18.337 Parallel Scientific Computing	February 23, 1995
Lecture 4	
Topic: Dense Linear Algebra	*1995 Scribe: Rajesh Venkataramani*

Outline of Todays Lecture: Dense Linear Algebra

- Introduction
- Applications
- Records
- Software Libraries
- Software Engineering

Introduction

We next look at *Dense Linear Systems* of the form $\mathbf{Ax} = \mathbf{b}$. Before solving dense linear algebra problems, we should define the terms sparse, dense, and structured.

Defn: A *sparse* matrix is a matrix with enough zeros that it is worth taking advantage of them.

Defn: A *structured* matrix has enough structure that it is worthwhile to use it.
Eg. Toeplitz Matrix 2n parameters

$$ToeplitzMatrix = \begin{bmatrix} 1 & 4 & & & & \\ 2 & 1 & 4 & & & \\ & 2 & 1 & 4 & & \\ \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\ & & & 2 & 1 & 4 \\ & & & & 2 & 1 \end{bmatrix}$$

Defn: A *dense* matrix is neither sparse nor structured.

Applications

There are not many applications for large dense linear algebra routines, perhaps due to the law of nature below. However, it would be wrong to conclude that the world does not need large linear algebra libraries. Medium sized problems are most easily solved with these libraries, and the first pass at larger problems are best done with the libraries. Dense

Year	Size of Dense System	Machine
1950's	≈ 100	
1991	55,296	
1992	75,264	Intel
1993	75,264	Intel
1993	76,800	CM
1993	128,600	Intel

Table 1: Largest Dense Matrices Solved

methods are the easiest to use, reliable, predictable, easiest to write, and work best for small to medium problems.
For large problems, it is not clear whether dense methods are best, but other approaches often require far more work.

- "Law of Nature": Nature does not throw n^2 numbers at us haphazardly, therefore there are few dense matrix problems. Some believe that there are no real problems that will turn up n^2 numbers to populate the $n \times n$ matrix without exhibiting some form of underlying structure. This implies that we should seek methods to identify the structure underlying the matrix. This becomes particulary important when the size of the system becomes large.

Records

Table 1 shows the largest dense matrices solved. Problems that warrant such huge systems to be solved are typically things like the Stealth bomber and large Boundary Element codes[1]. We see here that space considerations, *not* processor speeds, are what bound the ability to tackle such large systems. Another application for large dense problems arise in the "methods of moments", electromagnetic calculations used by the military.

Memory is the bottleneck in solving these large dense systems. Only a tiny portion of the matrix can be stored inside the computer at any one time.

Software Libraries

There are libraries of routines that provide standard algorithms used in scientific computing. These routines are meant to be fast, accurate, and portable.

History of Library Development:

- *LINPACK* - Linear Systems

[1]Typically this method involves a transformation using Greens Theorem from 3D to a dense 2D representation of the problems. This is where the large data sets are generated.

- $EISPACK$ - Eigenvalue Problems
- $LAPACK$ - Combination of LinPack and EisPack that takes advantage of vectorization
- $scaLAPACk$ - Linear Algebra Pack for Parallel Machines (2/28/95)

$LINPACK$ is high quality numerical software, but has no knowledge of blocking or layout. $EISPACK$ enhanced $LINPACK$ by adding the power of eigenvalues. Both packages are outdated by $LAPACK$. $scaLAPACK$ is the current package under development.
Extensions to Scalapack:

- $BLACS$ - C stands for communication
- $PB-BLACS$ - Parallel Block $BLAS$
- $PBLAS$ - basic $BLAS$ plus communication tied together

Other packages include $IMSL$ and NAG.
There is a library specially made for the CM machines -

- $CMSSL$ - Connection Machine Scientific Software Libraries.

Software Engineering

Wrong View of linear algebra on parallel machines:

- one element per processor
- lots of processors
- communication is instantaneous

This is taught frequently in theory classes, but has no practical application.

