Lecture 4

Dense Linear Algebra

4.1 Dense Matrices

We next look at *dense* linear systems of the form $\mathbf{Ax} = \mathbf{b}$. Here A is a given $n \times n$ matrix and b is a given n-vector; we need to solve for the unknown n-vector x. We shall assume that A is a nonsingular matrix, so that for every b there is a unique solution $x = A^{-1}b$. Before solving dense linear algebra problems, we should define the terms sparse, dense, and structured.

Definition. (Wilkinson) A *sparse* matrix is a matrix with enough zeros that it is worth taking advantage of them.

Definition. A *structured* matrix has enough structure that it is worthwhile to use it.

For example, a Toeplitz Matrix is defined by $2n$ parameters. All entries on a diagonal are the same:

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

Definition. A *dense* matrix is neither sparse nor structured.

These definitions are useful because they will help us identify whether or not there is any inherent parallelism in the problem itself. It is clear that a sparse matrix does indeed have an inherent structure to it that may conceivably result in performance gains due to parallelism. We will discuss ways to exploit this in the next chapter. The Toeplitz matrix also has some structure that may be exploited to realize some performance gains. When we formally identify these structures, a central question we seem to be asking is if it is it worth taking advantage of this? The answer, as in all of parallel computing, is: "it depends".

If $n = 50$, hardly. The standard $O(n^3)$ algorithm for ordinary matrices can solve $Tx = b$, ignoring its structure, in under one one-hundredth of one second on a workstation. On the other hand, for large n, it pays to use one of the "fast" algorithms that run in time $O(n^2)$ or even $O(n \log^2 n)$. What do we mean by "fast" in this case?

It certainly seems intuitive that the more we know about how the matrix entries are populated the better we should be able to design our algorithms to exploit this knowledge. However, for the purposes of our discussion in this chapter, this does not seem to help for dense matrices. This does not mean that any parallel dense linear algebra algorithm should be conceived or even code in a serial manner. What it does mean, however, is that we look particularly hard at what the matrix A represents in the practical application for which we are trying to solve the equation $\mathbf{Ax} = \mathbf{b}$. By examining the matrix carefully, we might indeed recognize some other less-obvious 'structure' that we might be able to exploit.

4.2 Applications

There are not many applications for large dense linear algebra routines, perhaps due to the "law of nature" below.

- "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 particularly important when the size of the system becomes large.

What does it mean to 'seek methods to identify the structure'? Plainly speaking that answer is not known not just because it is inherently difficult but also because prospective users of dense linear algebra algorithms (as opposed to developers of such algorithms) have not started to identify the structure of their A matrices. Sometimes identifying the structure might involve looking beyond traditional literature in the field.

4.2.1 Uncovering the structure from seemingly unstructured problems

For example, in communications and radar processing applications, the matrix A can often be modelled as being generated from another $n \times N$ matrix X that is in turn populated with independent, identically distributed Gaussian elements. The matrix A in such applications will be symettric and will then be obtained as $A = XX^T$ where $(.)^T$ is the transpose operator. At a first glance, it might seem as though this might not provide any structure that can be exploited besides the symmetry of A. However, this is not so. We simply have to dig a bit deeper.

The matrix $A = XX^T$ is actually a very well studied example in random matrix theory. Edelman has studied these types of problems in his thesis and what turns out to be important is that in solving $\mathbf{Ax} = \mathbf{b}$ we need to have a way of characterizing the condition number of A. For matrices, the condition number tells us how 'well behaved' the matrix is. If the condition number is very high then the numerical algorithms are likely to be unstable and there is little guarantee of numerical accuracy. On the other hand, when the condition number is close to 1, the numerical accurarcy is very high. It turns out that a mathematically precise characterization of the random condition number of A is possible which ends up depending on the dimensions of the matrix X. Specifically for a fixed n and large N (typically at least $10n$ is increased, the condition number of A will be fairly localized i.e. its distribution will not have long tails. On the other hand, when N is about the size of n the condition number distribution will not be localized. As a result when solving $x = A^{-1}b$ we will get poor numerical accuracy in our solution of x.

