ABSTRACT
Matrix presentation and operations have been examined for a long time. In
the conventional approach, the primary objective is the minimization of the
number of operations required by the algorithm. Here, our aim is minimizing
the number of page swaps between the main and the external memory in
paged memory systems. A carefully designed matrix algorithm can lead to
substantial savings in the number of page faults especially when a small part
of the total matrix can be held in the main memory at one time. When there
is not enough main memory available, there are two possible control options
for the retrieval strategy of all associated data sets and program of a
software: all control tasks are done by operating system, which is called
virtual memory management, or the task is handled by the programmer,
which is called out-of-core programming. Since a page swap may take an
amount of time that is five orders of magnitudes longer than that of an
arithmetic operation, I/O minimality is a crucial property for efficient
algorithms whose data sets cannot be accommodated in main memory at the
same time. Here, the development and implementation of an I/O minimal
algorithm for matrix multiplication is presented.

INTRODUCTION
The term virtual storage is associated with the ability to address a storage space much
larger than what is available in the primary storage of a particular computer system (Deitel,
1983). The advantages of the ability to execute software where program and data are only
partially in memory are as follows (Carr, 1984):
- Program and data are no longer constrained by the amount of physical
  memory that is available. Users are able to write programs for a very large virtual
  address space, simplifying the programming task (Shaw, Bic, 1988).
- Since each user requires less physical memory, more users can be run at the
  same time, with a corresponding increase in CPU utilization and throughput.

Virtual memory is not easy to implement, however, and may substantially decrease
performance, if it is used carelessly (Peterson, Silberschatz, 1986). Here comes the
importance of I/O management into place. There is a major discrepancy between the time
needed for performing calculations and the time needed for accessing data on external storage (Leiss, 1992). Hence, data transfers may significantly contribute to the overall execution time of a program. By out-of-core programming, I/O minimal algorithms can guarantee best performance of a given algorithm for the given memory space while virtual memory management can introduce nasty surprises.

In this paper, an I/O minimal algorithm is designed for matrix multiplication, which is a frequently used basic algorithm. The effect of data transfers on the overall execution time is examined using the dependence of I/O minimality on the memory space available and on the block size used for transfers. The next section discusses in-core and out-of-core programming and introduces the effects of I/O management on matrix multiplication. Then the general I/O minimal matrix multiplication algorithm is sketched and conclusions are drawn.

IN-CORE AND OUT-OF-CORE PROGRAMMING

A program's data sets may be retrieved and placed into memory in its entirety; this is called in-core programming. While this may need large physical memory space (Leiss, 1992), it is desirable as all data will be loaded into the main memory only once; thus no reloading is necessary. However, often the available memory is insufficient. When a program's data sets are retrieved in parts, there are two control options (Leiss, 1992):

- Control tasks, which include partitioning of data sets and determining which part should be in main memory and which part should be returned to tertiary memory, are performed by the operating system: this is called virtual memory management.
- All these control tasks plus the partitioning of the program itself will be handled exclusively by the programmer; this is called out-of-core programming.

Virtual memory systems are more convenient to use, but the programmer has little control over the memory management and also to some extent little control over the program. In out-of-core programming, on the other hand, the programmer is fully responsible for all data transfers and certain efficiencies are now attainable, but at the cost of complicating the program considerably (Leiss, 1992). There are also some problems associated with virtual memory (Davis, 1983). The main problems are increased task blocking and task switching when a page fault occurs, increased I/O traffic contention due to thrashing, etc. For these reasons some supercomputers do not support virtual memory systems, e.g. all Crays.

MATRIX MULTIPLICATION and I/O MINIMALITY

We call an algorithm I/O minimal (for a given amount of main memory space) if the number of retrievals of memory blocks, or pages, from external memory to the main memory or vice versa is minimal (Leiss, 1992).

We assume that we have two large matrices named A and B to be multiplied; the result will be stored in C. Also, we assume that all three matrices are of size (N,N), N a positive integer. We need 3*N^2 space to hold all matrices in the main memory for the case of in-
core programming. If less space is available in main memory, we must split the matrices into some submatrices and carry the multiplication out in steps. In the calculation of necessary space, memory used for counters, program code, etc. is not counted as this space is negligibly small compared with the memory for the matrices (Leiss, 1992).

