Dense matrix algebra and libraries (and dealing with Fortran)

CPS343

Parallel and High Performance Computing

Spring 2020

Outline

1. HPC == numerical linear algebra?
   - The role of linear algebra in HPC
   - Numerical linear algebra
2. Matrix-matrix products
   - Rowwise matrix-matrix multiplication
   - Recursive block oriented matrix-matrix multiplication
3. Dense Matrix Linear Algebra
   - BLAS
   - LAPACK
   - ATLAS

Outline

HPC == numerical linear algebra?

Perhaps the largest single group of HPC applications are devoted to solving problems described by differential equations.

- In these problems there is typically a *domain*, which may be space, time, both of these, or something else altogether
- The domain is *discretized*, resulting a system of equations that must be solved
- These equations are often linear so techniques from numerical linear algebra are used to solve them
- When the equations are nonlinear we may be able to *linearize* them (treat them as linear on a portion of the domain) to get the solution part-by-part

HPC != numerical linear algebra?

Some of the problems that can lead to differential equations and linear systems as well as others that don't involve differential equations can be be approached in completely different ways.

Two important examples:

1. simulation by Monte Carlo methods
2. dealing with data: searching and sorting

Numerical linear algebra libraries

- As we've seen in the case of matrix-matrix multiplication, standard "textbook" algorithms are often not the most efficient implementations
- In addition, they may not be the most "correct" in the sense of keeping errors as small as possible
- Since it's important to "get it right" **and** "do it fast," its nearly always advisable to use numerical libraries of proven code to handle linear algebra operations
- Today we'll focus on a few important libraries. There are many others, and you should always look for available appropriate libraries that can help as part of tackling any programming problem

Outline

Vector operations

Key operations yielding vectors include:

- copy one vector to another: $\mathbf{u} \leftarrow \mathbf{v}$
- swap two vectors: $\mathbf{u} \rightleftharpoons \mathbf{v}$
- scale a vector by a constant: $\mathbf{u} \leftarrow \alpha\mathbf{u}$
- add a multiple of one vector to another: $\alpha\mathbf{u} + \mathbf{v}$

Key operations yielding scalars include:

- inner product: $\mathbf{u} \cdot \mathbf{v} = \sum_i u_i v_i$
- 1-norm of a vector: $||\mathbf{u}||_1 = \sum_i |u_i|$
- 2-norm of a vector: $||\mathbf{u}||_2 = \sqrt{\sum_i u_i^2}$
- find maximal entry in a vector: $||\mathbf{u}||_\infty = \max_i |u_i|$

Matrix-vector operations

Key operations between matrices and vectors include:

- general (dense) or sparse matrix-vector multiplication: $A\mathbf{x}$
- rank-1 update: (adds a scalar multiple of $\mathbf{x}\mathbf{y}^T$ to a matrix): $A + \alpha\mathbf{x}\mathbf{y}^T$
- solution of matrix-vector equations: $A\mathbf{x} = \mathbf{b}$

Matrix-matrix operations

Key operations between pairs of matrices include:

- general matrix-matrix multiplication
- symmetric matrix-matrix multiplication
- sparse matrix-matrix multiplication

Key operations on a single matrix include:

- factoring/decomposing a matrix: $A = LU$, $A = QR$, $A = U\Sigma V^*$
- finding the eigenvalues and eigenvectors of a matrix

Outline

Rowwise Matrix-Matrix Multiplication

The rowwise matrix-matrix product for $C = AB$ is computed with the pseudo-code below.