This is important to remember because, as we have described all along, a central feature in parallel computing is our need to distribute the data among different computing nodes (processors,clusters, etc) and to work on that chunk by itself as much as possible and then rely on inter-node

Year	Size of Dense System	Machine
1950's	≈ 100	
1991	55,296	
1992	75,264	Intel
1993	75,264	Intel
1994	76,800	CM
1995	128,600	Intel
1996	128,600	Intel
1997	235000	Intel ASCI Red
1998	431344	IBM ASCI Blue
1999	431344	IBM ASCI Blue
2000	431344	IBM ASCI Blue
2001	518096	IBM ASCI White-Pacific
2002	1041216	Earth Simulator Computer
2003	1041216	Earth Simulator Computer

Table 4.1: Largest Dense Matrices Solved

communication to collect and form our answer. If we did not pay attention to the condition number of A and correspondingly the condition number of *chunks* of A that reside on different processors, our numerical accuracy for the parralel computing task would suffer.

This was just one example of how even in a seemingly unstructured case, insights from another field, random matrix theory in this case, could potentially alter our impact or choice of algorithm design. Incidentally, even what we just described above has not been incorporated into any parallel applications in radar processing that we are aware of. Generally speaking, the design of efficient parallel dense linear algebra algorithms will have to be motivated by and modified based on specific applications with an emphasis on *uncovering the structure* even in seemingly unstructured problems. This, by definition, is something that only users of algorithms could do. Until then, an equally important task is to make dense linear algebra algorithms and libraries that run efficiently regardless of the underlying structure while we wait for the applications to develop.

While there are not too many everyday applications that require dense linear algebra solutions, 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 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.

4.3 Records

Table 4.1 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]. Another application for large dense problems arise in the "methods of moments", electro-magnetic calculations used by the military.

[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.

It is important to understand that space considerations, *not* processor speeds, are what bound the ability to tackle such large systems. 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. It is also instructive to look at how technological advances change some of these considerations.

For example, in 1996, the record setter of size $n = 128,600$ required $(2/3)n^3 = 1.4 \times 10^{15}$ arithmetic operations (or four times that many if it is a complex matrix) for its solution using Gaussian elimination. On a fast uniprocessor workstation in 1996 running at 140 MFlops/sec, that would take ten million seconds, about 16 and a half weeks; but on a large parallel machine, running at 1000 times this speed, the time to solve it is only 2.7 hours. The storage requirement was $8n^2 = 1.3 \times 10^13$ bytes, however. Can we afford this much main memory? Again, we need to look at it in historical perspective.

In 1996, the price was as low as $10 per megabyte it would cost $ 130 million for enough memory for the matrix. Today, however, the price for the memory is much lower. At 5 cents per megabyte, the memory for the same system would be $650,000. The cost is still prohibitive, but much more realistic.

In contrast, the Earth Simulator which can solve a dense linear algebra system with $n = 1041216$ would require $(2/3)n^3 = 7.5 \times 10^{17}$ arithmetic operations (or four times that many if it is a complex matrix) for its solution using Gaussian elimination. For a 2.25 GHz Pentium 4 uniprocessor based workstation available today, at a speed of 3 GFlops/sec this would take 250 million seconds or roughly 414 weeks or about 8 years! On the Earth Simulator running at its maximum of 35.86 TFlops/sec or about 10000 times the speed of a desktop machine, this would only take about 5.8 hrs! The storage requirement for this machine would be $8n^2 = 8.7 \times 10^{14}$ bytes which at 5 cents a megabyte works out to about $43.5 million. This is still equally prohibitive athough the figurative 'bang for the buck' keeps getting better.

As in 1996, the cost for the storage was not as high as we calculated. This is because in 1996, when most parallel computers were specially designed supercomputers, "out of core" methods were used to store the massive amount of data. In 2004, however, with the emergence of clusters as a viable and powerful supercomputing option, network storage capability and management becomes an equally important factor that adds to the cost and complexity of the parallel computer.

In general, however, Moore's law does indeed seem to be helpful because the cost per Gigabyte especially for systems with large storage capacity keeps getting lower. Concurrently the density of these storage media keeps increasing as well so that the amount of physical space needed to store these systems becomes smalller. As a result, we can expect that as storage systems become cheaper *and* denser, it becomes increasingly more practical to design and maintain parallel computers.

The accompanying figures show some of these trends in storage density, and price.