Right View

- Large Chunks of each matrix on each processor

Time to execute an arithmetic operation depends on the number of accesses to slow memory. In Figure 1, the amount of memory is proportional to the area in the triangle. As seen, there is only a small amount of fast memory, and larger amounts of slower memory. The need for many memory references to slow memory slows down many computations.

An example is Matrix Multiply as shown in Figure 2.
This pseudo code produces terrible performance because too many memory references are needed to do the column times row vector calculation.

Table 2 shows why people vectorize. Blocking calculations are faster. The lesson of vector computers is that computations should be blocked to maximize efficiency.

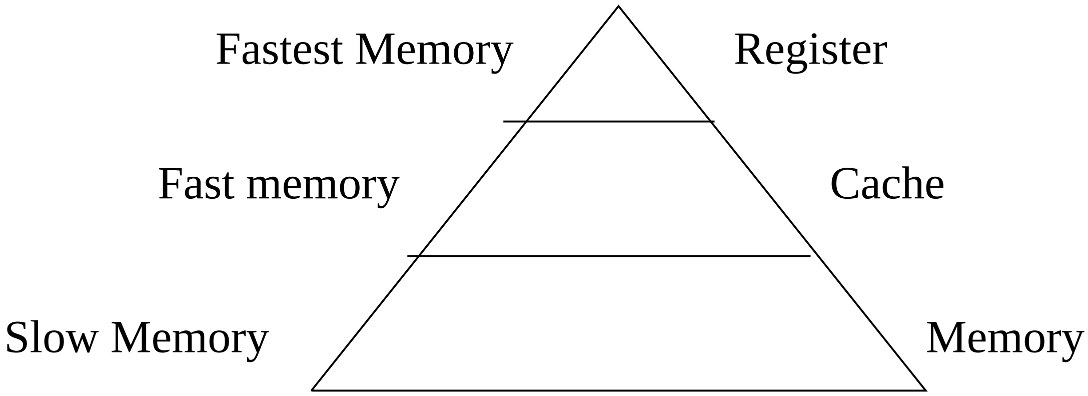

Figure 1: Matrix Operations on 1 processor

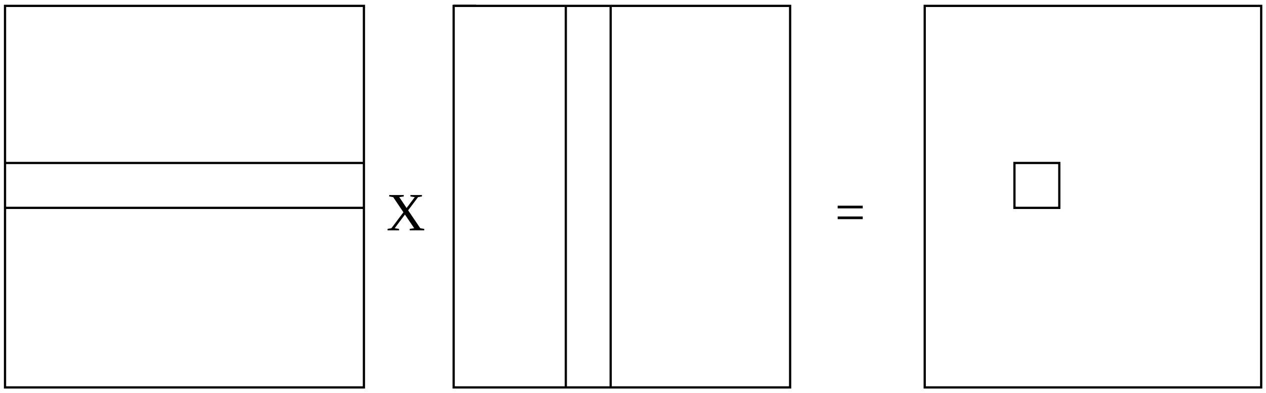

Figure 2: Matrix Multiply

Instruction	Operations	Memory Accesses (load/stores)	Ops /Mem Ref
BLAS1: SAXPY (Single Precision **Ax** Plus **y**)	2n	3n	$\frac{2}{3}$
BLAS1: SAXPY $\alpha =$ x . y	2n	2n	1
BLAS2: Matrix-vec $y = Ax + y$	$2n^2$	n^2	2
BLAS3: Matrix-Matrix $C = AB + C$	$2n^3$	$4n^2$	$\frac{1}{2}n$

