AMath 483/583 — Lecture 12 — April 22, 2011

Today:

- LAPACK and BLAS
- Parallel computing concepts

Monday:

- OpenMP

Read: Class notes and references

Notes:

R.J. LeVeque, University of Washington AMath 483/583, Lecture 12, April 22, 2011

The BLAS

Basic Linear Algebra Subroutines

Core routines used by LAPACK (Linear Algebra Package) and elsewhere.

Generally optimized for particular machine architectures, cache hierarchy.

Can create optimized BLAS using ATLAS (Automatically Tuned Linear Algebra Software)

See notes and `http://www.netlib.org/blas/faq.html`

- Level 1: Scalar and vector operations
- Level 2: Matrix-vector operations
- Level 3: Matrix-matrix operations

R.J. LeVeque, University of Washington AMath 483/583, Lecture 12, April 22, 2011

Notes:

R.J. LeVeque, University of Washington AMath 483/583, Lecture 12, April 22, 2011

The BLAS

Subroutine names start with:

- S: single precision
- D: double precision
- C: single precision complex
- Z: double precision complex

Examples:

- DDOT: dot product of two vectors
- DGEMV: matrix-vector multiply, general matrices
- DGEMM: matrix-matrix multiply, general matrices
- DSYMM: matrix-matrix multiply, symmetric matrices

R.J. LeVeque, University of Washington AMath 483/583, Lecture 12, April 22, 2011

Notes:

R.J. LeVeque, University of Washington AMath 483/583, Lecture 12, April 22, 2011

LAPACK

Many routines for linear algebra.

Typical name: XYYZZZ

X is precision

YY is type of matrix, e.g. GE (general), BD (bidiagonal),

ZZZ is type of operation, e.g. SV (solve system),
EV (eigenvalues, vectors), SVD (singular values, vectors)

Notes:

Installing LAPACK

On Virtual Machine or other Debian or Ubuntu Linux:

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

This will include BLAS (but not optimized for your system).

Alternatively can download tar files and compile.

Notes:

Using libraries

If `program.f90` uses BLAS routines...

```
$ gfortran -c program.f90
$ gfortran -lblas program.o
```

or can combine as

```
$ gfortran -lblas program.f90
```

When linking together `.o` files, will look for a file called `libblas.a` (probably in `/usr/lib`).

This is a archived static library.

Notes:

Making blas library

Download **http://www.netlib.org/blas/blas.tgz**.

Put this in desired location, e.g. `$HOME/lapack/blas.tgz`

```
$ cd $HOME/lapack
$ tar -zxf blas.tgz    # creates BLAS subdirect
$ cd BLAS
$ gfortran -O3 -c *.f
$ ar cr libblas.a *.o  # creates libblas.a
```

To use this library:

```
$ gfortran -lblas -L$HOME/lapack/BLAS  \
           program.f90
```

Note: Non-optimized Fortran 77 versions.

Better approach would be to use ATLAS.

Notes:

Creating LAPACK library

Can be done from source at
http://www.netlib.org/lapack/

but somewhat more difficult.

Individual routines and dependencies can be obtained from e.g.:

http://www.netlib.org/lapack/double

Download `.tgz` file and untar into directory where you want to use them, or make a library of just these files.

Some routines are in
`$CLASSHG/codes/lapack/lapack-subset`.

Notes:

Memory management for arrays

Often a program needs to be written to handle arrays whose size is not known until the program is running.

Fortran 77 approaches:

- Allocate arrays large enough for any application,
- Use “work arrays” that are partitioned into pieces.

We will look at some examples from LAPACK since you will probably see this in other software!

The good news:

Fortran 90 allows dynamic memory allocation.

Notes:

DGESV — Solves a general linear system

http://www.netlib.org/lapack/double/dgesv.f

```
      SUBROUTINE DGESV( N, NRHS, A, LDA, IPIV,
     $                  B, LDB, INFO )
```

`N` = size of system (square)

`A` = matrix on input, L,U factors on output,
`dimension(LDA,N)`

`LDA` = leading dimension of A
(number of columns in declaration of A)

```
real(kind=8) dimension(100,500) :: a
! fill a(1:20, 1:20) with 20x20 matrix
n = 20
lda = 100
```

Need this to index into `a(i,j) = (j-1)*lda + i`
(stored by columns)

Notes:

DGESV — Solves a general linear system

```
      SUBROUTINE DGESV( N, NRHS, A, LDA, IPIV,
     $                  B, LDB, INFO )
```

`NRHS` = number of right hand sides

`IPIV` = Returns pivot vector (permutation of rows)
`integer, dimension(N)`
Row `I` was interchanged with row `IPIV(I)`.

`B` = matrix whose columns are right hand side(s) on input
solution vector(s) on output.

`LDB` = leading dimension of `B`.

`INFO` = integer returning 0 if successful.