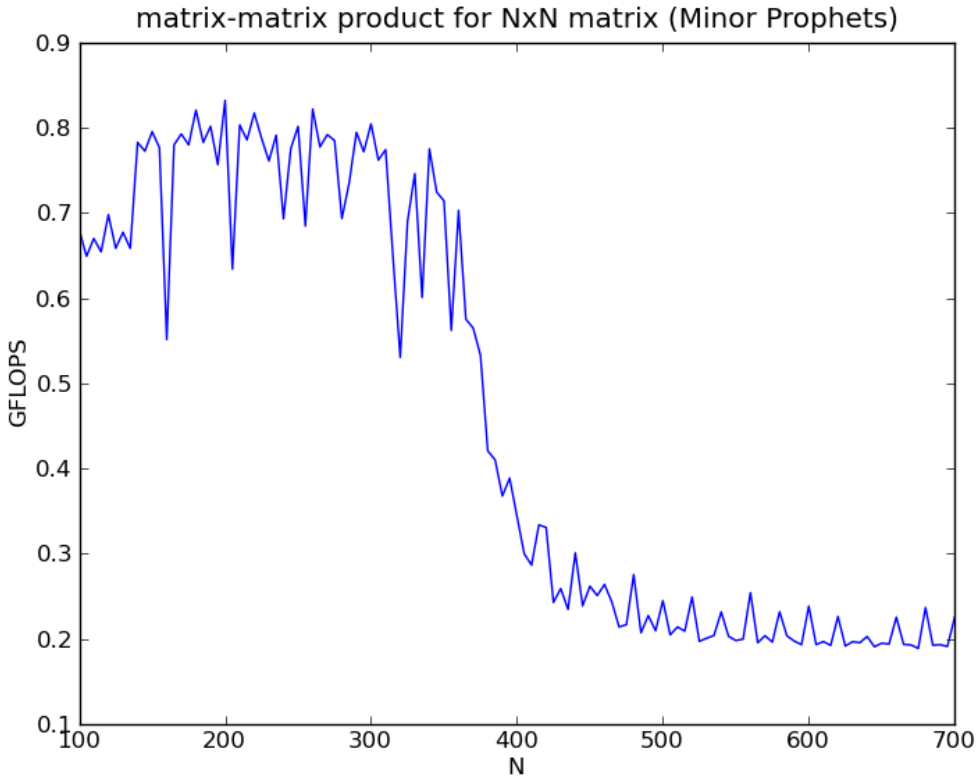
```
for i = 1 to n do
    for j = 1 to n do
        c_ij = 0
        for k = 1 to n do
            c_ij = c_ij + a_ik b_kj
        endfor
    endfor
endfor
```

Notice that both A and C are accessed by rows in the innermost loop while B is accessed by columns.

Effect of memory hierarchy sizes on performance

The graph below shows GFLOP/s vs matrix dimension for a rowwise matrix-matrix product. The drop-off in performance between $N = 300$ and $N = 400$ suggests that all of B can fit in the processor's cache when $N = 300$ but not when $N = 400$.

- A 300×300 matrix using double precision floating point values consumes approximately 700 KB.
- A 400×400 matrix needs at least 1250 KB.
- The cache on the machine used for this test is 2048 KB.



Note: This graph was generated on a previous generation Minor Prophets machines.

Outline

Key idea: minimize cache misses

If we partition A and B into appropriate sized blocks

$$A = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix}, \qquad B = \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

then the product $C = AB$ can be computed as

$$C = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{11}B_{11} \end{bmatrix}$$

where each of the matrix-matrix products involves matrices of roughly $1/4$ the size of the full matrices.

