COMPILER-DRIVEN RUNTIME I/O MINIMIZATION

Wendy Zhang
Southern University at New Orleans
Department of Computer Information Systems
New Orleans, USA, LA 70126
E-mail: wzhang@suno.edu

and

Ernst L. Leiss
University of Houston
Computer Science Department
Houston, USA, TX 77204
E-mail: coscel@cs.uh.edu

ABSTRACT

Most scientific programs have large input and output data sets that require out-of-core programming or use virtual memory management (VMM). VMM is not an effective approach because it results easily in substantial performance reduction. In contrast, compiler driven I/O management will allow a program’s data sets to be retrieved in parts, called blocks or tiles. Out-of-core programming defines the partitions of the data sets and the control of I/O. Comanche (COmpiler Managed CaCHE) is a compiler combined with a user level runtime system that can replace and out-perform standard VMM for out-of-core programs. In addition, this system does not require any special services from the operating system and does not require modification of the operating system kernel. Our research aims at improving the applicability of Comanche to achieve high performance of I/O-intensive out-of-core computations. This research shows that block mapping I/O management may improve the performance of out-of-core problems by one order of magnitude or more. We have studied the relationship between data access patterns and data locality optimization and created a tool for compile time data transfer analysis. Our extension of Comanche involves the use an out-of-core programming schema for optimizing locality in loop nested to minimize I/O.

Key Words: I/O Management, VMM, Out-of-core, Compiler, Tile mapping
1 INTRODUCTION

The speed gap between processors and disks continues to increase as VLSI technology advances at a tremendous rate. As a result, disk I/O has become a serious bottleneck for many high performance computer systems. Hence, it is critically important to be able to construct I/O minimal programs [10].

The size and complexity of applications, both in the scientific and the commercial world, have increased. As the data sets required by those applications exceed the capacity of main memories, the computation becomes an out-of-core computation. Processing out-of-core data requires staging data in smaller granules that can be fit in the main memory. Data required for the entire computation have to be fetched from files on disk so that improving disk I/O becomes extremely important.

Much research has been done on virtual memory management (VMM) and other related operating systems (OS) concepts, I/O subsystem hardware, and parallel file systems. Each of those approaches contributes to some degree to I/O performance, but they all lack a global view of application behavior, which limits their effectiveness.

Parallel I/O is a cost-effective way to address some I/O issues. The wide availability of inexpensive powerful PC clusters with high-speed networks makes parallel I/O a viable approach. Parallel I/O subsystems have increased the I/O capabilities of parallel machines significantly but much improvement is still needed to balance the CPU performance. The variety (private disks, shared disks or a combination of both) in the I/O architectures makes it difficult to design optimization techniques that reduce the I/O cost. The problem has become more severe since the size and complexity of applications have increased tremendously [13].

OS designers offer the handling of I/O activity via virtual memory management (VMM). Research on this approach considers the use of smart virtual memory, techniques which reshape the data reference patterns to exploit the given hardware facilities and system software, and replacement policies. Overall these techniques assume a considerable amount of help from the hardware.

Recently, a number of run-time libraries for out-of-core computations and a few file interfaces have been proposed, among them SIO [12], MPI-IO [2], and an extension of the traditional Unix file I/O interface for handling the parallel accesses to parallel disk subsystems [8]. The parallel file systems and run-time libraries for out-of-core computations provide considerable I/O performance, but they require much effort from the user; also they are not portable across a wide variety of parallel machines with different disk subsystems.

The difficulty of handling out-of-core data and writing an efficient out-of-core program limits the performance of high performance computers. Execution of some out-of-core programs does not perform well when they rely on the virtual memory management (VMM) system. There is a clear need for compiler directed explicit I/O for out-of-core computations [1] [3] [4] [5] [6] [15][17].

In this paper, we concentrate on the compiler-based approach to the I/O problem. The main rationale behind this approach is the fact that the compiler has unique information about the data needs of the program. The compiler can examine the size and shape of the data and the overall access pattern of the application. Compiler driven I/O management should generate code to restructure out-of-core data, computation, and the management of memory resources. A compiler combined with a user level runtime system can replace and outperform standard virtual memory management for out-of-core problems [14]. A compiler with a good combination of file layouts on disks and loop transformations is successful at optimizing programs, which depend on disk-resident data in distributed-memory machines [7].

Comanche (an acronym for COmpiler MANaged caCHE) is a compiler run time I/O management system [14]. It is a compiler combined with a user level runtime system. More details about Comanche will be given in Section 2.

