EXTENT: A Portable Programming Environment for Designing and Implementing High-Performance Block Recursive Algorithms*

D. L. Dai, S. K. S. Gupta, S. D. Kaushik, J. H. Lu, R. V. Singh, C.-H. Huang, P. Sadayappan
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210†
and
R. W. Johnson
Department of Computer Science
St. Cloud State University
St. Cloud, MN 56301‡

Abstract

EXTENT is an EXpert system for TENsor product formula Translation. In this paper we present a programming environment for automatic generation of parallel/vector programs from tensor product formulas. A tensor (Kronecker) product based programming methodology is used for designing high performance programs on various architectures. In this programming methodology, block recursive algorithms such as the fast Fourier transform and Strassen's matrix multiplication algorithm are expressed as tensor product formulas involving tensor product and other matrix operations. A tensor product formula can be systematically translated to parallel and/or vector code for various parallel architectures. A prototype system which generates programs for the Cray Y-MP, Cray T3D, and Intel Paragon has been developed. Performance results for some generated programs are presented.

Keywords: *Parallel programming environment, Tensor (Kronecker) product, Block recursive algorithm, Parallel program synthesis.*

1 Introduction

The future of scalable supercomputers crucially depends on the availability of sophisticated tools which can expedite the development of high performance programs for these machines. Although bigger and faster machines have been built over the past decade, the state-of-the-art in programming tools has noticeably lagged behind. This situation has been exacerbated by the lack of programming languages which are compilable into efficient codes for various supercomputer architectures. Existing software support for developing parallel programs broadly fall into two categories. The first category includes general purpose programming languages like pC++ [2], HPF [7], Fortran D [13], and Vienna Fortran [4] and program transformation/restructuring tools like Parafrase-2 [21], ParaScope [1], and SUPERB [8, 25]. The second category includes application oriented problem-solving and program development programming environments such as parallel ELLPACK [15] and libraries such as ScaLAPACK [5].

This paper presents EXTENT (*EX*pert system for *TEN*sor product formula *T*ranslation) which is a portable programming environment for a tensor (Kronecker) product based programming methodology. EXTENT is targeted towards development of high performance programs for block recursive algorithms. The tensor product has been used for modeling algorithms with recursive computational structure which occur in application areas such as digital signal processing [10, 22], image processing [23], linear system design [3], and statistics [11]. In recent years, the tensor product framework has been successfully used to design and implement high performance algorithms for the discrete Fourier transform (DFT) [18, 24] and matrix multiplication [16, 17] on shared-memory vector multiprocessors. The significance of the tensor product lies in its ability to model computational structures occurring in a wide spectrum of supercomputer architectures, as well as, the underlying hardware

*Supported in part by ARPA, order number 7898, monitored by NIST under grant number 60NANB1D1151 and ARPA, order number 7899, monitored by NIST under grant number 60NANB1D1150.

†e-mail {dai, sandeep, kaushik, lu, singh-rv, chh, saday }@cis.ohio-state.edu

‡e-mail: rwj@eeyore.stcloud.msus edu

structures, like the interconnection networks [19, 20]. This, coupled with the availability of a methodology for translating algorithms expressed using the tensor product and other matrix operations into parallel/vector codes for shared-memory vector multiprocessors [17, 18] and distributed-memory MIMD machines [12], provides a basis for the portable programming environment presented in this paper.

To design and implement an algorithm using the tensor product requires first that the algorithm be expressed as the evaluation of a matrix formula. Then the various constituent matrices are factored into tensor products of smaller matrices. The process is repeated until either no matrices factor or the desired primitive matrices are formed. The resulting tensor product formula is then translated into vector/parallel operations, iterative loops, and data movement operations. Other mathematically equivalent formulas can also be derived which, when translated to code, have widely different performance characteristics. Selection of a tensor product formula with the "best" performance characteristics depends, of course, on the architecture of the target machine. The system EXTENT greatly aids the search for the best implementation by automatically generating programs from the tensor product formulas and testing them.

