Generating Efficient Programs for Two-Level Memories from Tensor-Products[†]

Sandeep K. S. Gupta, Zhiyong Li, John H. Reif
Department of Computer Science,
Duke University, Durham, NC 27708

Abstract

This paper presents a framework for synthesizing efficient out-of-core programs for block recursive algorithms such as the fast Fourier transform (FFT) and Batcher's bitonic sort. The block recursive algorithms considered in this paper are described using tensor (Kronecker) product and other matrix operations. The algebraic properties of the matrix representation are used to derive efficient out-of-core programs. These programs are targeted towards a two-level disk model which allows HPF supported *cyclic*(B) data distribution on a disk array. The effectiveness of our approach is demonstrated through an example out-of-core FFT program implemented on a work-station.

Keywords: Parallel I/O, parallelizing compilers, tensor product, FFT computation.

1. Introduction

Massively parallel computers intend to provide solutions for the large-scale scientific applications such as the Grand Challenge Problems like computational fluid dynamics and seismic signal processing. These applications require intensive I/O operations and hence their performance is mainly dependent on the cost of disk accesses. However, the performance of the I/O subsystem in current massively parallel machines has not been largely improved considering the significant progress that has been made for the CPU and the communication networks over the last two decades [3].

Recently, many research projects [2] have studied the requirements of I/O intensive applications. Their efforts range from languages, compilers, runtime systems, file systems to performance models. In this paper, we present a framework of using tensor products to generate (or synthesize) disk-efficient I/O programs for a class of block recursive problems. The tensor product framework has been used to synthesize efficient programs for various architectures such as distributed-memory machines [5] and sequential I/O systems [8].

The programs synthesized from our approach are targeted towards a two-level disk model, which is a variation of Aggarwal and Vitter's two-level memory model[1]. The data distribution on the disks are described using the High Performance Fortran (HPF) data layout annotations. Algorithms considered in this paper are described by tensor products formulas and their efficient implementations on our two-level model are obtained by using the algebraic properties of the tensor product formulas. We consider two types of tensor products formulas : stride permutations (e.g. matrix transposition) and general tensor product formulas. We present a unified framework for deriving the known matrix transposition algorithms such as Eklundh's, Stone's, and Modified Stone's matrix transposition algorithms. Many other matrix transposition algorithms can also be suitably derived from this approach. The efficiency of these algorithms will be depended on the underlined computation model. We present an efficient algorithm for our two-level model. We further present efficient program generation strategies for a tensor product and for the tensor product formulas. We use the Cooley-Tukey FFT algorithm as an example to illustrate the methodology.

The paper is organized as follows. Section 2 introduces a two-level computation model. In Section 3, we define several basic concepts and summarize the strategies for program generation. Section 4 presents program generation strategies for stride permutations and Section 5 discusses program generation strategies for tensor products. In Section 6, we present program generation strategies for tensor product formulas. Conclusions are provided in Section 7.

2. Parallel I/O Computation Model

The two-level model used in this paper is similar to Aggarwal and Vitter's two-level memory model [10]. A computer system consists of a processor with an internal random access memory and a set of disks. The memory in each disk is infinite and is organized as a set of blocks. Four parameters, N(the size of the input); M(the size of the internal memory); B(the size of each block); and D(the number of disks), are used to define the model. For simplicity, we assume all parameters to be power of two. Further, we assume that $M < N$, $1 \leq B \leq \frac{M}{2}$ and $1 \leq D \leq \frac{M}{B}$.

The data is distributed on these disks using a block-cyclic distribution, such as for an one-dimensional array $A(0 : N-1)$ of size N on an array of D disks:

Definition 2.1 *A(cyclic(B)): the block size is B and element A(k) at global index k is on disk with disk index d = (k div B) mod D, at block index b = (k div DB), and at local index l = (k mod B) + (k div (DB))B.*

Each parallel I/O operation can access D blocks from different disks simultaneously. In this paper, we use the *striped disk* access model in which blocks in one I/O operation come from the same track as opposed to the *independent I/O* model in which block can come from different tracks, where a *track* consists of blocks which have the same block index number on all the disks under *cyclic*(B) distribution. We define the measure of I/O performance as the number of I/Os required. We also assume that both the input and output data are stored in a *cyclic*(B) manner on the disk array.