4.4 Algorithms, and mapping matrices to processors

There is a simple minded view of parallel dense matrix computation that is based on these assumptions:

- one matrix element per processor
- a huge number (n^2, or even n^3) of processors
- communication is instantaneous

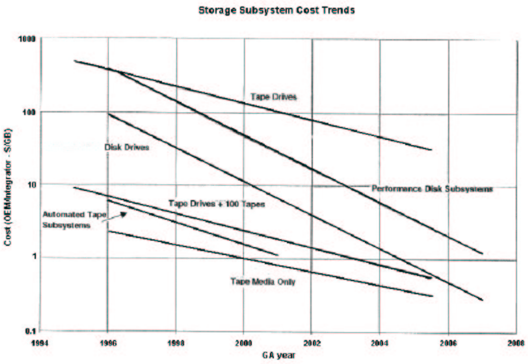


Figure 4.1: Storage sub system cost trends

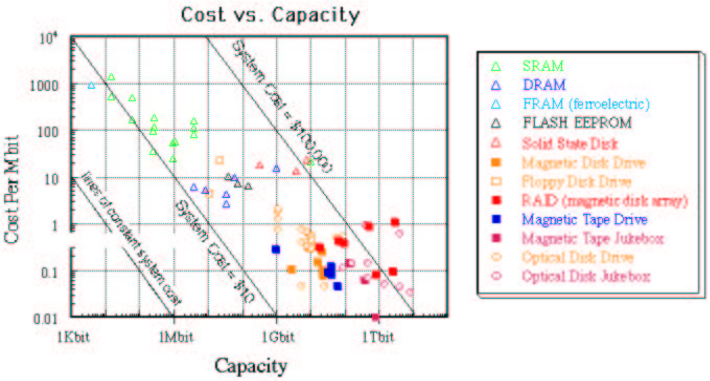


Figure 4.2: Trend in storage capacity

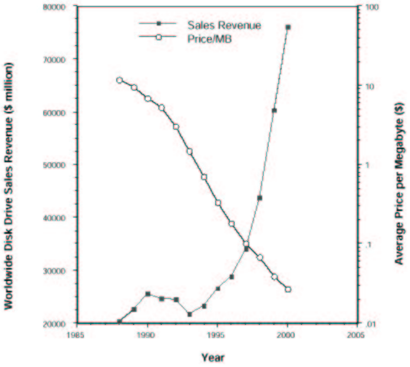


Figure 4.3: Average price per Mb cost trends

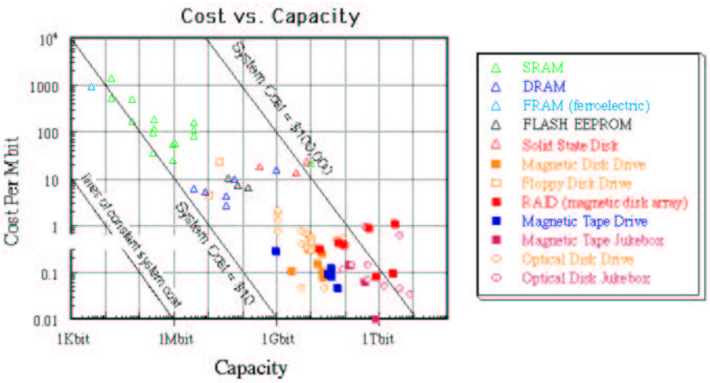


Figure 4.4: Storage density trends

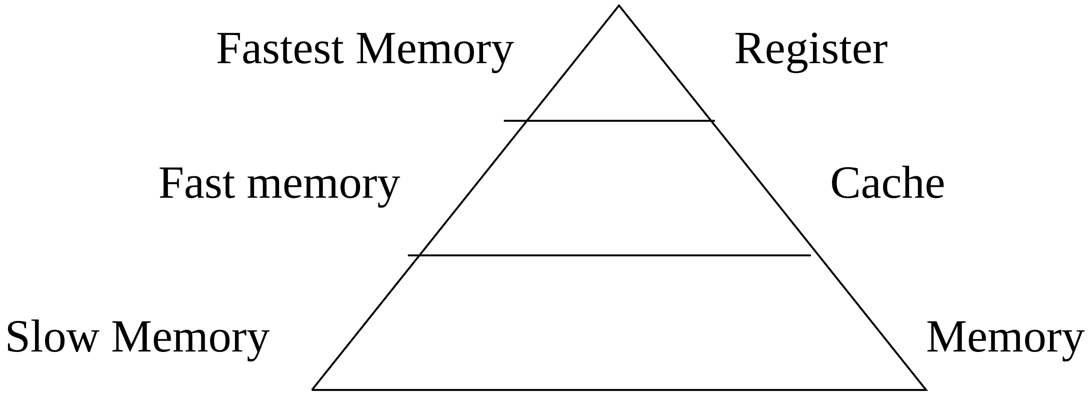


Figure 4.5: Matrix Operations on 1 processor

This is taught frequently in theory classes, but has no practical application. Communication cost is critical, and no one can afford n^2 processors when $n = 128,000$.[2]