EXTENT is designed so that an application developer can easily experiment with several representations. Users with knowledge of tensor product theory can enter their own tensor product formulas using EXTENT's input tensor product language. Besides providing a syntactic mechanism for specifying tensor product formulas, it also supports vectorization, parallelization, and data distribution directives so that the user can assist the system in generating high performance programs. However, if the user is not conversant with tensor products, EXTENT provides several application subsystems, such as the FFT subsystem. An application subsystem allows users to enter algorithm design specifications. EXTENT generates tensor product formulas according to these specifications. EXTENT translates these formulas into programs, compiles and executes the programs on the target machine, retrieves their raw performance data, and displays a performance summary. Usually, the performance of a program on a target machine can be further enhanced by using optimized basic computation kernels. For example, using optimized kernels for small DFTs can greatly improve the performance of programs to compute a large DFT. EXTENT's input interface supports importing of such kernels into the synthesized program.

The structure of the paper is as follows. In Section 2 we present an overview of EXTENT. Generation of tensor product formulas from their design specification is described in Section 3. Section 4 shows how parallel/vector code can be synthesized for various parallel machines. It also gives performance results for some programs generated by EXTENT. Conclusions are presented in Section 5.

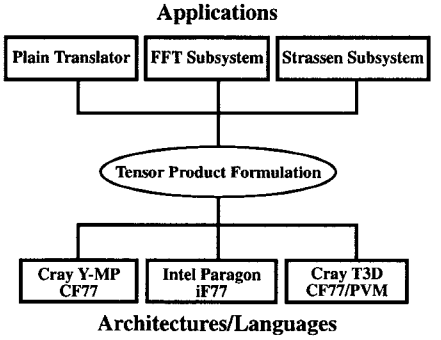

Figure 1: System Block Diagram.

2 System Overview of EXTENT

EXTENT has been developed to run on a Unix-based workstation such as the SUN SPARC workstation. As shown in Fig. 1, EXTENT has two classes of subsystems: application subsystems and architecture subsystems. The current version of EXTENT has the following application subsystems:

- FFT subsystem: for developing fast Fourier transform programs,
- Strassen subsystem: for developing block Strassen matrix multiplication programs.

From each application subsystem, programs can be developed for various machine-language pairs, such as (Cray Y-MP, CF77), (Intel Paragon, iF77), and (Cray T3D, CF77/PVM), using architecture subsystems.

EXTENT provides an X11 window system based user interface which supports the following functionality:

- loading and saving algorithm design specifications,
- generating tensor product formulas from algorithm design specifications,
- translating tensor product formulas to a program in the specified high-level language,
- executing the generated program on the specified target machine,

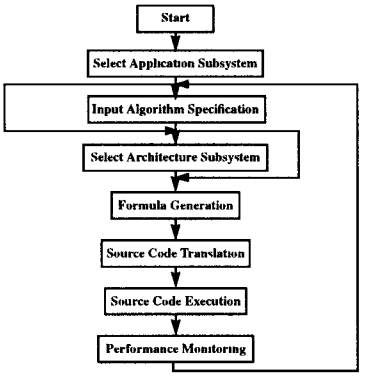

Figure 2: System Flow Diagram.

- displaying the performance summary,
- displaying tensor product formulas.

The system flow diagram in Figure 2 shows the program development steps in the EXTENT programming environment. The user can either generate the tensor product formula using an application subsystem or directly read it from an input file. EXTENT translates the formula to a program for the specified target machine. The generated program is then transferred to the target machine, where it is compiled and executed. Performance data is then fetched and a performance summary is displayed to the user. Using the performance summary, the user can choose to modify the formula or change the specification to generate a program with different performance characteristics.

An application subsystem allows an user to enter simple algorithm specifications to generate tensor product formulas for block recursive algorithms. So far, application subsystems include the FFT subsystem and the Strassen subsystem. A plain translator is provided for directly entering tensor product formulas. The tensor product formula can also be loaded from an input file. We will explain algorithm specification in Section 3.

An architecture subsystem translates the tensor product formulas into target code, executes the target code on the target machine, retrieves and displays performance statistics. An architecture subsystem consists of a translator, executor and performance monitor.