3. Basics for Program Generation

Assume that $A^{m,n}$ is an $m \times n$ matrix and $B^{p,q}$ is a $p \times q$ matrix. The *tensor product* $A^{m,n} \otimes B^{p,q}$ is the matrix obtained by replacing element $a_{i,j}$ by the matrix $[a_{i,j}B]$. A *vector basis* e_i^m, $0 \leq i < m$, is a column vector of length m with an one

[†]This work was supported under ARPA/SISTO contracts N00014-91-J-1985, N00014-92-C-0182 under subcontract KI-92-01-0182, Rome Labs Contract F30602-94-C-0037, and NSF Grant NSF-IRI-91-00681.

[‡]Authors' e-mail address: {zli,sandeep,reif}@cs.duke.edu

```
// X is an array of size M
DO i_32 = 0,1
  DO i_31 = 0,3
   // Parallel read from a track
   X(4i_31 : 4i_31 - 3) <- parallel_read(4i_32 + i_31)
  ENDDO
  // Perform operation for a memory load
  Code(X(1 : 16) <- (A).(X(1 : 16))
  // Write the result back
  DO i_31 = 0,3
   // Parallel write to a track
   parallel_write(4i_32 + i_31) <- X(4i_31 : (4i_31 - 3))
ENDDO  ENDDO
```

Figure 1: Code for the tensor product $I_4 \otimes F_2 \otimes I_4$.

at position i and zeros elsewhere. The tensor product of vector bases is called a *tensor basis*. A tensor basis $e_{i_1}^{m_1} \otimes \cdots \otimes e_{i_t}^{m_t}$ can be linearized into a vector basis $e_{i_1 m_2 \cdots m_t + \cdots + i_{t-1} m_t + i_t}^{m_1 \cdots m_t}$. Equivalently, a vector basis e_i^M can be factorized into a tensor product of vector bases $e_{i_1}^{m_1} \otimes \cdots \otimes e_{i_t}^{m_t}$, where $M = m_1 \cdots m_t$ and $i_k = (i \ div \ M_{k+1}) \ mod \ m_k, M_k = \prod_{i=k}^{t} m_i, \ M_{t+1} = 1$. For example, $e_i^{12} = e_{i_3}^2 \otimes e_{i_2}^3 \otimes e_{i_1}^2$.

The stride permutation L_n^{mn} is defined by tensor bases as, $L_n^{mn} \left(e_i^m \otimes e_j^n \right) = e_j^n \otimes e_i^m$, which also specifies the relationship between the *input* and *output tensor bases*. This gives the relationship between the indexing of the input and the output vectors. By linearizing the *input tensor basis* $e_i^m \otimes e_j^n$ to e_{in+j}^{mn}, we get the indexing function of the input vector to be $in + j$. Similarly, the indexing function of the output vector is obtained from linearizing the *output tensor basis* $e_j^n \otimes e_i^m$ to be $jm + i$.

Block recursive algorithms considered in this paper can be represented by (a multiplication of) the tensor product, $I_r \otimes A_p \otimes I_s$, where A_p is a $p \times p$ square matrix. When A_p is a permutation, the formula denotes a *tensor permutation*. When A_p is a general matrix, it represents a *tensor product*. The multiplication of the compatible tensor products is called a *tensor product formula*. The FFT computation is a notable example which can be represented by a tensor product formula. For example, the core computation in the Cooley-Tukey FFT algorithm can be expressed as follows,

$$X = \prod_{i=1}^{n} (I_{2^{n-i}} \otimes F_2 \otimes I_{2^{i-1}}) (X). \tag{1}$$

Definition 3.2 *For a vector of size N, its memory basis is defined as $\beta_m = e_i^{\frac{N}{BD}} \otimes e_d^D \otimes e_b^B$.*

The input and output data may be distributed in a different manner, which will be denoted by *input* and *output memory bases* respectively. The track number now can be obtained based on the memory basis as $T_i = i$.

Definition 3.3 *A loop basis is a permutation of the (factorized) memory basis.*

It is possible there are two loop structures in a synthesized program, one corresponding to the *input loop basis* and another corresponding to the *output loop basis*.