Notes:

Gaussian elimination as factorization

If A is nonsingular it can be factored as

$$PA = LU$$

where

P is a permutation matrix (rows of identity permuted),

L is lower triangular with 1's on diagonal,

U is upper triangular.

After returning from `dgesv`:
`A` contains L and U (without the diagonal of L),
`IPIV` gives ordering of rows in P.

Notes:

Gaussian elimination as factorization

Example:

$$A = \begin{bmatrix} 2 & 1 & 3 \\ 4 & 3 & 6 \\ 2 & 3 & 4 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 2 & 1 & 3 \\ 4 & 3 & 6 \\ 2 & 3 & 4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1 & 0 \\ 1/2 & -1/3 & 1 \end{bmatrix} \begin{bmatrix} 4 & 3 & 6 \\ 0 & 1.5 & 1 \\ 0 & 0 & 1/3 \end{bmatrix}$$

`IPIV` = (2,3,1)

and `A` ends up as

$$\begin{bmatrix} 4 & 3 & 6 \\ 1/2 & 1.5 & 1 \\ 1/2 & -1/3 & 1/3 \end{bmatrix}$$

Notes:

`dgesv` examples

See `$CLASSHG/codes/lapack/random`.

`randomsys1.f90` is with static array allocation.

`randomsys2.f90` is with dynamic array allocation.

Notes:

Parallel Computing

- Basic concepts
- Shared vs. distributed memory
- OpenMP (shared)
- MPI (shared or distributed)

Notes:

Some general references

[Lin-Snyder] C. Lin and L. Snyder, *Principles of Parallel Programming*, 2008.

[Scott-Clark-Bagheri] L. R. Scott, T. Clark, B. Bagheri, *Scientific Parallel Computing*, Princeton University Press, 2005.

Several good tutorials available from National Labs:

Livermore:
`https://computing.llnl.gov/?set=training&page=index`

NERSC: `http://www.nersc.gov/nusers/help/tutorials/`

Notes:

Increasing speed

Moore's Law: Processor speed doubles every 18 months.
$\Longrightarrow$ factor of 1024 in 15 years.

Going forward: Number of cores doubles every 18 months.

Top: Total computing power of top 500 computers

Middle: #1 computer

Bottom: #500 computer

`http://www.top500.org`

Notes:

Parallel processing

Two major classes:

Shared memory:

All processors have access to the same memory.

Multicore chip: separate L1 caches, L2 might be shared.

Distributed memory:

Each processor has it's own memory and caches.

Transferring data between processors is slow.

E.g., clusters of computers, supercomputers

Hybrid: Often clusters of multicore machines!

Notes:

Multi-thread computing

For example, multi-threaded program on dual-core computer.

Thread:

A thread of control: program code, program counter, call stack, small amount of thread-specific data (registers, L1 cache).

Shared memory and file system.

Threads may be spawned and destroyed as computation proceeds.

Languages like OpenMP.

Notes:

R.J. LeVeque, University of Washington AMath 483/583, Lecture 12, April 22, 2011

POSIX Threads

Portable Operating System Intefrace

Standardized C language threads programming interface

For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995).

Implementations adhering to this standard are referred to as POSIX threads, or Pthreads.

Notes:

R.J. LeVeque, University of Washington AMath 483/583, Lecture 12, April 22, 2011

Multi-thread computing

Some issues:

Limited to modest number of cores when memory is shared.

Multiple threads have access to same data — convenient and fast.

Contention: But, need to make sure they don't conflict (e.g. two threads should not write to same location at same time).

Dependencies, synchronization: Need to make sure some operations are done in proper order!

May need cache coherence: If Thread 1 changes x in its private cache, other threads might need to see changed value.

Notes:

R.J. LeVeque, University of Washington AMath 483/583, Lecture 12, April 22, 2011

Multi-process computing

A process is a thread that also has its own private address space.

Multiple processes are often running on a single computer (e.g. different independent programs).

For distributed memory parallel computers, a single computation must be tackled with multiple processes because of memory layout.

Larger cost in creating and destroying processes.

Greater latency in sharing data.

Processes communicate by passing messages.

Languages like MPI — Message Passing Interface.

Notes:

R.J. LeVeque, University of Washington AMath 483/583, Lecture 12, April 22, 2011

Multi-process computing with distributed memory

Some issues:

Often more complicated to program.

High cost of data communication between processes.
Want to maximize processing on local data relative to communication with other processes.

Often need to partition problem domain into subdomains, (e.g. domain decomposition for PDEs)

Generally requires coarse grain parallelism.

Notes:

R.J. LeVeque, University of Washington AMath 483/583, Lecture 12, April 22, 2011

Amdahl's Law

Typically only part of a computation can be parallelized.

Suppose 50% of the computation is inherently sequential, and the other 50% can be parallelized.

Question: How much faster could the computation potentially run on many processors?