Recursion

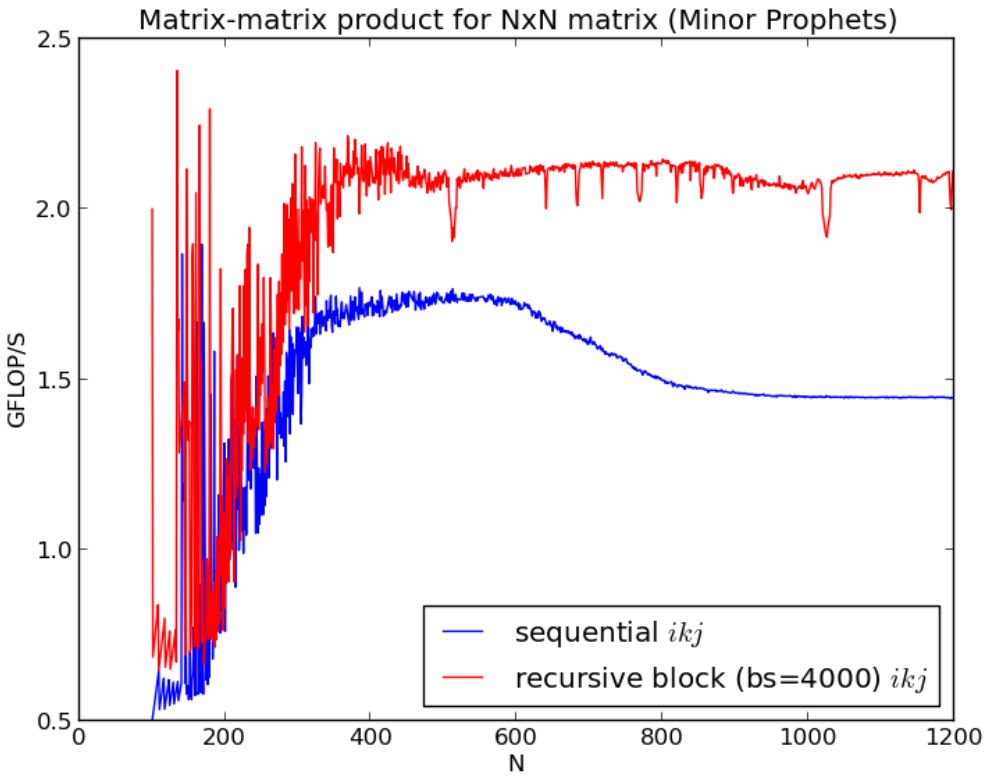
We can exploit this idea quite easily with a recursive algorithm.

matrix function C = mult(A, B)
    if (size(B) > threshold) then
        $C_{00} = \text{mult}(A_{00}, B_{00}) + \text{mult}(A_{01}, B_{10})$
        $C_{01} = \text{mult}(A_{00}, B_{01}) + \text{mult}(A_{01}, B_{11})$
        $C_{10} = \text{mult}(A_{10}, B_{00}) + \text{mult}(A_{11}, B_{10})$
        $C_{11} = \text{mult}(A_{10}, B_{01}) + \text{mult}(A_{11}, B_{11})$
    else
        $C = AB$
    endif
    return C

Recursive block-oriented matrix-matrix product

The graph below shows GFLOP/s vs matrix dimension for both a rowwise product and a recursive block-oriented matrix-matrix product.

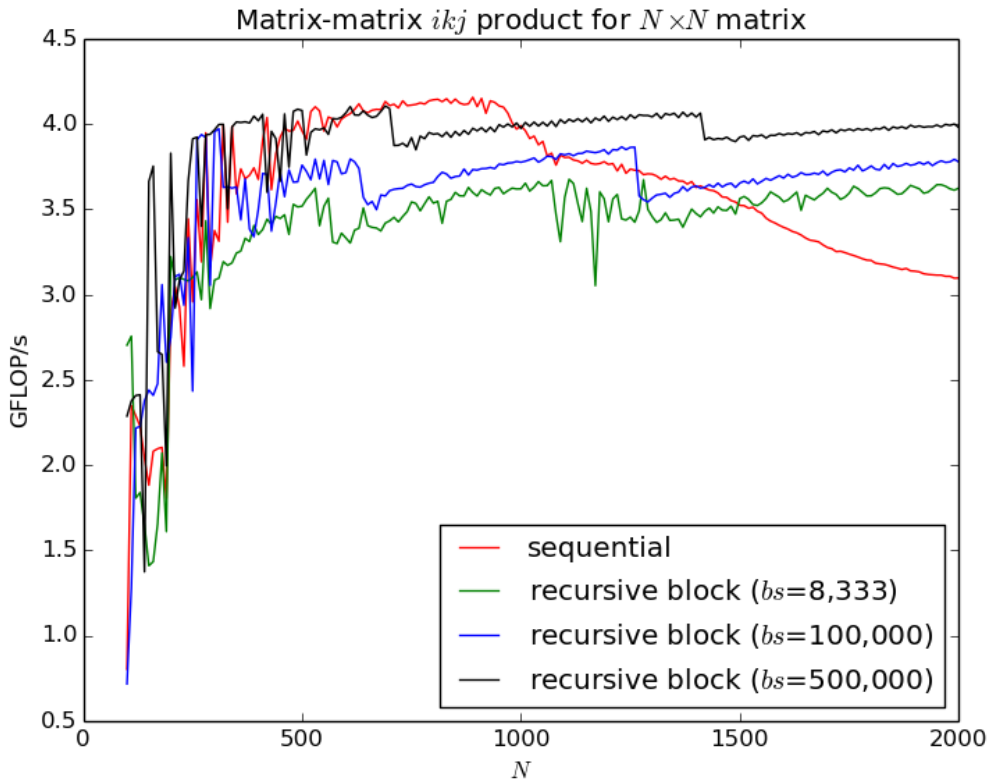
- Performance is improved across the board, including for smaller matrix sizes
- Performance is relatively flat - no degradation as matrix sizes increase.