Blocking or tiling [11] is a well-known technique that improves the data locality of numerical algorithms. Blocking can be used to achieve locality for different levels of memory hierarchy. [9] and [16] describe a locality optimization algorithm that applies a combination of loop interchanges, skewing, reversal, and tiling to improve the data locality of loop nests. [10] describes an I/O minimal algorithm where for a given amount of space the number of retrievals of blocks from external memory is minimal. [10] also points that the notion of I/O minimal depends on the block size available in the memory.
Out-of-core matrix multiplication is used in this paper to study the importance of blocking. An I/O minimal algorithm for matrix multiplication is sketched which forms the basis of this work. In Section 3, we discuss the minimal amount of space required to carry out the computation as well as the I/O scheme used with Comanche system. In testing our matrix multiplication algorithm, it became clear that the size of the blocks for the sub-matrices plays a very important role. We will compare the performance for different block sizes.

An out-of-core stepped problem is used in this paper to study the importance of combining data layout and loop transformation. In Section 4, we discuss the stepped problem and the amount of space required to carry out the computation as well as the I/O scheme used with the Comanche system. We compare the performance for different block sizes. In Section 5, we draw conclusions from the results of our study.

2 COMANCHE

Comanche (an acronym for COmpiler MANaged caCHE) is a compiler run time I/O management system [14]. It is a compiler combined with a user level runtime system and effectively replaces virtual memory management by allowing direct control over which pages are retained in the active memory set. The current Comanche system is running under the RedHat 5.0 Linux operating system on a PC with a PentiumPro (160MHz or faster) processor and 64MB or more RAM.

The standard entity in the Comanche runtime system is a two-dimensional array. Higher dimensions can be supported, or else the accesses can be translated to two-dimensional accesses. Data are assumed to be in row major layout on disk. The ideal array is a square matrix.

There are two structures declared in the Comanche system. One (block_s) is a block structure used to buffer a block of data, the other structure is a matrix structure (array_s) that holds the information of the matrix on disk and has buffers to hold several blocks in memory.

```c
typedef structure block_s {
    double *data;
    int flag;
    int refcnt;
    int rowindex;
    int columnindex;
    int nrow;
    int ncol;
    int blockno;
} Block;

typedef struct array_s {
    int nblocks, nelems, bsize;
    int elesize;
    FILE *fp;
    long offset;
    char *name;
    int *map;
    int nbuffers;
    Block **buffers;
    int victim, flag, total_mem;
} Matrix;
```

Besides these structures, there are several functions in Comanche. Two major functions are `attach_block` and `release_block` which are used to perform the block mapping.

When a data set is too large to fit into memory, VMM will map the array into a one-dimensional vector and then that vector is cut up into blocks. Comanche will take a row of an array and cut it up into blocks, so that a block only corresponds to a single row or a sub-matrix. Through the use of the functions provided by Comanche, the I/O behavior is under the control of the resulting out-of-core program.

The `attach_block` and `release_block` functions tell the runtime system that the block is to be mapped into memory; then an address to the cached data is to be returned. The runtime system will not reclaim memory that has been attached as long as it remains attached. It does not matter how much time has passed since the block was last referenced. The out-of-core program manages the duration of mapped data and ensures that the number of attach operations will not over-fill the available memory before the data’s subsequent release.

3 BLOCK MAPPING – OPTIMAL BLOCK SIZE

Many scientific programs with large input and output data sets use virtual memory management (VMM). VMM is not an effective approach because it results easily in substantial performance reduction. In contrast,
compiler driven I/O management will allow a program’s data sets to be retrieved in parts, called blocks. In out-of-core programming, the programmer defines the partitions of the data sets and controls I/O explicitly. Our study shows that block mapping I/O management may improve the performance of out-of-core problems by one order of magnitude or more. The study also shows that the block size chosen to partition the data sets plays an important role in compiler driven I/O management.

3.1 Block Matrix Multiplication

The object of our study of the Comanche run time system is matrix multiplication. Assume that there are three matrices A, B, and C of size (N, N) on disk. We will multiply A by B and store the result in matrix C. To store matrices A, B, and C in memory requires $3N^2$ space. If N is large, the total amount of main memory available is less than the needed space of $3N^2$. Since the data cannot be entirely loaded into memory, the problem becomes out-of-core.