In practical parallel matrix computation, it is essential to have large chunks of each matrix on each processor. There are several reasons for this. The first is simply that there are far more matrix elements than processors! Second, it is important to achieve *message vectorization*. The communications that occur should be organized into a small number of large messages, because of the high message overhead. Lastly, uniprocessor performance is heavily dependent on the nature of the local computation done on each processor.

4.5 The memory hierarchy

Parallel machines are built out of ordinary sequential processors. Fast microprocessors now can run far faster than the memory that supports them, and the gap is widening. The cycle time of a current microprocessor in a fast workstation is now in the $3 - 10$ nanosecond range, while DRAM memory is clocked at about 70 nanoseconds. Typically, the memory bandwidth onto the processor is close to an order of magnitude less than the bandwidth required to support the computation.

[2] Biological computers have this many processing elements; the human brain has on the order of 10^{11} neurons.

To match the bandwidths of the fast processor and the slow memory, several added layers of memory hierarchy are employed by architects. The processor has registers that are as fast as the processing units. They are connected to an on-chip cache that is nearly that fast, but is small (a few ten thousands of bytes). This is connected to an off-chip level-two cache made from fast but expensive static random access memory (SRAM) chips. Finally, main memory is built from the least cost per bit technology, dynamic RAM (DRAM). A similar caching structure supports instruction accesses.

When LINPACK was designed (the mid 1970s) these considerations were just over the horizon. Its designers used what was then an accepted model of cost: the number of arithmetic operations. Today, a more relevant metric is the number of references to memory that miss the cache and cause a cache line to be moved from main memory to a higher level of the hierarchy. To write portable software that performs well under this metric is unfortunately a much more complex task. In fact, one cannot predict how many cache misses a code will incur by examining the code. One cannot predict it by examining the machine code that the compiler generates! The behavior of real memory systems is quite complex. But, as we shall now show, the programmer can still write quite acceptable code.

(We have a bit of a paradox in that this issue does not really arise on Cray vector computers. These computers have no cache. They have no DRAM, either! The whole main memory is built of SRAM, which is expensive, and is fast enough to support the full speed of the processor. The high bandwidth memory technology raises the machine cost dramatically, and makes the programmer's job a lot simpler. When one considers the enormous cost of software, this has seemed like a reasonable tradeoff.

Why then aren't parallel machines built out of Cray's fast technology? The answer seems to be that the microprocessors used in workstations and PCs have become as fast as the vector processors. Their usual applications do pretty well with cache in the memory hierarchy, without reprogramming. Enormous investments are made in this technology, which has improved at a remarkable rate. And so, because these technologies appeal to a mass market, they have simple priced the expensive vector machines out of a large part of their market niche.)

4.6 Single processor condiderations for dense linear algebra

If software is expected to perform optimally in a parallel computing environment, performance considerations of computation on a single processor must first be evaluated.

4.6.1 LAPACK and the BLAS

Dense linear algebra operations are critical to optimize as they are very compute bound. Matrix multiply, with its $2n^3$ operations involving $3n^2$ matrix elements, is certainly no exception: there is on $O(n)$ *reuse* of the data. If all the matrices fit in the cache, we get high performance. Unfortunately, we use supercomputers for big problems. The definition of "big" might well be "doesn't fit in the cache."

A typical old-style algorithm, which uses the SDOT routine from the BLAS to do the computation via inner product, is shown in Figure 4.6.

This method produces disappointing performance because too many memory references are needed to do an inner product. Putting it another way, if we use this approach we will get $O(n^3)$ cache misses.

Table 4.6.1 shows the data reuse characteristics of several different routines in the BLAS (for Basic Linear Algebra Subprograms) library.