Recursive block-oriented matrix-matrix product

This graph shows data for matrix-matrix products but was generated on our 2016 workstation cluster. Notice the larger matrix dimension (now $N = 2000$) and overall higher GFLOP/s rates.

- Probably need to go to $N = 3000$ or more to show leveling-off.
- Note that increasing block size indefinitely does not help; the sequential algorithm essentially used $8N^2$ as the block size.



Outline

Basic Linear Algebra Subprograms

The BLAS are routines that provide standard building blocks for performing basic vector and matrix operations:

- the Level 1 BLAS perform scalar, vector and vector-vector operations,
- the Level 2 BLAS perform matrix-vector operations, and
- the Level 3 BLAS perform matrix-matrix operations.
- Because the BLAS are efficient, portable, and widely available, they are commonly used in the development of high quality linear algebra software.
- The BLAS homepage is `http://www.netlib.org/blas/`.

BLAS description

- The name of each BLAS routine (usually) begins with a character that indicates the type of data it operates on:
  - S Real single precision.
  - D Real double precision.
  - C Complex single precision.
  - Z Complex double precision.
- Level 1 BLAS handle operations that are $O(n)$ data and $O(n)$ work
- Level 2 BLAS handle operations that are $O(n^2)$ data and $O(n^2)$ work
- Level 3 BLAS handle operations that are $O(n^2)$ data and $O(n^3)$ work

Level 1 BLAS

Level 1 BLAS are designed for operations with $O(n)$ data and $O(n)$ work.

Some BLAS 1 subprograms

`xCOPY`	copy one vector to another
`xSWAP`	swap two vectors
`xSCAL`	scale a vector by a constant
`xAXPY`	add a multiple of one vector to another
`xDOT`	inner product
`xASUM`	1-norm of a vector
`xNRM2`	2-norm of a vector
`IxAMAX`	find maximal entry in a vector

Notice the exception to the naming convention in the last line. BLAS functions returning an integer have names starting with `I` so the datatype indicator is displaced to the second position.

BLAS 1 quick reference: `http://www.netlib.org/blas/blasqr.ps`.

Level 2 BLAS

Level 2 BLAS are designed for operations with $O(n^2)$ data and $O(n^2)$ work.

Some BLAS 2 subprograms

`xGEMV`	general matrix-vector multiplication
`xGER`	general rank-1 update
`xSYR2`	symmetric rank-2 update
`xTRSV`	solve a triangular system of equations

A detailed description of BLAS 2 can be found at
http://www.netlib.org/blas/blas2-paper.ps.

Level 3 BLAS

Level 3 BLAS are designed for operations with $O(n^2)$ data and $O(n^3)$ work.

Some BLAS 3 subprograms

`xGEMM`	general matrix-matrix multiplication
`xSYMM`	symmetric matrix-matrix multiplication
`xSYRK`	symmetric rank-k update
`xSYR2K`	symmetric rank-$2k$ update

A detailed description of BLAS 3 can be found at
http://www.netlib.org/blas/blas3-paper.ps.

BLAS are written in Fortran

- Calling BLAS routines from Fortran is relatively straightforward
- Fortran stores two-dimension arrays in column-major order. Many BLAS routines accept a parameter named `LDA` which stands for the *leading dimension of A*. In Fortran this is the column length.
- `LDA` is needed to handle strides between elements common to a single row of A.
- In contrast, C and C++ store two-dimensional arrays in row-major order. Rather than the leading dimension, the trailing dimension specifies the stride between elements common to a single column.
- Calling vector-only BLAS routines from C is relatively straightforward.
- One must work with transposes of matrices in C/C++ programs to use the BLAS matrix-vector and matrix-matrix routines.