Example 1: Assume that we have three pages in the memory, each being $N^2/4$ in size. We divide each matrix into four submatrices so that each fits into one page:

\[
\begin{array}{ccc|ccc}
A_{11} & A_{12} & B_{11} & B_{12} & C_{11} & C_{12} \\
A_{21} & A_{22} & B_{21} & B_{22} & C_{21} & C_{22}
\end{array}
\]

Hence, the multiplication involving the submatrices can be written as

\[
\begin{align*}
C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\
C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\
C_{21} &= C_{21} + A_{21}B_{12} \\
C_{22} &= C_{22} + A_{22}B_{21}
\end{align*}
\]

As we have only three pages in memory at any time and each step involves five submatrices, to calculate the $C_{ij}$'s we have to divide each addition into parts.

\[
\begin{align*}
C_{11} &= C_{11} + A_{11}B_{11} \\
C_{12} &= C_{12} + A_{12}B_{12} \\
C_{21} &= C_{21} + A_{21}B_{12} \\
C_{22} &= C_{22} + A_{22}B_{21}
\end{align*}
\]

We assume that the $C_{ij}$'s are initialized to 0. We can arrange the order of submultiplications in a way that will minimize the number of loads with out-of-core programming. Below we give a scheme where "*" means reuse of a submatrix.

\[
\begin{align*}
C_{11} &= A_{12}B_{21} \\
* &= A_{11}B_{11} \\
C_{12} &= A_{12}B_{22} \\
* &= A_{12}B_{22} \\
C_{22} &= A_{22}B_{21} \\
* &= A_{21}B_{12} \\
C_{21} &= A_{21}B_{12} \\
* &= A_{22}B_{21}
\end{align*}
\]

With this scheme we have 17 submatrix loads. This scheme is I/O minimal: At each instance we are able to keep only one submatrix fixed in memory; when one submatrix is fixed, the other two must be changed. In the case of in-core programming, the total number of loads is 12. With a brute force approach, the total number of loads is 24. Here we used one fourth the memory of in-core programming but needed only 17 loads.

Example 2: Now assume we have six pages in the memory of size $N^2/4$:

\[
\begin{align*}
C_{11} &= A_{11}B_{11} \\
* &= A_{12}B_{21} \\
C_{12} &= B_{22} \\
* &= B_{12} \\
C_{22} &= A_{22} \\
* &= A_{21} \\
C_{21} &= C_{21} \\
* &= C_{22}
\end{align*}
\]
Now the total number of loads is 12 which is equal to the one for in-core programming, hence we have an I/O minimal algorithm. In this example the memory size is \(6\times N^2/4\), half of the memory size needed for in-core programming even though we reached the performance of in-core programming. There is no need to increase the memory size, we already got the minimum number of I/O transfers possible.

**THE GENERAL ALGORITHM**

We are given two matrices \(A\) and \(B\) of size \(N^2\) to be multiplied (\(N\) a positive integer) and the result is to be stored in the matrix \(C\)

\[
C(N,N) = A(N,N) * B(N,N).
\]

Assume that the total size of available memory is \(M\), and that \(M\) can be written as:

\[
M = L \times P
\]

where \(L\) is the page size and \(P\) is the number of available pages. \(L\) and \(P\) are arbitrary positive integers.

When \(N^2\) is greater than \(L\), which means that a matrix cannot be held in its entirety in one memory page, matrices should be divided into submatrices. In this case we assume that the matrices are divided into \(d^2\) submatrices, where

\[
d = \lceil \sqrt{N^2 / L} \rceil
\]

The ceiling function \(\lceil x \rceil\) gives the least integer \( \geq x \). The range in each dimension of each array will be divided into \(d\) portions; this yields \(d^2\) subarrays for each matrix.

The I/O minimal loading strategy will change depending on matrix and memory size. The number of pages in memory is the dominant factor to determine the loading scheme. We distinguish three cases:

**Case 1:** \(P = 3\)  
There will be exactly one submatrix from each matrix in main memory at all times.

**Case 2:**  
\(P \geq (d+1)d + 1\) if \(d > 2\) and \(P \geq (d+1)d\) if \(d = 2\)  
Here, there is enough space in main memory to load each submatrix once, to use each until it is no longer needed (which means using it \(d\) times).

