On Distributed I/O Minimal Matrix Multiplication

E. L. Leiss
Department of Computer Science
University of Houston
Houston Texas 77204-3475
coscel@cs.uh.edu

ABSTRACT
We consider matrix multiplication carried out on processors connected by a broadcast network and derive an algorithm of distributing the matrices in such a way that the amount of memory required at each node is minimal. We assume that the original matrices to be multiplied reside on disk and that there is one disk controller for the network. The approach given relies heavily on a non-standard mapping of matrices into main memory. The proposed method is compared with the standard methods (row- and column-major mapping) and is found to be significantly better.

* Research supported by Sandia National Laboratories.
1. Introduction

In [Leiss and Akyildiz], the notion of I/O minimality has been introduced; this is a property of an algorithm that essentially requires that the number of blocks that are transferred from disk to main memory or from main memory to disk is minimal for a given amount of available main memory and a given block or page size. While in-core programs are trivially I/O minimal, this property becomes very important for programs where the available (physical) main memory is insufficient; this necessitates either out-of-core programming or the use of virtual memory management. In that paper, an I/O minimal algorithm was given for matrix multiplication, for any size of block and any size of main memory available. At the core of the algorithm presented there is a non-standard mapping of matrices into main memory; instead of the conventional row- or column-major mapping, a blocked mapping is proposed whereby a matrix is subdivided into square submatrices of block size. This effectively addresses the problems created by the access patterns of matrix multiplication which requires a row wise access to the first matrix and a column wise one to the second matrix.

In this note we extend the notion of I/O minimality to a distributed computing environment. In this context, I/O minimality stipulates (as in the uniprocessor case) that the number of blocks or pages transferred between disk and main memory (in either direction) be minimal subject to the amount of main memory available (now of course this is a condition that must be satisfied for each node), but additionally it requires that the space used at each node be as small as possible for equal block transfer numbers. Because in this note we assume that the underlying communication network allows the broadcasting of blocks, we concentrate primarily on the second aspect of the definition, namely the minimization of the required space at each node, since in a broadcast environment, each block will be broadcast once, provided sufficient memory at the node is available.

2. Problem Statement; Assumptions

We assume a network of \( n^2 \) homogeneous processors, \( n \geq 1 \), connected to each other and to a disk controller by a bus-like network that allows us to broadcast blocks. (The disk controller could reside at one of the \( n^2 \) processors; in this case the homogeneity assumption excludes this particular aspect.)
We want to multiply two matrices A and B, each of size NxN, to yield the matrix C of the same size. Assume that

\[ N = k \cdot n. \]

We want to determine an efficient algorithm for solving this problem.

We assume that the two matrices A and B reside on (a single) disk; consequently, blocks consisting of a certain number of elements are transferred from disk to the processor that requests a particular matrix element if that block is not yet in the main memory of that processor. We assume that each block is of size \( k^2 \). (This can be changed to \( k^2/m \) for \( m \) some constant integer without affecting the major conclusions of our work.)

Since in practice, systems tend to be balanced, we assume that the time needed to transfer a block from disk to a processor is related to the processing speed of that processor. More specifically, we assume that the time \( t_{tr} \) required to carry out one arithmetic operation is related to the time \( t_{tr} \) that is needed in the best case to transfer one element from disk:

\[ t_{tr} = c \cdot t_{ar}. \]

Here, the "best" time \( t_{tr} \) is simply the time required to transfer a block, divided by the number of elements in that block; this reflects the expectation that all elements in the block will be used, not just one of them. This expectation is certainly satisfied by the method proposed in this paper.

Since the matrices A and B reside on disk and disk accesses are quite slow (in excess of tens of msec), practical values for the constant \( c \) exceed 100 and may range up to 1000 and more. (Thus, all timings are expressed dimensionless, with a unit of "time for one arithmetic operation" implied.) As a general rule, we assume sustained processor speeds of at least 20 MFLOPS; given the slow access to disk, the speed of the interconnection network (bus) should be of no concern (but we assume at least Ethernet speeds).

Under these general assumptions, we want to determine an efficient algorithm for computing \( A \cdot B \); efficient here means that the time required to perform the operation is as short as possible. It should be clear from our set-up that a very significant aspect of any solution is the access to disk since for larger values of \( c \), the access time is likely to dominate the computation time proper.
3. Comparing Mappings

Following [Leiss and Akyildiz], we first pose the question how the matrices should be mapped into main memory. We start by considering the two standard mappings, namely row-major and column-major.