Answer: At most a factor of 2, no matter how many processors.

The sequential part is taking half the time and that time is still required even if the parallel part is reduced to zero time.

Notes:

R.J. LeVeque, University of Washington AMath 483/583, Lecture 12, April 22, 2011

Amdahl's Law

Suppose 10% of the computation is inherently sequential, and the other 90% can be parallelized.

Question: How much faster could the computation potentially run on many processors?

Answer: At most a factor of 10, no matter how many processors.

The sequential part is taking 1/10 of the time and that time is still required even if the parallel part is reduced to zero time.

Notes:

Amdahl's Law

Suppose $1/S$ of the computation is inherently sequential, and the other $(1 - 1/S)$ can be parallelized.

Then can gain at most a factor of S, no matter how many processors.

If T_S is the time required on a sequential machine and we run on P processors, then the time required will be (at least):

$$T_P = (1/S)T_S + (1 - 1/S)T_S/P$$

Note that

$$T_P \to (1/S)T_S \quad \text{as} \quad P \to \infty$$

Notes:

Amdahl's Law

Suppose $1/S$ of the computation is inherently sequential $\implies$

$$T_P = (1/S)T_S + (1 - 1/S)T_S/P$$

Example: If 5% of the computation is inherently sequential ($S = 20$), then the reduction in time is:

P	T_P
1	T_S
2	$0.525T_S$
4	$0.288T_S$
32	$0.080T_S$
128	$0.057T_S$
1024	$0.051T_S$

Notes:

Speedup

The ratio T_S/T_P of time on a sequential machine to time running in parallel is the speedup.

This is generally less than P for P processors.
Perhaps much less.

Amdahl's Law plus overhead costs of starting processes/threads, communication, etc.

Caveat: May (rarely) see speedup greater than P...
For example, if data doesn't all fit in one cache
but does fit in the combined caches of multiple processors.

Notes:

Scaling

Some algorithms scale better than others as the number of processors increases.

Typically interested on how well algorithms work for large problems requiring lots of time, e.g.
Particle methods for n particles,
algorithms for solving systems of n equations,
algorithms for solving PDEs on $n \times n \times n$ grid in 3D,

For large n, there may be lots of inherent parallelism.

But depends on many factors:
dependencies between calculations,
communication as well as flops,
nature of problem and algorithm chosen.

Notes:

Scaling

Typically interested on how well algorithms work for large problems requiring lots of time.

Strong scaling: How does the algorithm perform as the number of processors P increases for a fixed problem size n?

Any algorithm will eventually break down (consider $P > n$)

Weak scaling: How does the algorithm perform when the problem size increases with the number of processors?

E.g. If we double the number of processors can we solve a problem "twice as large" in the same time?

Notes:

Weak scaling

What does “twice as large” mean?

Depends on how algorithm complexity scales with n.

Example: Solving linear system with Gaussian elimination requires $O(n^3)$ flops.

Doubling n requires 8 times as many operations.

Problem is “twice as large” if we increase n by a factor of $2^{1/3} \approx 1.26$.

Notes:

R.J. LeVeque, University of Washington AMath 483/583, Lecture 12, April 22, 2011

Weak scaling

Solving steady state heat equation on $n \times n \times n$ grid.

n^3 grid points $\implies$ linear system with this many unknowns.

If we used Gaussian elimination (very bad idea!) we would require $\sim (n^3)^3 = n^9$ flops.

Doubling n would require $2^9 = 512$ times more flops.

Good iterative methods can do the job in $O(n^3)\log_2(n)$ work or less. (e.g. multigrid).

Developing better algorithms is as important as better hardware!!

R.J. LeVeque, University of Washington AMath 483/583, Lecture 12, April 22, 2011

Notes:

R.J. LeVeque, University of Washington AMath 483/583, Lecture 12, April 22, 2011

Speedup for problems like steady state heat equation

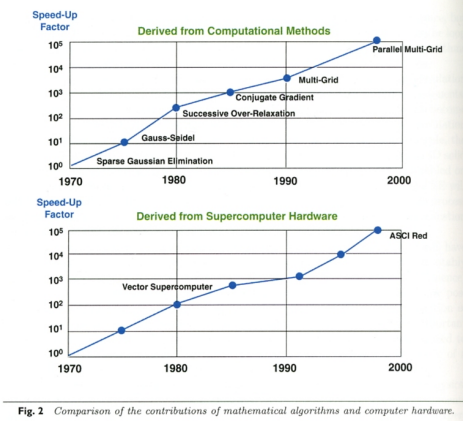

Fig. 2 *Comparison of the contributions of mathematical algorithms and computer hardware.*

Source: SIAM Review

R.J. LeVeque, University of Washington AMath 483/583, Lecture 12, April 22, 2011

Notes:

R.J. LeVeque, University of Washington AMath 483/583, Lecture 12, April 22, 2011