Dense Linear Algebra Exercises

1 LU Factorisation

The exercise is to solve a set of N linear equations by performing an LU factorisation of the matrix A. You are supplied with template codes in `lufact.tar` which you should download and then unpack as follows:

```
user$ tar xvf lufact.tar
```

You should change to the directory appropriate for the language you want to use (C, Fortran or Java) and work from there. We suggest you run these exercises on Cirrus and use the Intel compilers (`ifort` and `icc` for Fortran and C respectively). This is because it's easier to link in Lapack which you will need later. First load the following two modules:

```
module load intel-compilers-18
module load intel-tools-18
```

Compiling should then be straightforward:

```
ifort -o lufact lufact.f90
```

or

```
icc -o lufact lufact.c
```

As provided, the codes should compile and run as they stand but not perform any useful computation. However, they do produce diagnostics that will be useful when you are debugging your own code so you should check that you understand the program output. For values of N larger than 6 most of the output is suppressed so you should use small values of N when developing your code. However, once your code is working you should use larger values of N, for example N = 10, 50, 100, 200, 500 or 1000.

1.1 Exercise

The actual algorithms are described in Section 4, along with some explanatory text describing various important practical issues associated with implementing them in a real program.

(a) The template code initialises a random $N \times N$ matrix A and a vector b_i that is specifically constructed so that the solution is given by $x_i = 1$. Set $N = 6$ and then insert code to create the LU factors of A. Check that their product (automatically printed to the screen) is equal to A.

(b) Now use forward and backward substitution to solve $Ax = b$ and look at the elements of x. Do you get the correct answer $x_i = 1$? Is the product Ax equal to b as required?

(c) Compute the residual from the norm of the residue vector $r = b - Ax$ and the norm of b. How does this change as you increase N?

(d) In general the solution is not known in advance so it is not possible to check the correctness of the solution vector x directly. However, in this particular test case, we know the solution is $x_i = 1$. Calculate the average error per element by computing $|x - 1|/N$. How does this compare to the residual? What happens as you increase N?

1.2 Extra exercises

(a) What is the improvement in accuracy from using double precision?

(b) How does the rounding mode affect the error (look at systems with small and large residuals)?

(c) Implement iterative improvement and see by how much it reduces the residue.

(d) Make a plot of how the residue varies with N.

(e) Time your program and see if you can understand how it varies with N

2 Using the LAPACK library

The exercise is to perform the same LU factorisation and solution as before but to do it using routines from the LAPACK library rather than writing your own code. You should keep a copy of your solution to the previous exercise so you can compare the two approaches.

For Fortran and C you must include the LAPACK and BLAS libraries using

```
-mkl
```

at link time. This gives you highly optimised, numerically stable library routines. The libraries are written for Fortran: C programmers should consult the lecture slides to see how to call them appropriately.

For Java you will have to unpack a jar file containing generic implementations of the LAPACK routines. These will be numerically stable but not necessarily optimised in terms of performance. For full instructions see `javaLAPACK.txt`.

2.1 Exercise

(a) Call the LAPACK factorisation and solution routines to solve $Ax = b$ and compare the results for $N = 6$ with your own implementation. The LU decomposition and forward-backward substitution routines are called sgetrf and sgetrs respectively. For more details see the lecture slides or search online.

(b) Compare the residuals for the same N and note if you see any improvement due to partial pivoting (eg reduction in the residual and/or a smaller error in x). How does this change as you increase N?

(c) Measure the execution times of both codes and compute the increase in performance from using library routines (you should use large values of N). Can you quantify this in terms of Mflops?

2.2 Extra Exercises

(a) You will see an extra file called `matcondgen.f90` or `matcondgen.c`. This contains a routine similar to matgen for generating a random matrix A, but it is now constructed to have a userspecified condition number. The value of the condition number is supplied as the first argument — it should be called as follows in Fortran and C:

```
real cond = 1.0
...
call matcondgen(cond, a, nmax, n, b)

float cond = 1.0;
...
matcondgen(cond, n, a, b);
```

The code in matcondgen uses LAPACK routines so you must include the LAPACK library when compiling This is done by including the -mkl flag as mentioned above. Note that the Fortran files can be compiled stand-alone, and then linked e.g.

```
ifort -c -mkl -o lufact.o lufact.f90
ifort -c -mkl -o matcondgen.o matcondgen.f90
ifort -mkl -o lufact lufact.o matcondgen.o
```

whereas the C routine must be included in your main program as it needs to know the value of the constant `NMAX`. You should be able to compile and link the code in one go once you have added the matcondgen code to the lufact.c file:

```
icc -mkl -o lufact lufact.c
```

Use the new routine to generate A and run your program with a large value of N (eg 100 or more) for a variety of condition numbers (eg 10.0, 100.0, 1000.0, . . .). How do the residual and error in the solution vary as the equations become more difficult to solve? Does partial pivoting become more important (ie how much more accurate is the LAPACK solution compared to your own code)?