**Row-major:** In the usual matrix multiplication
\[ c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \ldots + a_{in}b_{nj} \]
which is assumed throughout this paper, the matrix A is accessed in rows, the matrix B in columns. Thus, to compute \( c_{ij} \) we need the blocks for \( a_{i1}, a_{i2}, \ldots, a_{in}, b_{1j}, b_{2j}, \ldots, b_{nj} \). Since we have row-major ordering, we need about \( N/k^2 \) blocks (more specifically, either \( \lceil N/k^2 \rceil \) or \( \lfloor N/k^2 \rfloor \) + 1) for the A-elements. But we will need considerably more blocks for the B-elements; this can be seen as follows. \( b_{1j} \) is in one block. \( b_{2j} \) is \( N \) elements removed from \( b_{1j} \) in this ordering; therefore if \( k^2 \leq N \), \( N \) blocks are necessary for the B-elements, and if \( k^2 > N \), say \( k^2 = a \cdot N \) for \( a > 1 \), then approximately \( N/a \) blocks are necessary for the B-elements. Thus, row-major mapping requires \( \lceil N/k^2 \rceil + \min(N,\lfloor (N^2-N+1)/k^2 \rfloor) + \varepsilon \), where the integer \( \varepsilon \) is either 0, or 1, or 2. By introducing a real number \( \varepsilon' \) with \( 0 \leq \varepsilon' < 4 \), we can simplify this to
\[ N/k^2 + \min(N, (N^2-N+1)/k^2) + \varepsilon' \].

The analysis for column-major is similar, with the identical result.

Thus, either of the standard mappings requires \( N/k^2 + \min(N, (N^2-N+1)/k^2) + \varepsilon' + 1 \) blocks for the single element \( c_{ij} \) (since \( c_{ij} \) also resides in a separate block).

Note that in general, we do not compute one \( c_{ij} \) but an entire block of \( k^2 \) elements. The corresponding analysis of the row- and column-major mappings which is similar to the analysis carried out above shows that we need a total of \( N + N/k^2 + 1 + \varepsilon_2 \) blocks (for \( \varepsilon_2 \) a small non-negative constant).

**Blocked mapping:** We now assume the non-standard mapping used in [Leiss and Akyildiz] where each matrix is subdivided into \( n^2 \) blocks of size \( k \times k \). Thus each row and each column of a matrix is distributed over \( n \) blocks. The number of blocks that are required for the calculation of \( c_{ij} \) is therefore as follows. The required A-elements are in a "row" of blocks and the required B-elements are in a "column" of blocks. Since \( N = kn \), the number of blocks required to compute all C-elements in a block is \( 2n+1 \).
Comparing this mapping with the previous ones, it follows that the number of blocks required in the non-standard mapping is significantly smaller than that in the conventional ones; in fact, the difference is given by

\[(N + N/k^2 + 1 + \epsilon_2) - (2n+1) = (N + N/k^2 + 1 + \epsilon_2) - (2N/k+1) = N(1 - 1/k)^2 + \epsilon_2,\]

and this function of \(N\) and \(k\) is strictly positive for all \(k > 1\) and approaches \(N\) for increasing \(k\). We conclude that the blocked mapping is vastly superior to either the row-or the column-major mapping from the point of view of memory requirements at each processor.

**Example:** Consider the case \(N=2^{12}\) and \(n=2^6\); it follows that each matrix has 16.78 million elements, there are 4096 processors, and each block is of size 4\(k\), since \(k=2^6\). The conventional mappings (row or column) require 4097 blocks or almost 17 million elements; note that \(A\) and \(B\) together have only about 33.6 million elements! In contrast, the blocked mapping requires only 129 blocks or about 530,000 elements. Thus, using either of the conventional mappings, the space requirement at each processor would be almost 32 times greater than that for the blocked mapping!

This example demonstrates rather graphically that the mapping method can have an enormous influence on the resulting performance of a solution. Since excessively high space requirements are often impossible to meet, one would have to resort to out-of-core programming even though the sum total of the memory at all the nodes vastly exceeds the space required by the three matrices. Out-of-core programming not only complicates the programming significantly; it also most likely will increase the execution time since blocks (pages) must be swapped in and out as the program at each nodes demands. This is almost guaranteed to introduce quite large delays due to the great difference in access time (several orders of magnitude) between main memory accesses and disk accesses. Even greater delays are most likely introduced by the use of virtual memory management.