The traditional way of coding this in C is something like this:

```c
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++) {
        C[i, j] = 0;
        for (k = 0; k < N; k++)
            C[i, j] += A[i, k] + B[k, j];
    }
```

Note that although the same row of A is reused in the next iteration of the middle loop, a large volume of data used in the intervening iterations may be replaced. During processing, a virtual memory system would do much swapping, which is very time consuming.

One solution to solve this out-of-core problem is to split each matrix into several sub-matrices (blocks) of size (M, M). Specially, if the dimensions of the matrix are divisible by the dimensions of the block we can use an algorithm suggested by the following observations.

Assume M is one half of N; then each matrix can be split into four sub-matrices of size (N/2, N/2). Schematically we have:

```
  A  
  |   |
  |   |
  X  
  |   |
  |
  B  
  |   |
  |   |
  =  
  |   |
  |   |
  C  
```

The $C_{ij}$’s are defined as follows:

\[
C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21} \\
C_{12} = A_{11} \times B_{12} + A_{12} \times B_{22} \\
C_{21} = A_{21} \times B_{11} + A_{22} \times B_{21} \\
C_{22} = A_{21} \times B_{12} + A_{22} \times B_{22}
\]

Assuming all $C_{ij}$ are initialized to zero, the following instructions can be executed in any order:

```
C_{11} = C_{11} + A_{11} \times B_{11} \\
C_{11} = C_{11} + A_{12} \times B_{21} \\
C_{12} = C_{12} + A_{11} \times B_{12} \\
C_{12} = C_{12} + A_{12} \times B_{22} \\
C_{21} = C_{21} + A_{21} \times B_{11} \\
C_{21} = C_{21} + A_{22} \times B_{21} \\
C_{22} = C_{22} + A_{21} \times B_{12} \\
C_{22} = C_{22} + A_{22} \times B_{22}
```

A brute force approach might retrieve each of the twenty-four sub-matrices (blocks) regardless of reusability. In out-of-core programming, we can sequence the instructions in order to reuse sub-matrices as often as possible. We use the Comanche runtime function `block_attach` when we need a block and `block_release` when we no longer need that block.

Assume that the amount of available main memory is equal to four sub-matrices, $4 \times N^2/4$. The following sketches an efficient approach (see [10]):

```
C_{11} A_{11} B_{11} // Attach C_{11}, A_{11}, B_{11} \\
C_{11} B_{21} A_{12} // Release B_{11}, attach B_{21}, A_{12} \\
C_{12} A_{11} B_{12} // Release B_{21}, attach C_{12}, B_{12} \\
C_{12} B_{22} A_{12} // Release B_{12}, attach B_{22} \\
C_{21} A_{21} B_{11} // Release all, attach C_{21}, A_{21}, B_{11} \\
C_{21} B_{21} A_{22} // Release B_{11}, attach B_{21}, A_{22} \\
C_{22} A_{21} B_{12} // Release B_{21}, C_{21}, Attach C_{22}, B_{12} \\
C_{22} B_{22} A_{22} // Release B_{12}, attach B_{22}, A_{22} // Release all
```
In this scheme, we only retrieve sixteen blocks from disk and store four blocks to disk. We use a similar, but more complicated scheme if N is not divisible by M.

### 3.2 Experiments

We have implemented the algorithm suggested in Section 3.1 in an out-of-core program written in the C language. The C compiler under Linux generates working code using the Comanche runtime system. On a single processor PC with a PentiumII, 333MHz microprocessor, the code was run under Redhat Linux 5.0 in command mode. The system used for the actual performance has 64MB of memory of which about 50MB are available to the program.

We have run one set of experiments for 1024 x 1024 double precision matrix multiplication. The total data set space is 1024 x 1024 x 8 x 3=24MB. Another set of experiments was run for 2048 x 2048 matrices, with a total data set of 96MB. Data files are initialized with random values in the interval (-1, +1).

The Linux implementation requires that we execute tests on an unloaded machine and use elapsed time. For each experiment, we have recorded the following:
- **Block size**: given as number of rows and columns;
- **Elapsed**: time used by executing the whole program (given in hour:min:sec);
- **Page fault**: number of page faults in the virtual memory system;
- **# of Seek**: number of seek function calls used in Comanche;
- **# of Read**: number of read function calls used to map data from disk to memory in Comanche;
- **# of Write**: number of write function calls used to map data from memory to disk in Comanche;
- **Memory usage**: amount of memory used to hold the blocks in memory.

During initial testing we found that block mapping large files has a significant impact on the system resources. We did additional tests to assess the impact of different block sizes on running time.