Loop bases and memory bases together capture the data access patterns to disks. The order of part of the indices in the loop basis specifies an ordering of the index variables in the generated loop structure. Consider the following example. Assume that $M = 16$, $D = 2$, $B_d = 2$, F_2 a 2×2 matrix, and data are distributed in $cyclic(2)$ manner, for example, then for the tensor product $I_4 \otimes F_2 \otimes I_4$, the input memory basis is $e_{i_3}^8 \otimes e_{i_2}^2 \otimes e_{i_1}^2$ and the output memory basis is the same as the input memory basis. Let the input and output loop bases are the same as $e_{i_{32}}^2 \otimes e_{i_{31}}^4 \otimes e_{i_2}^2 \otimes e_{i_1}^2$, which is an identity permutation of the factorized memory basis, then the code in Fig. 1 can be used for computing $I_4 \otimes F_2 \otimes I_4$.

For the program shown in Fig. 1, the data is read beginning from the first track until the main memory is full. Then computation for a memory load is performed. The order of the loop variables in the nested-loop structure is consistent with the order of the index variables in the loop basis.

The principal work of program generation is to determine the loop bases and the corresponding operations to the subset of data (called a *memory-load* M_L). When the loop bases and the operation to the subset of data have been determined, the following procedure can be used for synthesizing efficient I/O programs.

- The input loop basis can be written as, $\beta_{li} = (\otimes_{j=h_1+1}^{h_2} e_{i_j}^{k_i})(\otimes_{j=1}^{h_1} e_{i_j}^{l_j})$, where h_1 is the smallest integer such that $\otimes_{j=1}^{h_1} e_{k_j}^{l_j} = M_L$.
- Loop over indices which are not in j, $1 \leq j \leq h_1$.
  - read blocks to fill one memory load, and the track number is formed from the input memory basis;
  - apply the corresponding operations to M_L;
  - store the results using the index order corresponding to the output loop basis, and the track number is formed from the output memory basis.

If all of the factors of $e_{i_2}^D$ and $e_{i_1}^B$ are in the range of $e_{i_j}^{l_j}$, where $1 \leq j \leq h_1$, then a full *parallel_read* and *parallel_write* operations can be generated. Otherwise, we may not be able to use the full disk stripping capability.

4. Program Generation for Stride Permutations

The input tensor basis for the stride permutation is, $\beta_i = e_{i_2}^p \otimes e_{i_1}^q$. The principal approach to derive different algorithms for a stride permutation consists of two steps. The first step factorizes the vector bases $e_{i_2}^p$ and $e_{i_1}^q$. The second step regards the factorized tensor bases as a sequence, called *a vector basis sequence*, denoted as

$$\zeta = (e_{i_r}^{p_r}, ..., e_{i_1}^{p_1}, e_{j_s}^{q_s}, ..., e_{j_1}^{q_1}), \tag{2}$$

where $p_r \times ... \times p_1 = p$, $q_s \times ... \times q_1 = q$, and the ith element of the sequence is denoted as $\zeta(i)$. Then a set of permutations is applied to this sequence until the desired tensor basis $e_{i_1}^q \otimes e_{i_2}^p$ is derived. The permutations applied to the sequence could be simple, such as interchange, rotation, and their combination. We derive the block, Stone's [9], modified Stone's and Eklundh's [4] matrix transposition algorithms as follows.

Block matrix transposition algorithm Take $r = s = 2$ for ζ defined in the Formula (2). If the input (memory) basis is denoted as $e_{i_2}^{p_2} \otimes e_{i_1}^{p_1} \otimes e_{j_2}^{q_2} \otimes e_{j_1}^{q_1}$. Then the block matrix transposition algorithm can be described by the following steps,

1. Exchange $\zeta(2)$ and $\zeta(3)$, resulting in $e_{i_2}^{p_2} \otimes e_{j_2}^{q_2} \otimes e_{i_1}^{p_1} \otimes e_{j_1}^{q_1}$;
2. Exchange $\zeta(3)$ and $\zeta(4)$, resulting in $e_{i_2}^{p_2} \otimes e_{j_2}^{q_2} \otimes e_{j_1}^{q_1} \otimes e_{i_1}^{p_1}$;
3. Exchange $\zeta(1)$ and $\zeta(2)$, resulting in $e_{j_2}^{q_2} \otimes e_{i_2}^{p_2} \otimes e_{j_1}^{q_1} \otimes e_{i_1}^{p_1}$;
4. Exchange $\zeta(2)$ and $\zeta(3)$, resulting in $e_{j_2}^{q_2} \otimes e_{j_1}^{q_1} \otimes e_{i_2}^{p_2} \otimes e_{i_1}^{p_1}$.