Calling BLAS routine `DGEMM()` from Fortran

DGEMM() performs the operation

$$C \leftarrow \alpha \text{op}(A)\text{op}(B) + \beta C$$

where op(A) can either be A or A^T. The Fortran calling sequence is

```
DGEMM(TRANSA,TRANSB,M,N,K,ALPHA,LDA,B,LDB,BETA,C,LDC)
```

`TRANSA,TRANSB`: (`CHARACTER*1`) `'N'` for no transpose, `'T'` or `'Y'` to transpose

`M`: (`INTEGER`) number of rows in C and A

`N`: (`INTEGER`) number of columns in C and B

`K`: (`INTEGER`) number of columns in A and rows in B

`ALPHA, BETA`: (`DOUBLE PRECISION`) scale factors for AB and C

`LDA, LDB, LDC`: (`INTEGER`) leading dimensions of A, B, and C

CBLAS: The BLAS for C/C++

- A version of the BLAS has been written with routines that can be called directly from C/C++.
- The CBLAS routines have the same name as their Fortran counterparts except all subprogram and function names are lowercase and are prepended with `cblas_`.
- The CBLAS matrix routines also take an additional parameter to indicate if the matrix is stored in row-major or column-major order.
- The leading dimension parameters should correspond to the row length if matrix is in row-major order and the column length if the matrix is in column-major order.

Example: Calling CBLAS routine `cblas_dgemm()` from C

The corresponding prototype in CBLAS file would look like

```
void cblas_dgemm(const enum CBLAS_ORDER Order,
        const enum CBLAS_TRANSPOSE TransA,
        const enum CBLAS_TRANSPOSE TransB,
        const int M, const int N, const int K,
        const double alpha, const double *A,
        const int lda, const double *B, const int ldb,
        const double beta, double *C, const int ldc);
```

Changes from the Fortran BLAS:

- The routine name written in lowercase and prepended with `cblas_`
- The new first argument `Order` should be either `CblasColMajor` or `CblasRowMajor` to indicate how the matrices are stored.
- `TransA` and `TransB` and are typically either `CblasNoTrans` or `CblasTrans`.
- Many values are passed by value, whereas all values are passed by reference in the Fortran BLAS.

Outline

LAPACK

The **Linear Algebra PACKage** is a successor to both LINPACK and EISPACK.

- LINPACK is a library of Fortran routines for numerical linear algebra and written in the 1970s.
- EISPACK is a library of Fortran routines for numerical computation of eigenvalues and eigenvectors of matrices and was also written in the 1970s.
- One of the LINPACK authors, Cleve Moler, went on to write an interactive, user-friendly front end to these libraries called **MATLAB**.
- LINPACK and EISPACK primarily make use of the level 1 BLAS
- LAPACK largely replaces LINPACK and EISPACK but takes advantage of level 2 and level 3 BLAS for more efficient operation on computers with hierarcharical memory and shared memory multiprocessors.

The LAPACK homepage is `http://www.netlib.org/lapack/`.

LAPACK for C/C++

- LAPACK is written in Fortran 90, so calling from C/C++ has the same challenges as discussed for the BLAS.
- A version called CLAPACK exists that can be more easily called from C/C++ but still has the column-major matrix access issue.