**Case 3:** all other \(P\)

(a) \(P - 2 \leq d\)  
At each instant, the first page will be devoted to one B submatrix, the second page to one A submatrix, and the remaining \(P - 2\) pages to \(C\) submatrices as much as possible.

(b) \(P - 2 > d\)  
Let \(P = d + 2 + \text{REM}\) where \(\text{REM} > 0\); \(d + 2\) pages will be used as described in Part (a) above, the remaining \(\text{REM}\) pages be used for loading other B submatrices as much as possible.

For all these cases I/O minimality is achieved. More detailed formulations follow below.
CASE 1: \( P = 3 \) means that there are only 3 pages in the memory. In order to do matrix multiplication, we must have at least one \( A \), one \( B \) and one \( C \) submatrix in main memory at each instant. When the multiplication of one group is finished, a new submatrix triple that was not already done must be brought into memory. In this case, it is only possible to keep one submatrix in the memory during the switches to the next instance, guaranteeing I/O minimality, as described in Example 1. This scheme needs three loads at initialization and then two loads at each successive instance which yields a total number of loads equal to:

\[
3 + 2(d^3 - 1)
\]

For each case, we give the ratio of out-of-core method to that of the in-core method. The number of total loads in in-core programming is \( 3d^2 \) (there are \( d^2 \) subarrays of each matrix and there are three matrices).

Overall ratio = \((3 + (d^3 - 1)\ast 2) / (3\ast d^2) = (2\ast d) / 3 + 1 / (3\ast d^2)\)

Net memory saving = \(3\ast d^2 - P = 3\ast (d^2 - 1)\)

CASE 2: \( P \geq (d + 1) \ast d + 1 \) if \( d > 2 \) AND \( P \geq d^2 + d \) if \( d = 2 \)

First, an example of loading scheme where \( d=4 \) is given. Here, "\*" represents the repeated usage of a submatrix which is already in memory.

**Example 3:** \( d=4 \)

```
C11 A11 B11
  *  | A12 B21  |
  *  | A13 B31  |
  *  | A14 B41  |
C12 * * * * | B12  |
  * * * * | B22  |
C13 * * * * | B42  |
  * * * *   | B23  |
  * * * *   | B33  |
  * * * *   | B34  |
  * * * *   | B43  |
C14 * * * * | B24  |
  * * * *   | B34  |
  * * * *   | B44  |
C21 A21 * * * * | A22  |
  * * * *   | A23  |
  * * * *   | A24  |
C22 * * * *   | *    |
1 |---------- 2*d  |----------|---------- (d - 1)*d  |----------|
```

This scheme is I/O minimal because each submatrix is loaded into the main memory only once and used in its entirety (\( d \) times). The number of retrievals for this out-of-core program will be exactly equal to that of in-core programming; the overall ratio is 1.

Net memory space saving = \(3\ast d^2 - d^2 - d - 1 = 2\ast d^2 - d - 1\).
The special case where \( d = 2 \) is described in previous section (Example 2). Notice that in this case, the last slot for \( B_{22} \) is not needed anymore as the free place after \( A_{11} \) can be used for this purpose, because \( B_{22} \) will be used only twice; this is impossible for larger \( d \).

**Case 3:** all other values of \( P \)

a) \( P - 2 \leq d \)

As this case is quite complicated, we distinguish three subcases:

i) \( P - 2 = d \)

ii) \( P - 2 < d \) AND \( d \) is a multiple of \( P - 2 \)

\[ d = K \cdot (P - 2) \]

where \( K \in \mathbb{Z}^+ - \{1\} \) and \( K \leq d/2 \)

iii) \( P - 2 < d \) AND \( d \) is not a multiple of \( P - 2 \)

\[ d = K \cdot (P - 2) + R \]

where \( K, R \in \mathbb{Z}^+ \)

and \( 1 \leq K \leq (d - 1)/2, 1 \leq R < P - 2 \)

The memory pages will be used like this throughout this case:

```
 1 2 3 4 . . . . P
  .  .  .  .  .  .  .  .  .  .
B A C C C C C C
```

Using memory in this way guarantees I/O minimality for the following reason: \( P - 2 \) pages of memory are used totally before removal (the \( C \) submatrices are loaded only once and returned to disk after they are used, never to be needed again), one other slot used for \( A \) submatrices is also kept fixed until this \( A \) submatrix is used totally. It is indispensable to load different \( B \) submatrix at each instant. The loading scheme is illustrated in Example 4.

