positive definite	solve using factorization	SPTTRS	CPTTRS	DPTTRS	ZPTTRS
tridiagonal	estimate condition number	SPTCON	CPTCON	DPTCON	ZPTCON
	error bounds for solution	SPTRFS	CPTRFS	DPTRFS	ZPTRFS

Table 2.7: Computational routines for linear equations

Type of matrix	Operation	Single precision		Double precision	
and storage scheme		real	complex	real	complex
symmetric/Hermitian	factorize	SSYTRF	CHETRF	DSYTRF	ZHETRF
indefinite	solve using factorization	SSYTRS	CHETRS	DSYTRS	ZHETRS
	estimate condition number	SSYCON	CHECON	DSYCON	ZHECON
	error bounds for solution	SSYRFS	CHERFS	DSYRFS	ZHERFS
	invert using factorization	SSYTRI	CHETRI	DSYTRI	ZHETRI
complex symmetric	factorize		CSYTRF		ZSYTRF
	solve using factorization		CSYTRS		ZSYTRS
	estimate condition number		CSYCON		ZSYCON
	error bounds for solution		CSYRFS		ZSYRFS
	invert using factorization		CSYTRI		ZSYTRI
symmetric/Hermitian	factorize	SSPTRF	CHPTRF	DSPTRF	ZHPTRF
indefinite	solve using factorization	SSPTRS	CHPTRS	DSPTRS	ZHPTRS
(packed storage)	estimate condition number	SSPCON	CHPCON	DSPCON	ZHPCON
	error bounds for solution	SSPRFS	CHPRFS	DSPRFS	ZHPRFS
	invert using factorization	SSPTRI	CHPTRI	DSPTRI	ZHPTRI
complex symmetric	factorize		CSPTRF		ZSPTRF
(packed storage)	solve using factorization		CSPTRS		ZSPTRS
	estimate condition number		CSPCON		ZSPCON
	error bounds for solution		CSPRFS		ZSPRFS
	invert using factorization		CSPTRI		ZSPTRI
triangular	solve	STRTRS	CTRTRS	DTRTRS	ZTRTRS
	estimate condition number	STRCON	CTRCON	DTRCON	ZTRCON
	error bounds for solution	STRRFS	CTRRFS	DTRRFS	ZTRRFS

	invert	STRTRI	CTRTRI	DTRTRI	ZTRTRI
triangular	solve	STPTRS	CTPTRS	DTPTRS	ZTPTRS
(packed storage)	estimate condition number	STPCON	CTPCON	DTPCON	ZTPCON
	error bounds for solution	STPRFS	CTPRFS	DTPRFS	ZTPRFS
	invert	STPTRI	CTPTRI	DTPTRI	ZTPTRI
triangular	solve	STBTRS	CTBTRS	DTBTRS	ZTBTRS
band	estimate condition number	STBCON	CTBCON	DTBCON	ZTBCON
	error bounds for solution	STBRFS	CTBRFS	DTBRFS	ZTBRFS

Table 2.8: Computational routines for linear equations (continued)

Pasted from <http://www.netlib.org/lapack/lug/node38.html>

Band Storage

An ***m***-by-***n*** band matrix with ***kl*** subdiagonals and ***ku*** superdiagonals may be stored compactly in a two-dimensional array with ***kl+ku+1*** rows and ***n*** columns. Columns of the matrix are stored in corresponding columns of the array, and diagonals of the matrix are stored in rows of the array. This storage scheme should be used in practice only if

$$kl, ku \ll \min(m, n)$$

, although LAPACK routines work correctly for all values of ***kl*** and ***ku***. In LAPACK, arrays that hold matrices in band storage have names ending in `B'.

To be precise, a_{ij} is stored in AB(***ku+1+i-j,j***) for

$$\max(1, j - ku) \le i \le \min(m, j + kl)$$

. For example, when ***m*** = ***n*** = **5**, ***kl*** = **2** and ***ku*** = **1**:

Band matrix ***A***	Band storage in array AB
$\begin{pmatrix} a_{11} & a_{12} & & & \\ a_{21} & a_{22} & a_{23} & & \\ a_{31} & a_{32} & a_{33} & a_{34} & \\ & a_{42} & a_{43} & a_{44} & a_{45} \\ & & a_{53} & a_{54} & a_{55} \end{pmatrix}$	$\begin{pmatrix} * & a_{12} & a_{23} & a_{34} & a_{45} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & * \\ a_{31} & a_{42} & a_{53} & * & * \end{pmatrix}$

The elements marked

*

in the upper left and lower right corners of the array AB need not be set, and are not referenced by LAPACK routines.

Note: when a band matrix is supplied for ***LU*** factorization, space must be allowed to store an additional ***kl*** superdiagonals, generated by fill-in as a result of row interchanges. This means that the matrix is stored according to the above scheme, but with ***kl*** + ***ku*** superdiagonals.

Triangular band matrices are stored in the same format, with either ***kl*** = **0** if upper triangular, or ***ku*** = **0** if lower triangular.

For symmetric or Hermitian band matrices with ***kd*** subdiagonals or superdiagonals, only the upper or lower triangle (as specified by UPLO) need be stored:

- if UPLO = `U', a_{ij} is stored in AB(***kd+1+i-j,j***) for ;
- if UPLO = `L', a_{ij} is stored in AB(***1+i-j,j***) for .

For example, when ***n*** = **5** and ***kd*** = **2**:

UPLO	Hermitian band matrix ***A***	Band storage in array AB
`U'	$\begin{pmatrix} a_{11} & a_{12} & a_{13} & & \\ \bar{a}_{12} & a_{22} & a_{23} & a_{24} & \\ \bar{a}_{13} & \bar{a}_{23} & a_{33} & a_{34} & a_{35} \\ & \bar{a}_{24} & \bar{a}_{34} & a_{44} & a_{45} \\ & & \bar{a}_{35} & \bar{a}_{45} & a_{55} \end{pmatrix}$	$\begin{pmatrix} * & * & a_{13} & a_{24} & a_{35} \\ * & a_{12} & a_{23} & a_{34} & a_{45} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \end{pmatrix}$
`L'	$\begin{pmatrix} a_{11} & \bar{a}_{21} & \bar{a}_{31} & & \\ a_{21} & a_{22} & \bar{a}_{32} & \bar{a}_{42} & \\ a_{31} & a_{32} & a_{33} & \bar{a}_{43} & \bar{a}_{53} \\ & a_{42} & a_{43} & a_{44} & \bar{a}_{54} \\ & & a_{53} & a_{54} & a_{55} \end{pmatrix}$	$\begin{pmatrix} a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & * \\ a_{31} & a_{42} & a_{53} & * & * \end{pmatrix}$

Pasted from <http://www.netlib.org/lapack/lug/node124.html>

Similar to BLAS you need to link to the lapack libraries when compiling with a flag like -llapack. Generally, they are included in the Intel MKL libraries or AMD ACML libraries so linking with -lmkl or similar will pick up lapack & blas.