3 Running with ScaLAPACK

The following exercises give you an opportunity to run the same algorithms as for the previous exercise, but this time in parallel. The code is already written and there should be no need to make any changes to the code itself. There is only a Fortran version but it should be easy enough to follow even if you are less familar with Fortran than C. The code contains several new routines and variables to allow the scalapack library to work but is otherwise basically the same as for the serial version. Small changes have been made to make the array sizes dynamic and there are a few small changes to the `matgen` routine. The code also runs in both single and double precision which can be changed by setting `wp` to either `sp` or `dp` respectively at the start of the code.

The code runs as follows (see scripts for what goes before `lufact`):

```
./lufact matsize nprocrow nproccol blocksize
```

where `matsize` is the global matrix size, `nprocrow` and `nproccol` are the number or rows and columns respectively in the processor grid and `blocksize` equals the size of the blocks in the block-cyclic distribution. The block size is set to be the same in the row and column dimensions so the majority of the blocks should be square. Note it is your job to ensure that `nprocrow` X nproccol = nprocs (the total number of processors you run on).

Take a copy of `lufact-scalapack.tar` and unpack. See the `README` file for instructions for building the code.

Note that, as before, the code is designed to give a solution of $x_i = 1$. Rather than print out the entire x array, the code prints out the value of $|x - \mathbf{1}|$, where $\mathbf{1}$ is a vector with each element equal to 1.

3.1 Exercises

(a) Take a look at the code and compare how the scalapack library calls compare with your earlier lapack implementation.

(b) Submit the code to Cirrus using the batch script provided.

(c) Try running the code using different block sizes (eg 8, 16, 32, 64, 128, 256). This can be done with consecutive mpiexec commands in the batch file. What is the optimal block size for this particular problem? Can you give likely reasons for the performance when using different block sizes in terms of the parallelisation?

4 Algorithms

The L and U factors of a matrix A can be computed as follows. This is essentially Crout's algorithm but for simplicity there is no pivoting, and we store each of the factors separately rather than performing the decomposition in-place.

Note that, for convenience, all arrays are statically declared in the template codes to be much larger than required (of size `NMAX` by `NMAX`); the actual problem size is determined by the value of N.

First, initialise the factors:

- set L to the unit matrix: $l_{ii} = 1$; $l_{ij} = 0$ for $i \neq j$
- set U to zero: $u_{ij} = 0$

The rest of the iterative procedure is then as follows:

```
• loop over $j = 1, 2, \ldots, N$
    – loop over $i = 1, 2, \ldots, j$
        ∗ set $u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}$
    – end loop over $i$
    – loop over $i = j+1, j+2 \ldots, N$
        ∗ set $l_{ij} = \frac{1}{u_{jj}} \left( a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right)$
    – end loop over $i$
• end loop over $j$
```

Having obtained the LU factorisation of A, the solution to $Ax = b$ can be found by first solving $Ly = b$

```
• loop over $i = 1, 2, \ldots, N$
    – set $y_i = \frac{1}{l_{ii}} \left( b_i - \sum_{j=1}^{i-1} l_{ij} y_j \right)$
• end loop over $i$
```

followed by solving $Ux = y$

```
• loop over $i = N, N-1, \ldots, 1$
    – set $x_i = \frac{1}{u_{ii}} \left( y_i - \sum_{j=i+1}^{N} u_{ij} x_j \right)$
• end loop over $i$
```

Notes for C and Java programmers

The above algorithm is written using the mathematical format for matrix indices i and j which run from 1 to N. This is convenient in Fortran as the language uses the same convention for arrays. However, in C and Java, indices run from 0 to $N-1$ so some extra care is required when setting the limits of loops.

If a loop has a constant upper or lower limit (eg 1 or N), subtract one from each value; if a loop has a limit that depends on another index (eg i or j) then it should stay unchanged.

Also note that for a loop of the form

- loop over $i = 1, 2, \ldots, j$

you must make sure that you execute the final iteration of the loop where $i = j$. The way that loops are conventionally written in C and Java makes it easy to accidentally omit this last iteration.