### 4. Time, Space, and Transfer Analysis

**Space:** With the blocked mapping each processor must have \(2n+1\) blocks; thus it must have available a space of at least \((2n+1)k^2\) or \(k^2 + 2kN\) elements. The total space required in the system is therefore \(n^2(k^2 + 2kN)\) or \((2n+1)N^2\).

Thus, there a penalty of
is introduced by the blocked mapping since the three matrices require $3N^2$ space. As we have seen, the penalty for row- or column-major mapping is much greater, namely in the case $k^2 \leq n$ approximately

$$(N + n/k)/3$$

since each processor has about $N/k^2 + N$ blocks of size $k^2$; therefore the total amount of space required in all $n^2$ processors comes to at least $(N/k^2 + N) \cdot k^2 \cdot n^2$ or $N^2 \cdot (N + n/k)$.

**Transfers:** In this paper, we assume a broadcast network. Thus, it follows that every one of the blocks of $A$ and of $B$ must be transferred exactly once, provided each processor can accommodate all the blocks that it will need during the entire operation. Therefore, there is no difference in the number of transfers of blocks between the various approaches, if this condition holds. (Note however that the choice of mapping may have a significant influence on whether the condition can be met; see the above comments on the space requirement at each node.)

This conclusion is clearly not valid if no broadcasting functionality exists. In this case, each block would have to be transferred point-to-point (i.e., disk controller to processor) individually. Given the penalty derived above, it is unlikely that either of these approaches is very attractive; the blocked mapping approach would incur a penalty of $(2n+1)/3$ (or slightly less than the square root of the number of processors) which would render it quite marginal, compared with a uniprocessor, out-of-core program. The conventional mapping approach would be completely unacceptable.

**Time:** Since we assume that blocks are broadcast, the time to transfer the blocks of $A$ and $B$ (which need not be sent to main memory) to the $n^2$ processors is $2cN^2$; there is no penalty since each block is broadcast only once and is received by all the processors that need it. Also, it is obvious that each element of a block will be required; thus the assumption that $t_r'$ is the "best" time is valid.

As stated before, we use the straightforward matrix multiplication algorithm with three nested loops; the number of arithmetic operations involved is therefore $2N^3-N^2$. We distinguish two ways of performing the computations:

(I) All blocks (of $A$ and $B$) are first transferred; only after this transfer is complete do the computations commence.

(II) Transfer of blocks and computations may occur in parallel; thus, as soon as a processor has corresponding $A$- and $B$-blocks, it may commence its computations.
Note that at any rate, it is blocks that are transferred; thus, only once an entire block has been received can operations involving any elements of that block begin.

Note that the operations are independent of the particular mapping scheme; all that is required is that the requisite elements reside in main memory. Both conventional mappings as well as our blocked mapping guarantee this, assuming the space necessary for this is available at each node. In Approach (I), the total time for any of the mappings (blocked, row-, or column-major) is \(2cN^2 + k^2(2N-1)\) or

\[
2cN^2 + 2k^2N - k^2
\]

In Approach (II), the total time for any of the mappings is

\[
2cN^2 + \text{ computation time for only those operations that involve elements of the last transferred block}
\]

This assumes that by the time the \(i\)th block has been transferred, all operations involving exclusively elements from the blocks 1 through \(i-1\) have already been completed. This is only possible if \(c\) is relatively large (in practice, this is a safe assumption). The last block's elements are involved in exactly \(n\) processors, as can easily be verified. Now assume that all those \(n\) processors do their computations at the same time and that all operations involving other elements have already been carried out. Therefore, the time to carry out these new operations and thereby complete the entire matrix multiplication is

\[
2k^2
\]

since there are \(k^2\) (size of a block) "incomplete" C-elements and at each of these \(n\) processors, and for each of these \(k^2\) elements one multiplication and one addition must be performed. Thus, we get as overall time

\[
2cN^2 + 2k^2
\]

The gap between Approaches (I) and (II) is

\[
k^2(2N - 3)
\]

which is independent of \(c\) (except that we made the assumption that \(c\) is sufficiently large; see above). This function can assume rather large values for large \(N\) (the only time one would want to use distributed matrix multiplication anyway!) and smaller \(n\) (and therefore larger \(k\)).