Outline

ATLAS

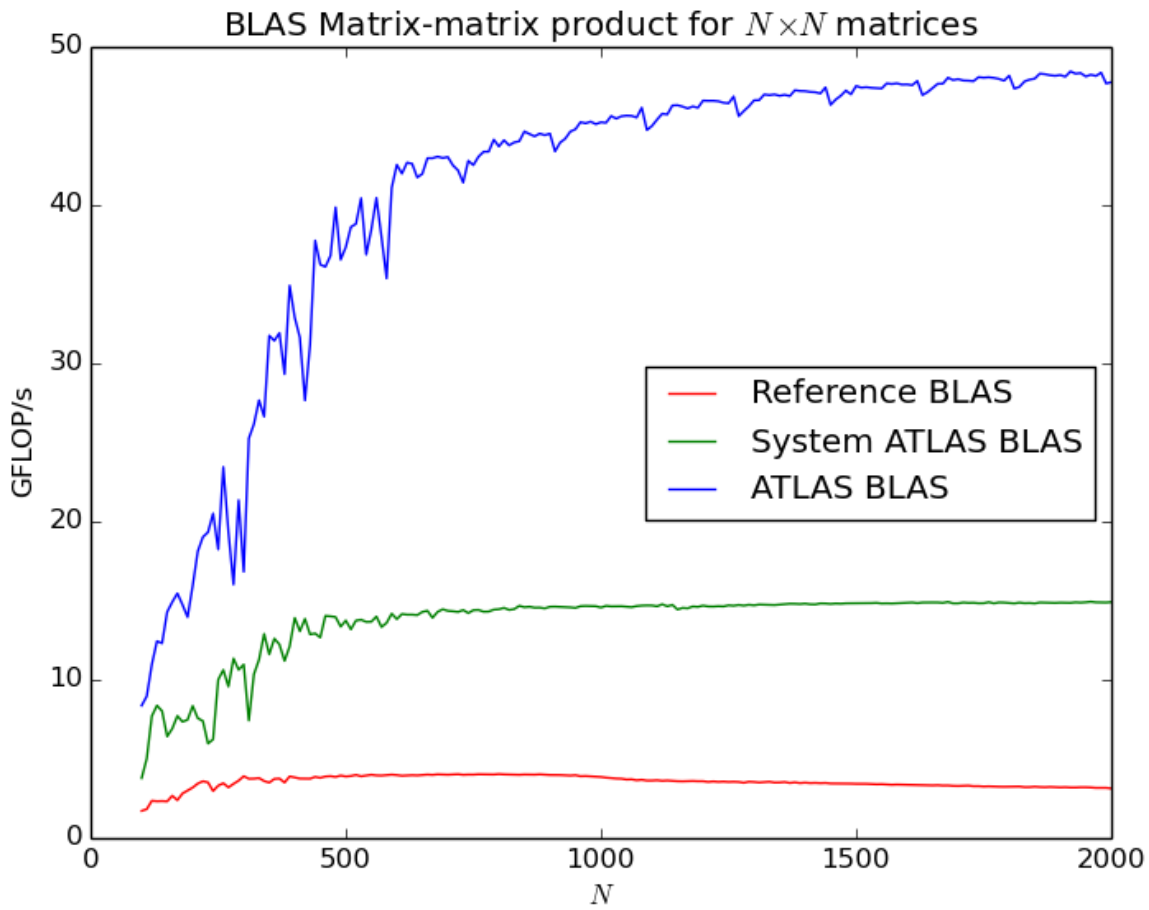
ATLAS stands for **Automatically Tuned Linear Algebra Software** and consists of the BLAS and some routines from LAPACK.

- The key to getting high performance from the BLAS (and hence from LAPACK) is using BLAS routines that are tuned to each particular machine's architecture and compiler.
- Vendors of HPC equipment typically supply a BLAS library that has been optimized for their machines.
- The ATLAS project was created to provide similar support for HPC equipment built from commodity hardware (e.g. Beowulf clusters).

The ATLAS homepage is `http://math-atlas.sourceforge.net/`.

ATLAS Performance Example

The graph below shows GFLOP/s vs. matrix dimension for matrix products using several BLAS implementations. The testing was done on a Lenovo ThinkStation E32 with 8GB RAM running Ubuntu 14.04.



The Reference BLAS are supplied with LAPACK and are not optimized.

The System ATLAS BLAS are supplied with Ubuntu 14.04.

The ATLAS BLAS are version 3.11.38 and were compiled on the host.

Multithreaded Open BLAS Performance Example

The graph below shows GFLOP/s vs. matrix dimension for matrix products using several BLAS implementations. The testing was done on a Lenovo ThinkStation E32 with 4 cores and 8GB RAM running Ubuntu 14.04.