**Example 4:** \( d=3, P=5 \)

```
B_{11} A_{11} C_{11}
B_{12} * C_{12}
B_{13} * C_{13}
B_{21} A_{12} *
B_{22} * *
B_{23} * *
B_{31} A_{13} *
  * B_{32} *
  * B_{33} *
A_{23} * C_{23}
  * C_{22}
  * C_{21}
B_{23} A_{22} *
B_{22} * *
B_{21} * *
B_{13} A_{21} *
  * B_{12}
  * B_{11}
```
Total number of loads =
\[3 + 2(d - 1) + (d - 1)[2d + (d - 1)(d + 1)] - (d - 1)(d - 1)\]
\[= d^3 + d^2 + d\]  
(3.a.i.1)  
(3.a.i.2)  
(3.a.i.3)  
(3.a.i.4)

For this case, we give explanations of the various terms below:

**Term 3.a.i.1:** 3 is the number of initial loads. Then for the first use of the first row of C submatrices there will be \(d - 1\) C and \(d - 1\) B submatrix loads.

**Term 3.a.i.2:** There are \(d - 1\) remaining uses of the first row of C. For each use there will be \(d + 1\) loads: \(d\) B submatrices and 1 A submatrix loads.

**Term 3.a.i.3:** There are \(d - 1\) remaining rows of C, for each row there will be \(2d\) loads for initial loading of B and C submatrices, plus \((d - 1)(d + 1)\) loads for successive loading of B and A submatrices as described above in Term 3.a.i.2.

**Term 3.a.i.4:** When one row of C submatrices is used \(d\) times, B submatrices will be loaded in their place while we switch to the other row of C submatrices. These B submatrices will be used again just after the switch, for the first use of the new row. This will happen \(d - 1\) times and at each time \(d - 1\) loads will be saved by loading B submatrices in the place of already used C submatrices.

The overall ratio will be \((d^3 + d^2 + d)/(3d^2) = d + 1 + 1/d\).

The net memory saving is \(3d^2 - (d + 2)\).

**ii )** \(P - 2 < d\) and \(d\) is a multiple of \((P - 2)\)

\[d = K*(P - 2)\]

where \(K \in \mathbb{Z}^+ - \{1\}\) and \(K \leq d/2\)

The sketch of loading C submatrices is as follows:
So the procedure in Case 3.a.i can be followed exactly, except that instead of being able to load one whole row of C submatrices, now only $P - 2$ or $d/K$ elements of one row of C submatrices will be loaded in the remaining $P - 2$ pages of the memory. Again after using those C submatrices $d$ times, they will be removed from the memory taking the other group of $d/K$ elements of the next row of C submatrices. This will continue until all $d/K$ elements of all rows of C submatrices are finished. Then we will repeat the procedure for the next group of $d/K$ elements of all rows until all $K$ of them are finished.

Again, the loaded $P - 2$ C submatrices are used in their entirety before they are removed from memory. But now the A submatrices corresponding to these C submatrices will be kept only $P - 2$ times (or $d/K$ times), because after $P - 2$ instances, the first loaded $C_{ij}$ will be used which will need a new $A_{ik}$. Again, I/O minimality is achieved as all the submatrices are kept in memory as long as possible and also using the pages which are left from the already finished C submatrices to load B submatrices during the switch between successive C submatrix rows. As the $A_{ij}$s at the end of each column are kept fixed (for minimality) during passing to the next column, the loading order of the new row group must be reversed as can be seen from the arrows.