Example continued: We had \(N=2^{12}\) and \(k=n=2^6\). In this case the difference between (I) and (II) comes to almost \(2^{25}\) or 33.6 million instructions; note however that the transfer time is \(2c \cdot 2^{24}\) or about 8.4\(c\) million, which for practical values of \(c\) (\(c>>100\)) is significantly larger, assuming virtually any modern processor chip. In this example the overall time would come to

\[
8.4c \cdot 10^6 + 12.6 \cdot 10^6 \quad \text{(I)} \quad \text{versus} \quad 8.4c \cdot 10^6 + 8192 \quad \text{(II)}
\]
The uniprocessor computation time would be about 137.4 billion operations. At 100 MFLOPS, this would come to about 23 min. Approach (I) takes less than 1 sec for the computation, Approach (II) is even faster. If c=400, which corresponds to a rather high transfer rate of about 60 blocks per sec, the transfer of the two matrices takes over two min!

5. I/O Minimality

We must show that the proposed method is I/O minimal. First of all, the number of blocks transferred is minimal since each block is transferred exactly once. Therefore the I/O minimality of our approach follows if we can show that no other approach requires less main memory space at each node. This however can be seen from the discussing of the conventional mappings and the observation that only blocked mappings are unbiased toward row or column wise access.

6. Future Work

The algorithm as given assumes that the two operands, A and B, reside on disk and that the resulting product, C, remain distributed over the $n^2$ processors. If the matrix C is to be stored back to disk, there are again two possible approaches; either one waits until all of C is computed (at all the processors) -- Approach (I') -- or one begins transferring back those portions (blocks) of C that are complete as soon as they become available during the computation -- Approach (II'). The analysis of (I') is simple; one adds the transfer time for C of $cN^2$ to the overall computation time as determined above. The analysis of (II') is more involved. It is is clear from the discussions of Approaches (I) and (II) that there are $n$ processors that will be last. Thus, if one can guarantee that all the other processors have successfully completed the transmission of their blocks to disk, the overall time will be the computation time for Approach (II) plus the time to transmit the last $n$ blocks, which is $cnk^2$. However, to design a strategy of sending blocks that assures this is non-trivial. This is because there is only one controller that receives blocks, and therefore the sending of the blocks must be carefully orchestrated so that at any point in time, only one block is sent. Given the asynchronous nature of our network, this will require extensive synchronization, which can be either explicit (thereby introducing significant overhead) and/or implicit (i.e., using the timing of operations; this relies heavily on the homogeneity of the processors and assumes that no
network delays occur -- a dangerous assumption unless no other network traffic occurs!). This is the object of further study.

Also yet to be studied is the question of a block size that differs from $k^2$. Recall that $N=k\cdot n$ where $n^2$ is the number of processors. This means that the block size in the proposed algorithm is directly tied to the size of the matrices to be multiplied and the number of processors available. It is relatively easy to modify the given algorithm if the block size is $k^2/m$, for $m$ a given fixed integer. Other block sizes are not that easily handled.

Another problem to be addressed in later work deals with the question how to handle the situation where less main memory is available at each node than is required by the method proposed here. Clearly, the I/O minimal algorithm for uniprocessors given in [Leiss and Akyildiz] provides a solution; however, it is not clear whether the resulting method is still I/O minimal. In other words, even though the uniprocessor method is I/O minimal and the distributed algorithm is I/O minimal, it is conceivable that combining them will not yield an I/O minimal algorithm. One cause of this might be that different processors could require to receive blocks in different orders as stipulated by the uniprocessor algorithm, and that these ordering requirements might conflict.

Finally, we address a possible objection to our proposed scheme, namely that existing matrices are most likely stored in either row- or column-major order and therefore the required blocked mapping is simply not feasible. While in general, we would argue that matrices should never be stored in the conventional way if out-of-core programming or virtual memory management are used, in the problem at hand, there is a more immediate and direct solution to this problem. Suppose the matrix $M$ is stored in row-order. Retrieve the first $k$ rows of $M$ from disk, using one processor; then broadcast the $n$ blocks of size $k\times k$ that the blocked mapping yields. For column-order, an analogous approach using columns can be used. This requires a total of $N\cdot k$ main memory space, which translates to $N/k$ blocks. Since this is a good deal less that what each processor requires to do matrix multiplication, this effectively solves the problem since the complexity of this conversion is on the order of the original block transfer from disk to main memory. For the conversion from blocked order to row or column order, this process can be reversed.
Reference