Table 2: Basic Linear Algebra Subroutines (BLAS)

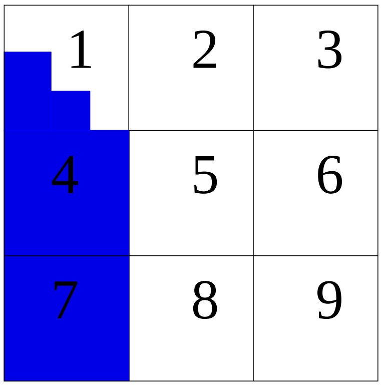

Figure 3: Gaussian Elimination With Bad Load Balancing

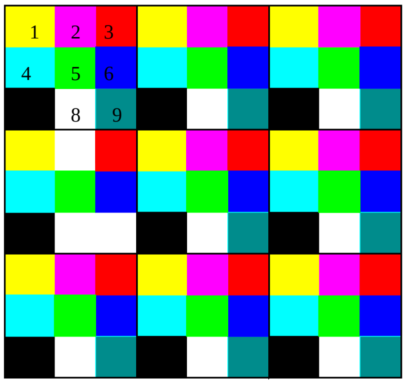

Figure 4: Row and Column Block Cyclic Ordering of Data

The grand plan is that manufacturers would write fast *BLAS* (not quite successful). *LAPACK* would write codes that calls the *BLAS*.

Load Balancing:

We will use Gaussian Elimination to demonstrate the advantage of cyclic order in dense linear algebra. In consecutive order, as the computation proceeds, processors in the lower triangular part become idle. In Figure 3, the lower processors would become idle as specified by the shaded processors, 4 and 7. Thus, this is bad load-balancing. While in cyclic order, we eliminate the same load for each processor, and the load remains balanced as the computation proceeds (See Figure 4) .

There are two methods to eliminate load imbalances:

- Rearrange the data for better load balancing (costs: communication).
- Rearrange the calculation: eliminate in unusual order.

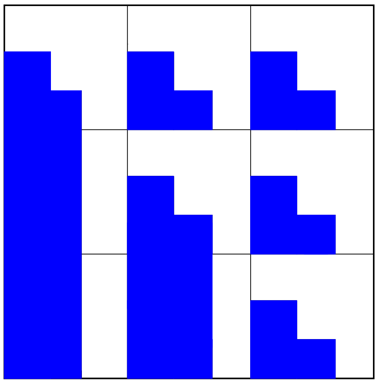

Figure 5: A stage in Gaussian elimination using cyclic order, where the shaded portion refers to the zeros and the unshaded refers to the non-zero elements

So, should we convert the data from consecutive to cyclic order and from cyclic to consecutive when we are done? The answer is "no", and the better approach is to reorganize the algorithm rather than the data. The idea behind this approach is to regard matrix indices as a set (not necessarily ordered) instead of an ordered sequence.

In general if you have to rearrange the data, maybe you can rearrange the calculation.

Lesson of Compuation Ordering: Matrix indices are a set (unordered), not a sequence (ordered). We have been taught in school to do operations in a linear order, but there is no mathematical reason to do this.

As Figure 5 demonstrates, we store data consecutively but do Gaussian Elimination cyclicly. In particular, if the block size is 10×10, the pivots are $1, 11, 21, 31, \ldots, 2, 22, 32, \ldots$.

We can apply the above reorganized algorithm in block form, where each processor does one block at a time and cycles through.

Here we are using all of our lessons, blocking for vectorization, and rearrangement of the calculation, not the data.