#### 3.2.1 Matrix Multiplication

The original code for matrix multiplication $C = A \times B$ under VMM is given in Figure 3.1. The performance values are shown in Table 3.1. Compared with block mapping, this method causes more page faults so the elapsed time is generally very large (over half an hour for the smaller problem and 4.5 hours for the larger).

```c
double matmul(int n, double *A){
    int i, j, k;
    double dval, sum = 0.0;
    double *a, *b, *c;
    double *B, *C;
    B = A + n*n;
    C = B + n*n;
    a = A;    // A[i]
    b = B;    // B[i]
    c = C;    // C[i]
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            dval = 0.0;
            for (k = 0; k < n; k++)
                dval += a[k] * b[k*n+j];
            c[j] = dval;
            sum += dval;
        }
        a += n; c += n;
    }
    return sum;
}
```

Figure 3.1 C Code for Matrix Multiplication under VMM
Regular matrix multiplication (VMM): \( C = A \times B \)

<table>
<thead>
<tr>
<th>Matrix size</th>
<th>Elapsed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024x1024</td>
<td>31:05.7</td>
</tr>
<tr>
<td>2048x2048</td>
<td>4:31:27</td>
</tr>
</tbody>
</table>

Table 3.1 Performance of Matrix Multiplication for Different Matrix Sizes

3.2.2 The Dimensions of the Matrix are Divisible by the Dimensions of the Block

To implement block multiplication, we divided the matrix into blocks. Each block has size \( M \times M \). Then we perform the block multiplication described in Section 3.1. We map one row of blocks of matrix \( A \) into the memory as well as one block of \( B \) and one block of \( C \). After each block multiplication, we store the block of matrix \( C \) and release the block of matrix \( B \). We release all blocks after we finish one row of blocks. We start over for another row of blocks until the whole matrix multiplication is finished. The block multiplication code is given in Figure 3.2.

```c
double matmul1(int n, Matrix *mptr, int bs)
{
    int i, j, k, bno, cno, ano, newbindex;
    int bindex = n, cindex = n + n;
    double dval, *a, *b, *c;
    double sum = 0.0;
    int bkrow = n/bs;  // the number of blocks each row
    for (i = 0; i < n; i += bs)
    {
        for (j = 0; j < n; j += bs)
        {
            ano = i/bs * bkrow;    // get the block number of a
            a = block_attach(mptr, ano, -1, i, 0, bs, bs); // attach a[i][0]
            assert(a);
            // get block number of c and attach c block
            cno = (cindex + i)/bs * bkrow + j/bs;
            c = block_attach(mptr, cno, 0, cindex+i, j, bs,bs); assert(c);
            // get block no of b and attach b block
            bno = bindex/bs * bkrow + j/bs;
            b = block_attach(mptr, bno,-1, bindex, j, bs,bs); assert(b);
            sum += sub_matmul(a, b, c, bs, bs, bs);
            // calculate for the rest sub-matrix in row
            for (k = 1; k < bkrow; k++)
            {
                block_release(mptr, bno); // release the b block
                // calculate the block number of new sub-block of b
                newbindex = bindex + k * bs;
                bno = newbindex / bs * bkrow + j/bs;
                b=block_attach(mptr,bno,-1,newbindex,j,bs,bs); assert(b);
                ++ano; // increase the block number of a
                a = block_attach(mptr,ano,-1,i,k*bs,bs,bs); assert(a);
                sum += sub_matmul(a, b, c, bs, bs, bs);
            }
        // release sub-blocks of b and c
        block_release(mptr, bno);
        block_release(mptr, cno);
        }
    // release the row of sub-blocks of a
    for (k = 0; k < bkrow; k++)
    {
        block_release(mptr, ano--);
    }
    return sum;
}
```

Figure 3.2 C Code for Block Multiplication – Dimensions of the Matrix are Divisible by the Block Size
We have measured the performance of the matrix multiplication for different block sizes, namely 512 x 512, 256 x 256, 128 x 128, 64 x 64, 32 x 32, 16 x 16, and 8 x 8. We applied these different block sizes to the data sets of 24MB and 96MB. The performance values are listed in Tables 3.2 and 3.3.

We observe that block multiplication uses only 24.5% to 55.0% of the execution time of regular multiplication (VMM). The one exception is for a laughably small block size (8 x 8).