Step 1 rearranges the data. Step 2 transposes the submatrix of $p_1 \times q_1$. Step 3 transposes the matrix by regarding the submatrix as a single element. The last step arranges the result. A tensor product representation for this algorithm is shown below [7]:

$$\begin{aligned} L_{q_2 q_1}^{p_2 p_1 q_2 q_1} &= (I_{q_2} \otimes L_{q_1}^{p_2 q_1} \otimes I_{p_1})(L_{q_2}^{p_2 q_2} \otimes I_{p_1 q_1}) \\ &\quad (I_{p_2 q_2} \otimes L_{q_1}^{p_1 q_1})(I_{p_2} \otimes L_{q_2}^{p_1 q_2} \otimes I_{q_1}). \end{aligned}$$

Stone's algorithm Factorize q however remain p unfactorized. The algorithm simply corresponds to right rotating the vector basis sequence ζ s times. A tensor product representation for this algorithm is: $L^{pq}_{q_s\cdots q_1} = \prod_{i=1}^{s} L^{pq}_{q_i}$.

Modified Stone's algorithm Factorize q however remain p unfactorized. The algorithm consists of s stages. Each stage i consists of two steps : step 1 corresponds to the right rotating $e^{q_k}_{i_k}, ..., e^{q_1}_{i_1}$; step 2 corresponds to the right rotating the remained elements in the vector basis sequence of ζ. A tensor product representation for this algorithm is: $L^{pq_s\cdots q_1}_{q_s\cdots q_1} = \prod_{i=1}^{s}\left(L^{pq_i}_{q_i}\otimes I_{q_s\cdots q_{i+1}q_{i-1}\cdots q_1})(I_p\otimes L^{q}_{q_i}\right)$.

Eklundh's algorithm Take $r = s$ for ζ. Eklundh's algorithm consists of r stages. Each stage i corresponds to the exchange of $\zeta(i)$ with $\zeta(i+r)$, where $1 \le i \le r$. For $p = q = 2^n$, the Eklundh's algorithm can be represented by the following tensor formula [7]: $L^{2^{n+1}}_{2^n} = \prod_{i=1}^{n-1}\left(I_{2^{n-i-1}}\otimes(I_2\otimes L^{2^n}_{2^{n-1}})L^{2^{n+1}}_{2}\otimes I_{2^i}\right)$.

Many of other possible algorithms can be obtained by the above approach. In the next subsection, we describe an algorithm which reaches the efficient performance in our two-level model.

4.1. Efficient implementation of stride permutations for the two-level model

A program is called *disk-optimal* if it requires $\frac{2N}{DB}$ I/O operations. The terminology of *an efficient program* is also used in this paper. An efficient program may have a constant factor k, where $k \ge 3$, in the I/O operations more than those in the optimal implementation. The following lemma proves that a disk-optimal implementation can be obtained for stride permutations under the condition $M \ge (DB)^2$.

Lemma 4.1 *For the stride permutation L^{pq}_{q}, if both p and $q \ge DB$, then an optimal implementation can be obtained by using the block algorithm.*

Proof: If both p and $q \ge DB$, then there is a factorization $p = p_2p_1$ and $q = q_2q_1$ such that p_1, $q_1 \ge DB$ and $p_1 \times q_1 \le M$. The block matrix transposition algorithm presented in the last subsection consists of four steps. It seems that four passes are required to access the data set. However, by reading and writing in a strided fashion as discussed in [7] , we can combine four passes into one. The idea is to use the order specified by the tensor basis in Step 1 as the input loop basis and Step 3 as the output loop basis. In order to read and write in parallel, we factorize $e^{q_1}_{j_1}$ and $e^{p_1}_{i_1}$ as $e^{\frac{q_1}{DB}}_{j_{13}}\otimes e^{D}_{j_{12}}\otimes e^{B}_{j_{11}}$ and $e^{\frac{p_1}{DB}}_{i_{13}}\otimes e^{D}_{i_{12}}\otimes e^{B}_{i_{11}}$, respectively. Since $p_1 \times q_1 \le M$, we can load the whole submatrix into the main memory. The loaded submatrix is transposed in the main memory. Finally, this submatrix is written out corresponding to the output loop basis. We refer for details of the synthesized program to [6]. □