The formula translator generates high-level programs from tensor product formulas. It analyzes tensor product formulas to generate architecture specific code with proper loop structure and array indexing. For a shared-memory vector multiprocessor, parallel loops with array assignment statements are generated. For example, the translator for the Cray Y-MP system generates vectorized code with parallel directives for autotasking. For distributed-memory machines the computation is partitioned according to the owner computes rule and a single program multiple data (SPMD) style node program with explicit message passing primitives is generated. For programs which use data redistributions, the communication code for redistributions is automatically synthesized. The translator for the Intel Paragon generate if77 code with communication commands from Intel's communication library. The translator for the Cray T3D generates CF77 code with either PVM or T3D-specific communication primitives: virtual channels and fast sends and receives. We will present an overview of program synthesis in Section 4.

The executor executes the generated program on the target machine and retrieves the raw performance data. The performance monitor extracts the most important performance statistics from the raw performance data and displays them in the performance window. For example, for the Cray Y-MP, the performance data includes the CPU execution time, wall clock time, CPU percentage of the entire system (CPU%), floating operation per second (MFLOPS), and average vector length (VL). From the performance summary, the user can compare the performance of different tensor product formulas. Using this information the user can then fine tune the generated tensor product formulas to develop more efficient programs.

3 Formula Generation

An application subsystem generates tensor product formulas from a given algorithm design specifications. Before describing the application subsystems, we first give an overview of tensor products.

Let $A^{m,n}$ be an $m \times n$ matrix and $B^{p,q}$ be a $p \times q$ matrix. The *tensor product* $A^{m,n} \otimes B^{p,q}$ is the following block matrix:

$$A^{m,n} \otimes B^{p,q} = \begin{bmatrix} a_{0,0}B^{p,q} & \cdots & a_{0,n-1}B^{p,q} \\ \vdots & \ddots & \vdots \\ a_{m-1,0}B^{p,q} & \cdots & a_{m-1,n-1}B^{p,q} \end{bmatrix}.$$

We will use I_n to represent the $n \times n$ identity matrix. The product $\prod_{i=0}^{n} A_i$ denotes $A_n \cdots A_0$. An application of a computation matrix A to a vector v will be denoted by $A(v)$.

The *stride permutation* L_n^{mn} of a vector X^{mn} is a vector Y^{mn} which is defined as:

$$Y^{mn} = L_n^{mn}(X^{mn}) = \begin{bmatrix} X^{mn}(0 : mn-1 : n) \\ X^{mn}(1 : mn-1 : n) \\ \vdots \\ X^{mn}(m-1 : mn-1 : n) \end{bmatrix},$$