The total number of loads is therefore:

$$3 + 2 \times (d/K - 1)$$
$$+ (d - 1) \times [(1 + d/K) + (1 + d/K) \times (d - 1)]$$
$$+ (K - 1) \times \left\{(1 + 2d/K) + (d - 1) \times (1 + d/K) + (d - 1) \times [(1 + d/K) + (1 + d/K) \times (d - 1)]\right\}$$
$$- (K - 1) \times 1$$
For an explanation of these terms as well as for an example of this loading scheme, we refer to (Akyildiz, 1993).

If we substitute \( d/(P-2) \) for \( K \), the total number of loads will be:

\[
1 + \frac{d^3(P-1)(P-2) + d(P-3)(P-2)}{(P-2)}
\]

The overall ratio is

\[
d^2(P-1)/(3*(P-6)) + (P-3)/(3*(P-6)*d) + 1/(3*d^2).
\]

The net memory saving is \( 3*d^2 - P \).

\[\text{iii) } \quad P-2 < d \quad \text{and} \quad d \text{ is not a multiple of } P-2\]

\[d = K*(P-2) + R \quad \text{where } K,R \in \mathbb{Z}^+ \text{ and } 1 \leq K \leq (d-1)/2, \quad 1 \leq R < P-2\]

The scheme of loading the \( C \) submatrices can be represented graphically as follows:

```
First P-2 columns

Kth P-2 columns

Last R columns
```

```

\[
\text{Region 1} \quad \text{Region 3}
\]

\[
\text{Region 2}
\]
```

**Figure 2:** The loading scheme in Region 1 of Case 3.a.iii

All loads in Region 1 are performed exactly as in Case 3.a.ii to supply I/O minimality, until we get to the last group of columns whose width is less than \( P-2 \). As this last group is not wide enough to use all remaining \( P-2 \) pages (only \( d - (P-2)/K \) \( (=R) \) columns remain), the scheme will switch to a "column" loading strategy. So, \( P-2 \) elements of the corresponding column are loaded into the remaining \( P-2 \) pages. While
doing this it must be remembered where the arrow was pointing at the end of the Kth group of P - 2 C submatrices. If the arrow was pointing up, the P - 2 elements starting from the first row of that group must be loaded (for obtaining minimality by keeping the last A submatrix fixed in the memory), and if the arrow was pointing down, the P - 2 elements starting from the last row of that group of C submatrices should be loaded into memory.

After finishing all K*R element groups, each of which has P - 2 elements, the last region, Region 3, is reached whose dimension is R^2. In passing from Region 2 to Region 3 again the direction of the last arrow in the horizontal axis (out or in) must be remembered as well as where the arrow was pointing in the vertical axis (up or down). In this last third region, the last R^2 C submatrices will be used. We remark that in this region R + 2 pages will be used for matrix multiplication, R pages to hold C submatrices and two pages for one A, one B submatrix. The other P - 2 - R pages will be used during the switches between successive C submatrix row groups. The necessary A submatrices will be loaded in these pages which are used both, just before and just after the switches. An example sketch is given where d = 8 and P - 2 = 3 to clarify the loading procedures.

![Figure 3: Sample loading scheme for Case 3.a.iii](image)

The total number of page loads can therefore be calculated as follows:

; the first region
3 + 2*(P - 3) + (d - 1)*(P - 1) + (d - 1)*[(P - 1) + (d - 1)*(P - 1)] + (K - 1)*{ [(P - 2) + d*(P - 1)] + (d - 1)*[(P - 1) + (P - 1)*(d - 1)] } - (K - 1)*1

; the second region
+ 2*(P - 2) + (d - 1)*(P - 1) + (R - 1)*[(P - 1)*d] + (K - 1)*[R*d*(P - 1) + (P - 1)]
\[
\begin{align*}
\text{the third region} \\
&= 2^K(P - 3) + K(P - 1)d(d + R) + R^2(1 + d) + R(d + 2 - P) + P - 2
\end{align*}
\]

The explanations of the terms above are given in (Akyildiz, 1993); here we merely note that Region 1 is exactly the same as Case 3.a.ii. Notice that if \( P - 2 \) is replaced by \( d/K \) in all formulae of Case 3.a.ii, the same terms of Case 3.a.iii from 1 to 5 will be obtained.