<table>
<thead>
<tr>
<th>Block Size</th>
<th>512 x 512</th>
<th>256 x 256</th>
<th>128 x 128</th>
<th>64 x 64</th>
<th>32 x 32</th>
<th>16 x 16</th>
<th>8 x 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elapsed</td>
<td>17:06.0</td>
<td>09:52.0</td>
<td>07:30.0</td>
<td>07:59.0</td>
<td>07:46.0</td>
<td>13:50.0</td>
<td>42:33.0</td>
</tr>
<tr>
<td># of Seek</td>
<td>40,960</td>
<td>122,880</td>
<td>409,600</td>
<td>1,474,560</td>
<td>5,570,560</td>
<td>21,626,880</td>
<td>85,196,800</td>
</tr>
<tr>
<td># of Read</td>
<td>10,240</td>
<td>20,480</td>
<td>40,960</td>
<td>81,920</td>
<td>163,840</td>
<td>327,680</td>
<td>655,360</td>
</tr>
<tr>
<td># of Write</td>
<td>8,388,863</td>
<td>3,146,199</td>
<td>1,311,911</td>
<td>593,607</td>
<td>292,103</td>
<td>186,759</td>
<td>267,911</td>
</tr>
</tbody>
</table>

Table 3.2 Performance for Matrices of Size 1024 x 1024, Dimensions Divisible by Block Size

3.2.3 The Dimensions of the Matrix are Not Divisible by the Dimensions of the Block

To implement block multiplication, we have divided the matrix N x N into blocks of size M x M. Since the dimensions of the matrix are not divisible by the dimensions of the block, the last block in a row of blocks is of size M x M % N, the last row of blocks is of size N % M x M, and the last block of a matrix is of size N % M x N % M. Before we can perform block multiplication, we save the block dimensions in a table. Then we apply the block multiplication sketched in Section 3.1. Since we have to maintain different dimensions of blocks and perform different dimensions macro multiplication, the code is more complicated (code is not given here due to space limitations).

We measured the performance of the matrix multiplication for different block sizes, namely 500 x 500, 250 x 250, 130 x 130, 65 x 65, 33 x 33, 17 x 17, and 9 x 9. We applied these different block sizes on the data sets of 24MB and 96MB. The performance values are listed in Tables 3.4 and 3.5.

We observe that our block multiplication uses only 49.2% to 72.7% of the execution time of regular multiplication. There one exception is again a tiny block size (9 x 9). We also see that our block multiplication uses about double the time compared to the case where the dimensions of the matrix are divisible by the dimensions of the block.

<table>
<thead>
<tr>
<th>Block Size</th>
<th>500 x 500</th>
<th>260 x 260</th>
<th>130 x 130</th>
<th>65 x 65</th>
<th>33 x 33</th>
<th>17 x 17</th>
<th>9 x 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elapsed</td>
<td>23:28.0</td>
<td>17:05.0</td>
<td>15:18.0</td>
<td>15:45.0</td>
<td>16:25.0</td>
<td>22:35.0</td>
<td>43:09.0</td>
</tr>
<tr>
<td># of Seek</td>
<td>76,800</td>
<td>122,880</td>
<td>409,600</td>
<td>1,474,560</td>
<td>5,570,560</td>
<td>19,676,880</td>
<td>67,706,880</td>
</tr>
<tr>
<td># of Read</td>
<td>61,400</td>
<td>102,400</td>
<td>368,640</td>
<td>1,392,640</td>
<td>5,406,720</td>
<td>19,363,840</td>
<td>67,123,200</td>
</tr>
<tr>
<td># of Write</td>
<td>15,360</td>
<td>20,480</td>
<td>40,960</td>
<td>81,920</td>
<td>163,840</td>
<td>327,680</td>
<td>655,360</td>
</tr>
<tr>
<td>Memory usage</td>
<td>10,000,351</td>
<td>3,245,271</td>
<td>1,353,191</td>
<td>612,183</td>
<td>309,783</td>
<td>192,639</td>
<td>235,359</td>
</tr>
</tbody>
</table>

Table 3.4 Performance for Matrices of Size 1024 x 1024, Dimensions Not Divisible by Block Size
### 4 STEPPED PROBLEMS