We present the following efficient two pass algorithm for the matrices which do not satisfy the conditions in Lemma 4.1. If the conditions in Lemma 4.1 are not satisfied, then either p or q will be less than DB. Since we are interested in large data sets, either p or q will be much larger than DB (such as $p > k(DB)^2$). Without loss of generality, we assume that p is much larger than DB. We factorize p and separate the factorization result into two parts $p_r, ..., p_{i+1}$ and $p_i, ..., p_1$ such that both t_2 $(= p_r...p_{i+1})$ and t_1 $(p_i...p_1)$ are greater than DB. The first step of the algorithm does a matrix transposition corresponding to $L^{pq}_{qt_1}$. The second step does a matrix transposition corresponding to $L^{pq}_{qt_2}$. Since each of the two steps can be implemented optimally, the algorithm needs $\frac{4N}{DB}$ I/O operations. In other words, our algorithm uses the following property of the stride permutation:

Theorem 4.1 $L^{rst}_{t} = L^{rst}_{rt}\ L^{rst}_{st}$.

```
// X,Y:  arrays of size M
DO i32 = 0, N/M - 1
  DO i31 = 0, M/(DB) - 1
    X(i31 DB : (i31 + 1)DB - 1) <- parallel_read(i32 M/(DB) + i31)
  ENDDO
  // Operation for a memory load
  Code(Y(1 : M) <- (A).(X(1 : M))
  // Write the result back
  DO i31 = 0, M/(DB) - 1
    parallel_write(i32 M/(DB) + i31) <- X(i31 DB : (i31 + 1)DB - 1)
ENDDO  ENDDO
```

Figure 2: Code for a tensor product with $ps \le M$.

5. Program Generation for Tensor Products

An efficient implementation of a tensor product is dependent upon the underlying model, data distributions, data access patterns (or loop structure), and the values of r, q and s in the tensor product.

Theorem 5.2 *For the tensor product $(I_r\otimes A_p\otimes I_s)$ with A_p being an operator,*

1. *if $ps \le M$, then an optimal implementation exists;*
2. *if $ps > M$ then,*
   a. *if $M \ge pDB$, a disk-optimal program can be generated.*
   b. *if $M < pDB$ and $M \ge pB$, a program can be generated with at least a factor $\frac{D}{D'}$ of optimal, where $D' = \frac{M}{pB}$.*
   c. *if $M < pB$, a program can be generated with at least a factor $\frac{B}{B'}$ of optimal, where $B' = \frac{MB}{ps}$.*

Proof : Because of the space limitation, we just give the proof for 1 and 2.b. We refer for further details to [6].