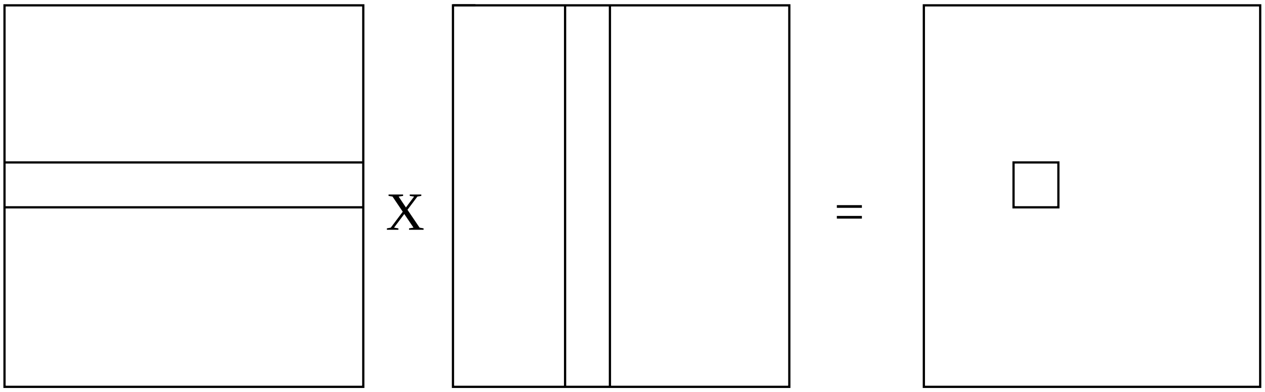


Figure 4.6: Matrix Multiply

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 4.2: Basic Linear Algebra Subroutines (BLAS)

Creators of the LAPACK software library for dense linear algebra accepted the design challenge of enabling developers to write portable software that could minimize costly cache misses on the memory hierarchy of any hardware platform.

The LAPACK designers' strategy to achieve this was to have manufacturers write fast BLAS, especially for the BLAS3. Then, LAPACK codes call the BLAS. *Ergo*, LAPACK gets high performance. In reality, two things go wrong. Manufacturers dont make much of an investment in their BLAS. And LAPACK does other things, so Amdahl's law applies.

4.6.2 Reinventing dense linear algebra optimization

In recent years, a new theory has emerged for achieving optimized dense linear algebra computation in a portable fashion. The theory is based one of the most fundabmental principles of computer science, recursion, yet it escaped experts for many years.

The Fundamental Triangle

In section 5, the memory hierarchy, and its affect on performance, is discussed. Hardware architecture elements, such as the memory hierarchy, forms just one apex of *The Fundamental Triangle*, the other two represented by software algorithms and the compilers. The Fundamental Triangle is a model for thinking about the performance of computer programs. A comprehensive evaluation of performance cannot be acheived without thinking about these three components and their relationship to each other. For example, as was noted earlier algorithm designers cannot assume that memory is infinite and that communication is costless, they must consider how their algorithms they write will behave within the memory hierarchy. This section will show how a focus on the

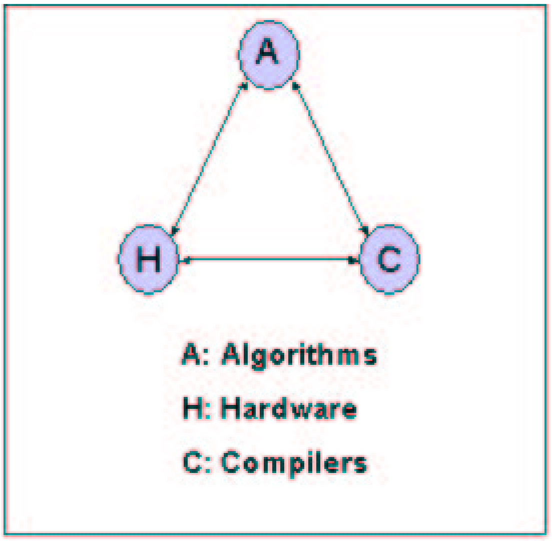


Figure 4.7: The Fundamental Triangle

interaction between algorithm and architecture can expose optimization possibilities. Figure 4.7 shows a graphical depiction of The Fundamental Triangle.