Scientific computations such as Fast Fourier Transforms (FFT) and Multi-grid methods use increasing and decreasing strides. Stepped programs traverse a two-dimensional grid using step sizes that are powers of two. Figure 4.1 illustrates this on a portion of the array with step sizes (A) 1, (B) 2, (C) 4, and (D) 8. A naively implemented stepped program running under Comanche is considerably slower than under VMM, in real test cases. Such programs reveal limitations of the simpler, row base API, as the stride grows beyond the size of a single page; reading an entire row will consist mostly of dead weight [14].

```cpp
void stepped(int n, double [][]A) {
    int i, j, k;
    double d0, d1;
    for (k=1; k<n; k<<=1) {
        for (i=0; i<n-k; i+=k) {
            for (j=0; j<n-k; j+=k) {
                d0=A[i][j]/2.0;
                A[i][j]=d0 +d1;
            }
        }
    }
}
```

Figure 4.1 Stepped Access Patterns

Since the primary reason for the poor performance in this example is the high amount of dead weight that occurs when the strides become large, we are studying subarray optimization to avoid reading the dead weight in the rows. Subarrays represent mappings from remote storage to local storage using non-unit strides. An example of an array starting at 2, and consisting of every fourth element, up to 24, is illustrated in Figure 4.2. In this way, we map an array of size 24 into an array of size 6 to save memory space and reduce I/O bandwidth. The initial results of our experiments using this naïve approach were disappointing.

#### 4.1 Stepped Calculation

When we take a closer look at the stepped program, we realize that the difficulty of this access pattern is that it is not just crisscross but also overlapped. It appears that the performance of the stepped calculation cannot be improved by using data mapping optimization alone. When two adjacent loops have the same loop limits, they can sometimes be fused into a single loop. To solve our stepped problem, there is no single loop transformation method or mapping pattern available. As a result, we combined tiling, subarrays, loop fusing, and the seeker/reaper paradigm of Comanche in combination.
Since our program is out-of-core, we cannot load the whole array into memory. We tile the array into several super-rows (blocks). In each tile, we seek and attach two rows at a time. First, we attach two rows to perform the inner loop (column control) of step size 1. Then we have to fuse a stepped loop with row and column loop control to seek another row to perform. When we fuse the stepped loop into the row loop, we have to keep the original reference order. There is a loop dependence, for instance, before we can perform the step size 2 calculations on rows 0 and 2, since both rows must have already completed the step size 1 calculation.

Because of the loop dependence, we have to seek half a block plus 1 element ahead for each block. During the calculation, we release any row which is no longer needed in future calculations. After finishing the calculation on one super-row, we shrink the first row of the block into a sub-array for further calculation. We perform stepped calculation on the sub-array starting with step size 1 and return the final results to the original rows. We use the reaper to release all the rows and terminate the calculation. We have added to Comanche a subarray data structure:

```c
typedef struct type_t {
    double *data;
    int nrow;
    int ncol;
    int start;
    int stop;
    int step;
} Subarray;
```

To illustrate this process, assume we have a matrix of size 12 x 12 and a block size of 4; the calculation is denoted as \( \otimes \). Here is the complete program; we assume that \( R_i \) denotes row \( i, i = 0, 1 \ldots, 11 \):

1. seek \( R_0 \), attach it;
2. attach \( R_1 \) and \( R_2 \);
3. \( R_0 \otimes R_1 \) step size 1, \( R_1 \otimes R_2 \) step size 1;
4. release \( R_1 \);
5. attach \( R_3 \) and \( R_4 \);
6. \( R_2 \otimes R_3 \) step size 1, \( R_3 \otimes R_4 \) step size 1;
7. release \( R_3 \);
8. \( R_0 \otimes R_2 \) step size 2;
9. attach \( R_5 \) and \( R_6 \);
10. \( R_4 \otimes R_5 \) step size 1, \( R_5 \otimes R_6 \) step size 1;
11. release \( R_5 \);
12. \( R_2 \otimes R_3 \) step size 2;
13. release \( R_2 \);
14. put \( R_6 \) into subarray;
15. attach \( R_7 \) and \( R_8 \);
16. \( R_6 \otimes R_7 \) step size 1, \( R_7 \otimes R_8 \) step size 1;
17. release \( R_7 \);
18. \( R_4 \otimes R_5 \) step size 2;
19. attach \( R_9 \) and \( R_{10} \);
20. \( R_8 \otimes R_9 \) step size 1, \( R_9 \otimes R_{10} \) step size 1;
21. release \( R_9 \);
22. \( R_6 \otimes R_8 \) step size 2;
23. release \( R_6 \);
24. put \( R_4 \) into subarray;
25. attach \( R_{11} \);
26. \( R_8 \otimes R_{11} \) step size 1;
27. release \( R_{11} \);
28. \( R_9 \otimes R_{10} \) step size 2;
29. release \( R_{10} \);
30. put \( R_5 \) into subarray;
31. inside subarray \( R_0 \otimes R_4 \) step size 1;
32. inside subarray \( R_4 \otimes R_8 \) step size 1;
33. inside subarray \( R_6 \otimes R_8 \) step size 2;
34. write back to \( R_0 \) and release \( R_0 \);
35. write back to \( R_4 \) and release \( R_4 \);
36. write back to \( R_8 \) and release \( R_8 \).