1. $ps \le M$: In this case, the loop basis can be obtained by factorizing the memory basis as follows: $\beta_m = e^{\frac{N}{M}}_{i_{32}}\otimes e^{\frac{M}{DB}}_{i_{31}}\otimes e^{D}_{i_2}\otimes e^{B}_{i_1}$. By using the procedure listed in section 2.1, the code can be synthesized as shown in Fig. 2, which uses the same order as the indices appeared in the memory basis (In order to keep the code simple, in Fig. 2, we assume that both X and Y has size M, or the size of the main memory is $2M$. The same assumptions are made henceforth.). From Figure 2, we can find out that the number of I/Os needed is $\frac{N}{DB}$, therefore the code is disk-optimal.
2. $ps > M$ and $M \ge pDB$: Factorize the memory basis as: $\beta_m = e^{r}_{i_{34}}\otimes e^{p}_{i_{33}}\otimes e^{\frac{N}{rM}}_{i_{32}}\otimes e^{\frac{M}{pDB}}_{i_{31}}\otimes e^{D}_{i_2}\otimes e^{B}_{i_1}$, where $e^{k}_{i_{3j}}$, $j \in \{1,2,3,4\}$, is a factor of $e^{k}_{i_3}$. By permuting β_m, the following loop basis can be derived: $\beta_l = e^{r}_{i_{34}}\otimes e^{\frac{N}{rM}}_{i_{32}}\otimes e^{p}_{i_{33}}\otimes e^{\frac{M}{pDB}}_{i_{31}}\otimes e^{D}_{i_2}\otimes e^{B}_{i_1}$. The above memory and loop basis can be explained as follows. The input vector X is separated into r equal segments at first. Each segment is then separated into p equal subsegments. Each of these subsegments is further separated into $\frac{N}{rM}$ subsubsegments. We now read in first $\frac{M}{p}$ elements from each of the subsubsegments, which consists of a memory load. The operation to this subset of data can be determined by examining the loop basis. The vector basis $e^{p}_{i_{33}}$ in the loop basis corresponds to the operator of the original tensor product. The data set following $e^{p}_{i_{33}}$ represents a segment of one memory load, which has $\frac{M}{p}$ records. Therefore, the operation to the subset is $(A_p\otimes I_{\frac{M}{p}})$. By reading in successive $\frac{M}{p}$ records each time, we can finally finish the whole computation. The detailed

```
// X,Y: arrays of size M
DO i34 = 0, r-1
  DO i32 = 0, N/(rM) - 1
   DO i33 = 0, p-1
    DO i31 = 0, M/(pDB) - 1
      X((i33 M/(pDB) + i31)DB : (i33 M/(pDB) + i31 + 1)DB - 1)
        <- parallel_read(i34 N/(rDB) + i33 N/(rpDB) + i32 M/(pDB) + i31)
   ENDDO  ENDDO
   // Operation for a memory load
   Code(Y(1 : M) <- (A⊗I_{M/p}).(X(1 : M))
   // Write the result back
   DO i33 = 0, p-1
    DO i31 = 0, M/(pDB) - 1
      parallel_write(i34 N/(rDB) + i33 N/(rpDB) + i32 M/(pDB) + i31)
       <- X((i33 M/(pDB) + i31)DB : (i33 M/(pDB) + i31 + 1)DB - 1)
ENDDO  ENDDO  ENDDO  ENDDO
```

Figure 3: Code for a tensor product with $ps \geq M \geq pDB$.

code is shown in Fig. 3. The number of I/Os required in the code is $\frac{2N}{DB}$, therefore it is disk-optimal. □

6. Program Generation for Tensor Product Formulas

We now discuss program generation strategies for the following tensor product formula,

$$Y^N = \prod_{i=1}^{n} \left(I_{p^{n-i}} \otimes A_p \otimes I_{p^{i-1}} \right) (X^N). \tag{3}$$

where $N = p^n$ and we assume that $p < M$. There are several strategies for developing disk-efficient programs, such as exploiting locality and exploiting parallelism of data input and output. Similar ideas have been discussed in [8], where they use *factor grouping* to exploit the locality and *data rearrangement* to reduce the cost of I/O operations.

Factor grouping combines contiguous factors in a tensor product formula together and therefore may reduce the number of passes to access the secondary storage. For example, if we group the successive tensor products in a tensor product formula under the conditions in either case 1 or case 2 of Theorem 5.2, we can always reduce the number of I/O operations. Data rearrangement uses the property of the stride permutation to change the data access pattern.

We now present a simple greedy algorithm which is based on using factor grouping to improve the I/O performance. According to Theorem 5.2, we can always derive an optimal implementation for each tensor product under the following condition: $ps \leq M$ or ($ps > M$ and $M \geq pDB$). Since grouping contiguous tensor products will decrease the number of passes to access the secondary storage, we would like to group as many contiguous factors in the tensor product formula as possible. If the number of factors we can group is k, then from the properties of tensor products, the resulting formula can be written as, $I_{r_1} \otimes (\otimes_{i=1}^{k} A_p) \otimes I_{s_1}$. Therefore the constraint for the value of k is $p^k s_1 \leq M$ or ($p^k s_1 > M$ and $p^k DB \leq M$). Since the size of the basic operator A_p is the same and s's in the later terms of the formula are larger than those in the earlier terms, we will begin the grouping from the last factor and group from there greedily. It turns out that this simple approach is quite effective as shown by the following example. Assuming that $N = 2^{40}$, $D = 2^4$, $B = 2^9$ and $M = 2^{22}$, then by using the above greedy algorithm, $F_{2^{40}}$ can be transformed as,