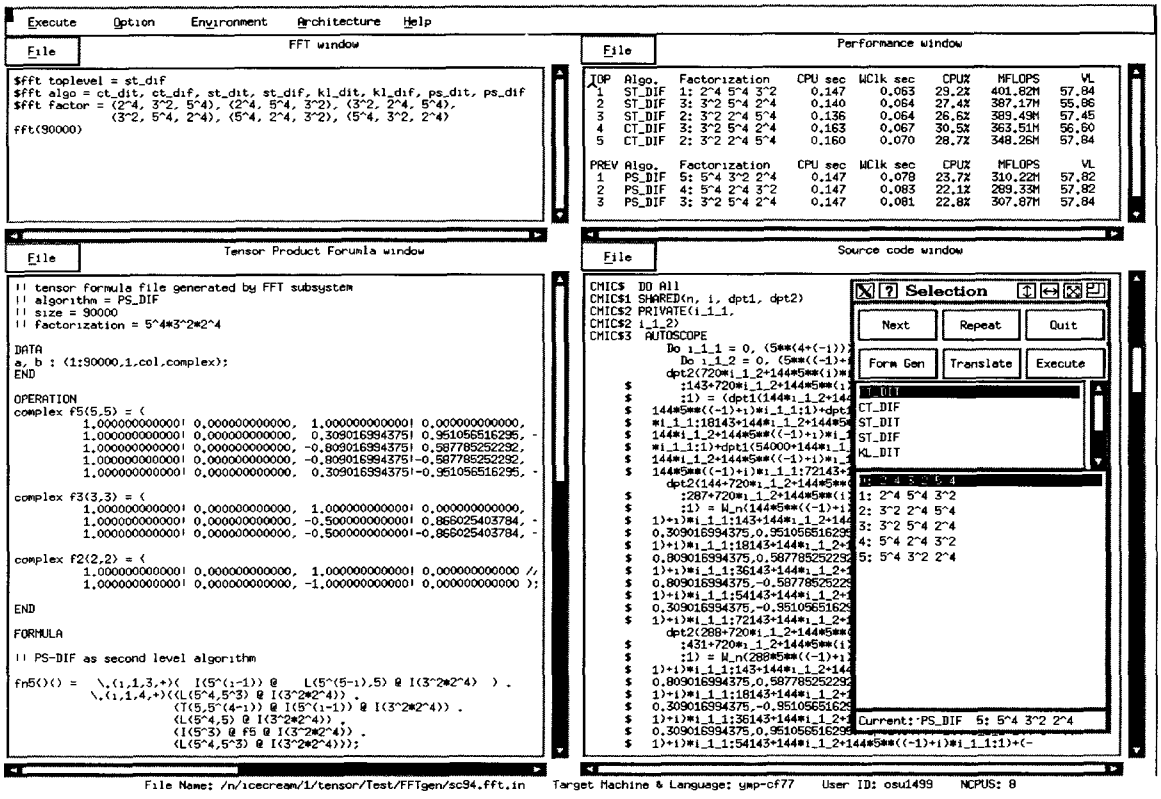

Figure 3: Window screen display for the FFT subsystem.

i.e., the first m elements of Y^{mn} are elements of X^{mn} at stride n starting with element 0, the next m elements are elements of X^{mn} at stride n starting with element 1, and so on.

For details of the theory of tensor products, the reader is referred to [9, 14]. We next describe the FFT subsystem and the Strassen subsystem.

3.1 FFT Subsystem

The FFT subsystem is used for developing high performance programs for computing the DFT. An FFT algorithm for an input of size N is a tensor factorization of an $N \times N$ DFT matrix F_N. Different factorizations of a DFT matrix leads to different FFT algorithms. An FFT algorithm is a divide-and-conquer algorithm which is characterized by the "splitting rule" used to factorize the computation of a larger DFT into smaller DFTs. For example, the splitting rule used for the Cooley-Tukey (CT) decimation-in-time (DIT) FFT is:

$$F_{rc} = (F_r \otimes I_c) T_c^{rc} (I_r \otimes F_c) L_r^{rc},$$

where T_c^{rc} is a diagonal matrix of twiddle factors. The splitting rule is recursively applied till the size of DFTs to be computed belongs to a set of a few small numbers. For example if $N = 2^n$, the following tensor product formula for the radix-2 CT-DIT FFT results by using the above splitting rule:

$$F_{2^n} = \left(\prod_{i=1}^{n} (I_{2^{n-i}} \otimes F_2 \otimes I_{2^{i-1}})(I_{2^{n-i}} \otimes T_{2^{i-1}}^{2^i}) \right) R_{2^n},$$

where $R_{2^n} = \prod_{i=0}^{n-1} (I_{2^i} \otimes L_2^{2^{n-i}})$ and $F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$.

R_{2^n} permutes the input sequence in the bit-reversed order. Tensor product formulation of various other FFT algorithms such as the Stockham (ST) FFT, the Pease (PS) FFT, and the Korn-Lambiotte (KL) FFT can be obtained by using splitting rules which are obtained by an algebraic manipulation of the CT FFT splitting rule. A tensor product formulation for a decimation-in-frequency (DIF) FFT algorithm is obtained by transposing the DIT formula.

An FFT algorithm which uses only a single small DFT matrix in its computation is called a single-radix algorithm. If the problem size N is not a power of a single small number then a mixed-radix FFT algorithm is used. Associated with each mixed-radix