Note that we can change the execution order if this does not violate the data dependences. This schema performs tiled stepped calculations without subarrays.

4.2 Experiments

We have implemented the algorithm suggested in Section 4.1 in an out-of-core program written in the C language. The platform is the same as it is used in block mapping (refer to Section 3.2).

We have run one set of experiments for a double precision matrix of size 3072x3072 involved in the stepped calculation. The total data set space is 3072 x 3072 x 8 = 72MB. Another set of experiments was run for the size 6144 x 6144 with a total data set space of 288MB. Data files were initialized with random values.
in the interval \((-1, +1)\). For each experiment, we have recorded the block size, elapsed time, page fault, number of seek, read, and write, and memory usage in Comanche (refer to Section 3.2).

### 4.2.1 Stepped Calculations

In ordinary stepped calculations, the matrix \(A\) is mapped into memory by an `mmap` system call. Virtual memory is tested by mapping files into memory in the conventional way. The original code for the stepped calculation is given in Figure 4.3.

```c
void stepped(int n, double *A) {
    int i, j, k; int step;
    double d0, d1; *a, *ap;
    for (k = 1; k < n; k <<= 1) {
        step = k*n; a = A; ap = A + step;
        for (i = 0; i < n - k; i += k)
            for (j = 0; j < n - k; j += k)
                d0 = a[j]/2.0; d1 = (ap[j] + a[j+k]+ ap[j+k])/6.0;
                a[j] = d0 + d1;
        a += step; ap += step;
    }
}
```

Figure 4.3  Original C Code for Stepped Calculation

The performance values are given in Table 4.1. Compared with the tiled stepped calculation, this method causes more page faults so the elapsed time is generally large.

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Elapsed</th>
</tr>
</thead>
<tbody>
<tr>
<td>3072 x 3072</td>
<td>07:56.8</td>
</tr>
<tr>
<td>6144 x 6144</td>
<td>1:06:52</td>
</tr>
</tbody>
</table>

Table 4.1 Performance of Stepped Calculations

### 4.2.2 Tiled Stepped Calculations with Subarrays

To implement tiled stepped calculations, we divided the matrix into blocks. Each block has size \(M\). Then we perform the stepped calculation sketched in Section 4.1. We map two rows of the matrix \(A\) into memory. After one round of calculation, we release the row which is no longer needed in future calculations. We start over for another row until the whole stepped calculations are finished.

For a matrix of size \(N \times N\), the memory space needed (in bytes) is \(N \times N \times 8\). If the block size is \(M\), the number of needed buffers, \(B\), is \(\log_2 N + M/2 + 3\) and the size of the subarrays, \(S\), is \(N/M \times N/M\). The memory space needed for our tiled subarray method is \(B \times N + S\). This method works for any block size that is a power of 2 and the matrix size is divisible by the block size. The code is not given due to space limitations.

We have measured the performance of the tiled stepped calculation for different block sizes. The performance values are listed in Tables 4.2 and 4.3.

<table>
<thead>
<tr>
<th>Block Size</th>
<th>512x3072</th>
<th>256 x 3072</th>
<th>128 x 3072</th>
<th>64 x 3072</th>
<th>32 x 3072</th>
<th>16 x 3072</th>
<th>8 x 3072</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subarray Size</td>
<td>6 x 6</td>
<td>12 x 12</td>
<td>24 x 24</td>
<td>48 x 48</td>
<td>96 x 96</td>
<td>192 x 192</td>
<td>384 x 384</td>
</tr>
<tr>
<td>Elapsed</td>
<td>03:37.5</td>
<td>03:36.0</td>
<td>03:33.5</td>
<td>03:36.4</td>
<td>03:39.6</td>
<td>03:42.5</td>
<td>03:57.1</td>
</tr>
<tr>
<td># of Seek</td>
<td>30,704</td>
<td>30,704</td>
<td>30,704</td>
<td>30,704</td>
<td>30,688</td>
<td>30,660</td>
<td>30,616</td>
</tr>
<tr>
<td># of Read</td>
<td>15,352</td>
<td>15,352</td>
<td>15,352</td>
<td>15,352</td>
<td>15,344</td>
<td>15,330</td>
<td>15,308</td>
</tr>
<tr>
<td># of Write</td>
<td>15,352</td>
<td>15,352</td>
<td>15,352</td>
<td>15,352</td>
<td>15,344</td>
<td>15,220</td>
<td>15,308</td>
</tr>
<tr>
<td>Memory usage</td>
<td>456,927</td>
<td>584,307</td>
<td>872,319</td>
<td>1,507,515</td>
<td>2,940,759</td>
<td>6,384,819</td>
<td>15,509,391</td>
</tr>
</tbody>
</table>