The overall ratio will be:

\[
\left\{ 2^K(P - 3) + K(P - 1)d(d + R) + R^2(1 + d) + R(d + 2 - P) + P - 2 \right\} / (3d^2)
\]

The net memory saving is equal to \( 3d^2 - P \).

**Case 3:** all other \( P \), \( b ) P - 2 > d \)

The condition can be rewritten as:

\[
P = d + 2 + \text{REM} \quad \text{where} \quad 1 \leq \text{REM} < d^2 - 1.
\]

Here, 1 + \( d \) pages will be used as described in Case 3.a.i: \( d \) pages for one row of C submatrices and one page for the corresponding A submatrices. The remaining pages (REM) will all be used to store B submatrices. All C and A submatrices loaded will reside in memory until they are used \( d \) times. When A and C submatrices are used completely before the switch to the next row, all the REM pages will be full of B submatrices which will be used again after the switch. Hence I/O minimality is guaranteed. There are again different cases to be considered separately, although the loading scheme is the same:

\begin{itemize}
\item[i)] \( 1 \leq \text{REM} < d^2 - 2^d + 1 \)
\end{itemize}

There are REM + \( d \) pages to load B submatrices, one of them is originally fixed for B. These REM + \( d \) pages are not able to hold all \( d^2 \) B submatrices, hence it is unavoidable to re-load some B submatrices. The total number of loads in this case is equal to:

\[
3d^2 + (d - 1)(d^2 - \text{REM} - d).
\]

Here \( 3d^2 \) is the initial loading of all submatrices. The rest is the number of these unavoidable additional loads: the number of B submatrices that need reloading is \( d^2 - \text{REM} - d \). There are \( d \) switches for row-wise loaded C submatrices, so in \( d - 1 \) of them additional B submatrices will be loaded again from disk.

The overall ratio is \( = d/3 + 1/3 + (1/d)[(l - \text{REM})/3] + (1/d^2)(\text{REM}/3) \).

The net memory saving is equal to \( 3d^2 - P \). The example below illustrates this loading scheme.

**Example 4:** \( d = 3, \text{REM} = 2 \) and \( P = 7 \); additional reloads of B submatrices are shown underlined.
ii) \( d^2 - 2d + 1 \leq \text{REM} < d^2 - d \)

In this case, in odd numbered switches \( d - 1 \) B submatrices will be loaded, but in even numbered switches there will be only \( d^2 - d - \text{REM} \) submatrix loads. The total number of loads will therefore be the sum of \( 3d^2 \) and the necessary reloads of B submatrices in even and odd numbered switches; thus we can state:

If \( d \) is odd, the total number of loads is

\[
3d^2 + \lceil d/2 - 1 \rceil (d^2 - \text{REM} - 1).
\]

If \( d \) is even, the total number of loads is

\[
3d^2 + ((d - 2)/2)(d^2 - \text{REM} - 1) + (d^2 - d - \text{REM}).
\]

iii) \( d^2 - d \leq \text{REM} < d^2 - 1 \)

In this case, there will be no extra reloading of B submatrices on even numbered switches. Additional loads, which are equal to \( d^2 - (\text{REM} + 1) \) will take place only on odd numbered switches. Hence the total number of loads is:

\[
3d^2 + \lceil d/2 - 1 \rceil (d^2 - \text{REM} - 1).
\]

If we order all of the cases as the number of pages in the memory increases, we will get the following progression:
Hence, when we increase the number of pages in the memory we will switch cases. At each switch two different formulae approach the same value; for the verification of this claim, we refer again to (Akyildiz, 1993).

CONCLUSIONS

An I/O minimal algorithm for matrix multiplication was developed and implemented. The problem is decomposed into three main cases. For each case, we derive a separate loading scheme and show local I/O minimality.

Several experiments using this implementation were conducted; for more details see (Akyildiz, 1993). Examining the results of a gradual increase in memory, we deduce that the worst case is having only three pages in the memory, which is the smallest available memory for given d. When the number of pages, in other words, the memory size, increases we switch to better cases where the number of I/O transfers is smaller. But when P reaches a certain value, which is \( d^2 + d + 1 \), then there is no need to increase the memory size, as the number of loads is now equal to the one for in-core programming. So we can have the same performance of in-core programming with much smaller memory requirements.

REFERENCES