Examining dense linear algebra algorithms

Some scalar $a(i, j)$ algorithms may be expressed with square submatrix $A(I : * + NB - 1, J : J + NB - 1)$ algorithms. Also, dense matrix factorization is a BLAS level 3 computation consisting of a series of submatrix computations. Each submatrix computation is BLAS level 3, and each matrix operand in Level 3 is used multiple times. BLAS level 3 computation is $O(n^3)$ operations on $O(n^2)$ data. Therefore, in order to minimize the expense of moving data in and out of cache, the goal is to perform $O(n)$ operations per data movement, and amortize the expense over ther largest possible number of operations. The nature of dense linear algebra algorithms provides the opportunity to do just that, with the potential closeness of data within submatrices, and the frequent reuse of that data.

Architecture impact

The floating point arithmetic required for dense linear algebra computation is done in the L1 cache. Operands must be located in the L1 cache in order for multiple reuse of the data to yield peak performance. Moreover, operand data must map well into the L1 cache if reuse is to be possible. Operand data is represented using Fortran/C 2-D arrays. Unfortunately, the matrices that these 2-D arrays represent, and their submatrices, do not map well into L1 cache. Since memory is one dimensional, only one dimension of these arrays can be contiguous. For Fortran, the columns are contiguous, and for C the rows are contiguous.

To deal with this issue, this theory proposes that algorithms should be modified to map the input data from the native 2-D array representation to contiguous submatrices that can fit into the L1 cache.

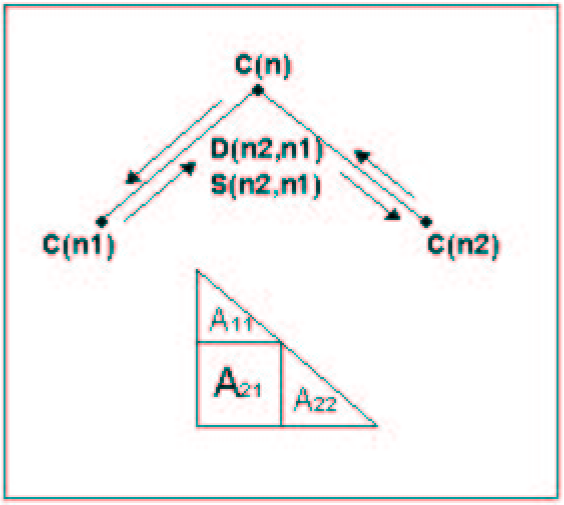


Figure 4.8: Recursive Submatrices

Blocking and Recursion

The principle of re-mapping data to form contiguous submatrices is known as blocking. The specific advantage blocking provides for minimizing data movement depends on the size of the block. A block becomes adventageous at minimizing data movement in and out of a level of the memory hierarchy when that entire block can fit in that level of the memory hierarcy in entirety. Therefore, for example, a certain size block would do well at minimizing register to cache data movement, and a different size block would do well at minimizing chach to memory data movement. However, the optimal size of these blocks is device dependent as it depends on the size of each level of the memory hierarchy. LAPACK does do some fixed blocking to improve performance, but its effectiveness is limited because the block size is fixed.

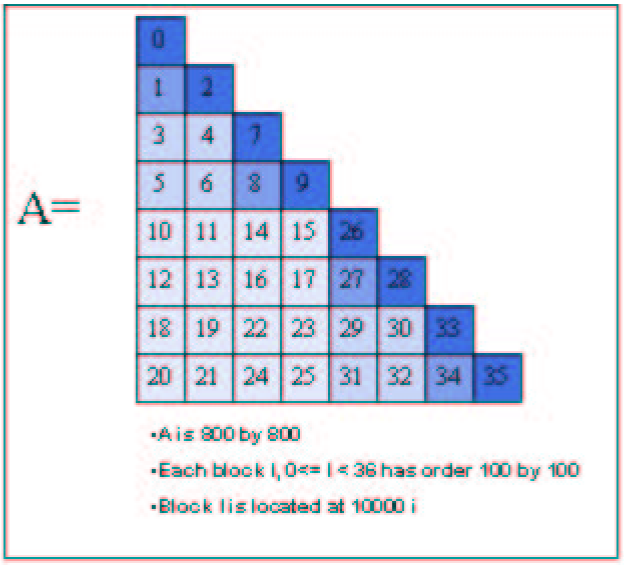
Writing dense linear algebra algorithms recursively enables automatic, variable blocking. Figure 4.8 shows how as the matrix is divided recursively into fours, blocking occurs naturally in sizes of $n, n/2, n/4...$ It is important to note that in order for these recursive blocks to be contiguous themselves, the 2-D data must be carefully mapped to one-dimensional storage memory. This data format is described in more detail in the next section.