Table 4.2  Performance of Tiled Stepped Calculations with Subarrays  for Matrix Size 3072 x 3072
Block Size  |  512 x 6144  |  256 x 6144  |  128 x 6144  |  64 x 6144  |  32 x 6144  |  16 x 6144  |  8 x 6144  
---|---|---|---|---|---|---|---
Subarray Size | 12 x 12  | 24 x 24  | 48 x 48  | 96 x 96  | 192 x 192  | 384 x 384  | 768 x 768  
Elapsed     | 14:33.1  | 14:53.6  | 15:01.3  | 15:18.3  | 15:47.5  | 16:18.8  | 17:12.60  
# of Seek   | 61,424  | 61,424  | 61,424  | 61,418  | 61,402  | 61,332  | 61,204  
# of Read   | 30,712  | 30,712  | 30,712  | 30,709  | 30,701  | 30,666  | 30,602  
# of Write  | 30,712  | 30,712  | 30,712  | 30,709  | 30,701  | 30,666  | 30,602  
Memory usage | 1,211,031  | 1,769,379  | 2,969,823  | 5,558,139  | 11,336,919  | 25,155,507  | 61,689,231  

From the experiments we see that tiling and subarray stepped calculations use only 21.8% to 49.7% of the execution time of the regular stepped calculation. The experiments on different block and subarray sizes suggest that the block and subarray sizes do not play an important role in the performance.

4.2.3 Tiled Stepped Calculations without Subarrays

To implement tiled stepped calculation, we map two rows of the matrix A into memory. After one round of calculation, we release the row which is no longer needed in future calculations. We start over for another row until the whole stepped calculation is finished. Assuming the size of matrix is $N \times N$, the number of the buffers, B, is $\log_2 N + 3$. The memory space needed is $B \times N$. This schema works for any even matrix size. Again, the code is not given here.

We have measured the performance of stepped calculation for different data sets of size 72MB and 288MB. The performance values are listed in Table 4.4. We observe that tiled stepped calculations use only 21.7% to 46.5% of the execution time of the regular stepped calculation (Table 4.1). We also see that tiled stepped calculations without subarray use 85.8% to 100% of the execution time of the tiles stepped calculation with subarrays (Tables 4.2 and 4.3).

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Elapsed</th>
<th># of Seek</th>
<th># of Read</th>
<th># of Write</th>
<th>Memory usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>3072 x 3072</td>
<td>03:41.8</td>
<td>30,712</td>
<td>15,356</td>
<td>15,356</td>
<td>356,919</td>
</tr>
<tr>
<td>6144 x 6144</td>
<td>14:32.3</td>
<td>61,432</td>
<td>30,712</td>
<td>30,716</td>
<td>762,459</td>
</tr>
</tbody>
</table>

Table 4.4 Performance of Tiled Stepped Calculations without Subarrays

5 CONCLUSION

Building on previous work on Comanche [14], we have presented an approach to the problem of improving the calculation of out-of-core problems. Our extension of Comanche involves the use of block mapping to minimize I/O. From the experiments we see that block multiplication uses only 25% to 50% of execution time of regular multiplication (VMM). Our results indicate that the blocks of size 128 x 128 and 32 x 32, using elapsed time as primary measure, perform better than the others. We also studied irregular access patterns. Stepped program optimization has been implemented using techniques which are a combination of tiling, subarrays, loop fusing, and the seeker/reaper paradigm. From the experiments we see that tiling and subarray stepped calculations use only 21.8% to 49.7% of the execution time of the regular stepped calculation. The experiments on different block and subarray size suggest that the block and subarray sizes do not play an important role in the performance.

REFERENCES