$$\begin{aligned} F_{2^{40}} &= \prod_{i=1}^{40} (I_{2^{40-i}} \otimes F_2 \otimes I_{2^{i-1}}) \\ &= ((\otimes_{i=1}^{9} F_2) \otimes I_{2^{31}})(I_{2^9} \otimes (\otimes_{i=1}^{9} F_2) \otimes I_{2^{22}}) \\ &\quad (I_{2^{18}} \otimes (\otimes_{i=1}^{22} F_2)) \end{aligned}$$

A program for the resulting formula can be synthesized by using the algorithms presented in the proof of Theorem 5.2. The transformed tensor product formula requires $\frac{6N}{DB}$ I/O operations. However the original tensor product formula needs at least $\frac{80N}{DB}$ I/O operations.

6.1. Performance results

We compare programs synthesized using greedy algorithms described above with programs using virtual memory systems on a Sun SPARCstation. The virtual memory implementation can be regarded as a direct implementation of the original tensor product formula. The paging system generates appropriate I/O calls. The performance results are shown in Table 1.

	$F_{2^{20}}$	$F_{2^{21}}$	$F_{2^{22}}$	$F_{2^{23}}$	$F_{2^{24}}$
V.M.	7	15	34	184	386
G.M.	7	16	40	55	117

Table 1: Performance comparison on a Sun SPARCstation using the virtual memory (V.M.) and the greedy method (G.M.) (unit: second).

We can see that when the input size is relatively small ($\leq 2^{22}$), the performance of the programs using the greedy method is worse than the performance of the program using virtual memory systems. The reason is that for small data size, the overhead of loop structures and file accesses in the synthesized programs outweighs the paging I/O overhead. However, for larger input data size, the overhead of I/O operations becomes a dominating factor, and programs relying solely on the virtual memory system run significantly slower than the synthesized programs.

7. Conclusion

In this paper, we introduced a framework for synthesizing efficient out-of-core programs from tensor product formulas. The distinguish features of this framework are: (1) it is based on the analysis of tensor bases; (2) the analyses are constructive, i.e., various I/O efficient programs can be generated. We are extending this work to a model with multiple disks and multiple processors.

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Comm. of the ACM*, 31(9):1116–1127, Sept. 1988.

[2] A. Choudhary, I. Foster, G. Fox, K. Kennedy, C. Kesselman, C. Koelbel, J. Saltz, and M. Snir. Languages, compilers, and runtime systems support for parallel input-output, 1994. Scalable I/O Initiative Working Paper Number 3.

[3] J. M. del Rosario and A. Choudhary. High performance I/O for parallel computers: Problems and prospects. *IEEE Computer*, 27(3):59–68, March 1994.

[4] J. O. Eklundh. A fast computer method for matrix transposing. *IEEE Trans. on Computers*, 20(7):801–803, 1972.

[5] S. K. S. Gupta. *Synthesizing Communication-efficient Distributed-Memory Parallel Programs for Block Recursive Algorithms*. PhD thesis, The Ohio State Univ., March 1995.

[6] S. K. S. Gupta, Z. Li, and J. H. Reif. Synthesizing efficient parallel I/O programs for block recursive algorithms from tensor-products. Technical report, Dept. of Computer Sci., Duke Univ., 1995.

[7] S. D. Kaushik, C.-H. Huang, J. R. Johnson, R. W. Johnson, and P. Sadayappan. Efficient transposition algorithms for large matrices. In *Supercomputing '93*, Nov. 1993.

[8] S. D. Kaushik, C.-H. Huang, R. W. Johnson, and P. Sadayappan. A methodology for generating efficient disk-based algorithms from tensor product formulas. In *Proc. Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, pages 358–338, Aug. 1993.

[9] H. S. Stone. Parallel processing with the perfect shuffle. *IEEE Trans. on Computers*, 20(2):153–161, 1971.

[10] J. S. Vitter and E. A. M. Shriver. Optimal disk I/O with parallel block transfer. In *Proceedings of 22nd Annual ACM Symposium on Theory of Computing*, pages 159–169, 1990.