The Recursive Block Format

The Recusive Block Format (RBF) maintains two dimensional data locality at every level of the one-dimensional tierd memory structure. Figure 4.9 shows the Recursive Block Format for a triangular matrix, an i right triangle of order N. Such a triange is converted to RBF by dividing each isoceles right triangle leg by two to get two smaller triangles and one “square”(rectangle).

Cholesky example

By utilizing the Recursive Block Format and by adopting a recursive strategy for dense linear algorithms, concise algorithms emerge. Figure 4.10 shows one node in the recursion tree of a recursive Cholesky algorithm. At this node, Cholesky is applied to a matrix of size n. Note that



•A is 800 by 800

•Each block I, 0<= I < 36 has order 100 by 100

•Block I is located at 10000 i

Figure 4.9: Recursive Block Format

n need not be the size of the original matrix, as this figure describes a node that could appear anywhere in the recursion tree, not just the root.

The lower triangular matrix below the Cholesky node describes the input matrix in terms of its recursive blocks, $A_{11}, A_{21}, and A_{22}$

- $n1$ is computed as $n1 = n/2$, and $n2 = n - n1$
- $C(n1)$ is computed recursively: Cholesky on submatrix A_{11}
- When $C(n1)$ has returned,L_{11} has been computed and it replaces A_{11}
- The DTRSM operation then computes $L_{21} = A_{21}L_{11_T}{}^{-}1$
- L_{21} now replaces A_{21}
- The DSYRK operation uses L_{21} to do a rank $n1$ update of A_{22}
- $C(n2)$, Cholesky of the updated A_{22}, is now computed recursively, and L_{22} is returned

The BLAS operations (i.e.DTRSM and DSYRK) can be implemented using matrix multiply, and the operands to these operations are submatrices of A. This pattern generalizes to other dense linear algebra computations (i.e. general matrix factor, QR factorization). Every dense linear algebra algorithm calls the BLAS several times. Every one of the multiple BLAS calls has all of its matrix operands equal to the submatrices of the matrices, A,B, .. of the dense linear algebra algorithm. This pattern can be exploited to improve performance through the use of the Recursive Data Format.

A note on dimension theory

The reason why a theory such as the Recursive Data Format has utility for improving computational performance is because of the mis-match between the dimension of the data, and the dimension

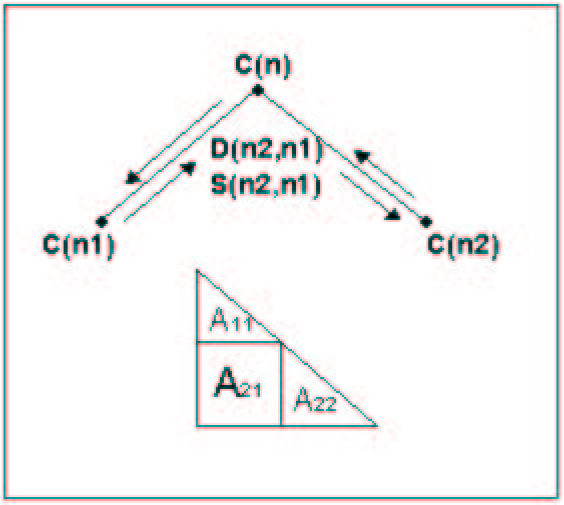


Figure 4.10: Recursive Cholesky

the hardware can represent. The laws of science and relate two and three dimensional objects. We live in a three dimensional world. However, computer storage is one dimensional. Moreover, mathmeticians have proved that it is not possible to maintain closeness between points in a neighborhood unless the two objects have the same dimension. Despite this negative theorem, and the limitations it implies on the relationship between data and available computer storage hardware, recursion provides a good approximation. Figure 4.11 shows this graphically via David Hilberts space filling curve.

4.7 Parallel computing considerations for dense linear algebra

Load Balancing:

We will use Gaussian elimination to demonstrate the advantage of cyclic distribution in dense linear algebra. If we carry out Gaussian elimination on a matrix with a one-dimensional block distribution, then as the computation proceeds, processors on the left hand side of the machine become idle after all their columns of the triangular matrices L and U have been computed. This is also the case for two-dimensional block mappings. This is poor load-balancing. With cyclic mapping, we balance the load much better.

In general, there are two methods to eliminate load imbalances:

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

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.

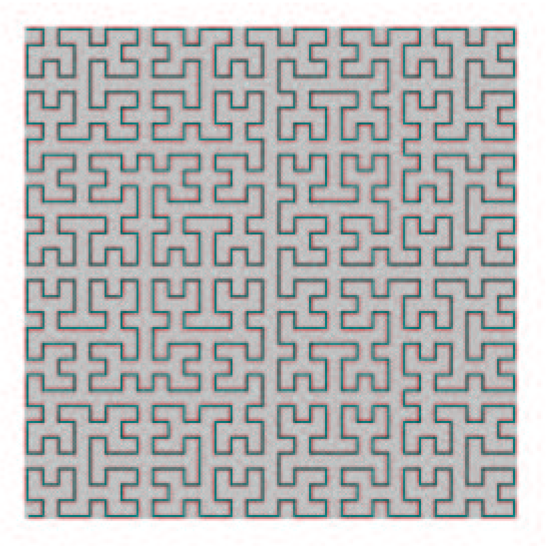


Figure 4.11: Hilbert Space Filling Curve

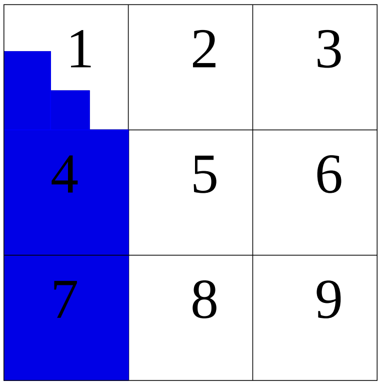


Figure 4.12: Gaussian elimination With Bad Load Balancing

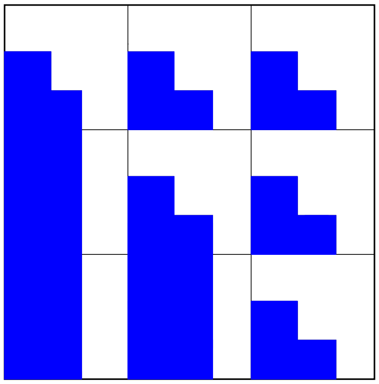


Figure 4.13: A stage in Gaussian elimination using cyclic order, where the shaded portion refers to the zeros and the unshaded refers to the non-zero elements

Lesson of Computation Distribution:

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 4.9 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, …, 2, 22, 32, ….

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.

4.8 Better load balancing

In reality, the load balancing achieved by the two-dimensional cyclic mapping is not all that one could desire. The problem comes from the fact that the work done by a processor that owns A_{ij} is a function of i and j, and in fact grows *quadratically* with i. Thus, the cyclic mapping tends to overload the processors with a larger first processor index, as these tend to get matrix rows that are lower and hence more expensive. A better method is to map the matrix rows to the processor rows using some heuristic method to balance the load. Indeed, this is a further extension of the moral above – the matrix row and column indices do not come from any natural ordering of the equations and unknowns of the linear system – equation 10 has no special affinity for equations 9 and 11.

4.8.1 Problems

1. For performance analysis of the Gaussian elimination algorithm, one can ignore the operations performed outside of the inner loop. Thus, the algorithm is equivalent to

```
do k = 1, n
    do j = k,n
        do i = k,n
           a(i,j) = a(i,j) - a(i,k) * a(k,j)
        enddo
    enddo
enddo
```

The “owner” of $a(i, j)$ gets the task of the computation in the inner loop, for all $1 \leq k \leq \min(i, j)$.

Analyze the load imbalance that occurs in one-dimensional block mapping of the columns of the matrix: $n = bp$ and processor r is given the contiguous set of columns $(r-1)b+1, \ldots, rb$. (Hint: Up to low order terms, the average load per processor is $n^3/(3p)$ inner loop tasks, but the most heavily loaded processor gets half again as much to do.)

Repeat this analysis for the two-dimensional block mapping. Does this imbalance affect the scalability of the algorithm? Or does it just make a difference in the efficiency by some constant factor, as in the one-dimensional case? If so, what factor?

Finally, do an analysis for the two-dimensional cyclic mapping. Assume the $p = q^2$, and that $n = bq$ for some blocksize b. Does the cyclic method remove load imbalance completely?