$$
\begin{aligned}
F_{5^4} &= \left(\prod_{i=1}^{3}\left(I_{5^{i-1}} \otimes L_5^{5^{5-i}} \otimes I_{3^2 \cdot 2^4}\right)\right)\left(\prod_{i=1}^{4}\left(L_{5^3}^{5^4}\left(T_{5^{4-i}}^{5\cdot 5^{4-i}} \otimes I_{5^{i-1}}\right) L_5^{5^4}\left(I_{5^3} \otimes F_5\right) L_{5^3}^{5^4}\right)\right), \\
F_{3^2} &= L_3^{3^2}\left(\prod_{i=1}^{2}\left(L_3^{3^2}\left(T_{3^{2-i}}^{3\cdot 3^{2-i}} \otimes I_{3^{i-1}}\right) L_3^{3^2}\left(I_3 \otimes F_3\right) L_3^{3^2}\right)\right), \\
F_{2^4} &= \left(\prod_{i=1}^{3}\left(I_{2^{i-1}} \otimes L_2^{2^{5-i}}\right)\right)\left(\prod_{i=1}^{4}\left(L_{2^3}^{2^4}\left(T_{2^{4-i}}^{2\cdot 2^{4-i}} \otimes I_{2^{i-1}}\right) L_2^{2^4}\left(I_{2^3} \otimes F_2\right) L_{2^3}^{2^4}\right)\right), \\
F_{90000} &= \left(F_{2^4} \otimes I_{5^4\,3^2}\right)\left(L_{2^4}^{3^2\,2^4} \otimes I_{5^4}\right)\left(T_{2^4}^{3^2\cdot 2^4} \otimes I_{5^4}\right)\left(F_{3^2} \otimes I_{5^4\,2^4}\right) L_{3^2\,2^4}^{5^4\cdot 3^2\cdot 2^4} T_{3^2\cdot 2^4}^{5^4\cdot 3^2\,2^4}\left(F_{5^4} \otimes I_{3^2\,2^4}\right).
\end{aligned}
$$

Figure 4: Tensor product formulation for a 90000 points FFT using the Stockham DIF FFT algorithm for the top-level, the Pease DIF FFT algorithm for the bottom-level, and the factorization $5^4 \cdot 3^2 \cdot 2^4$.

algorithm is a factorization of $N = n_1 \cdots n_k$. The integer n_j specifies the size of synthesis at the j-th step. In a mixed-radix algorithm which uses $F_{r_1}, \ldots, F_{r_m}$, each n_j is a power of some r_l, where $1 \leq l \leq m$. A radix-r_l algorithm is used to compute an F_{n_j} at step j. The FFT subsystem also supports algorithms obtained by a "two-level" tensor factorization of the DFT matrix. A two-level FFT algorithm uses different algorithms for computing a "bottom-level" FFT F_{n_j} corresponding to a factor n_j in the factorization of N and the "top-level" FFT corresponding to synthesizing the output from the results of the bottom-level FFTs. Different algorithms can be used for the bottom-level FFTs. However, the current version of the FFT subsystem supports two-level FFT algorithms which use same algorithm for all the bottom-level FFTs. For a given DFT size, different algorithm-factorization pairs have different performance characteristics on the target machine. There are a large number of algorithm-factorization pairs if N is a highly composite number. The FFT subsystem assists an user in finding an efficient implementation for computing a DFT on a specified target machine. As an example of using the FFT subsystem, Fig. 3 shows the window display for an FFT subsystem session for designing a 90000 points FFT for the Cray Y-MP/8 system.

The FFT subsystem takes as an input a file containing design specifications An input file to the FFT subsystem contains a list of algorithm design specifications. In the simplest case, the file can contain just the problem size, which is specified as `fft(N)`. The FFT subsystem algorithm design specifications have the following format:

```
$fft <spec-name> = <spec-parameters>
<spec-name> = toplevel|algo|factor|radix
```

A two-level FFT algorithm is specified by the top-level (`toplevel`) specification in conjunction with the algorithm (`algo`) specification, which specifies the bottom-level algorithm. For example, the FFT window in Fig. 3 shows the input file used for designing a 90000 points FFT. It specifies ST-DIF as the top-level algorithm:

```
$fft toplevel = st_dif
```

The bottom-level algorithm is chosen from the algorithms specified by the following algorithm specification:

```
$fft algo = ct_dif, ct_dit, st_dit, ...
```

The factorization (`factor`) specification specifies the factorizations which are to be used for generating the formulas. A factorization $n_1 \cdot n_2 \cdots n_l$ is specified as $(n_1, n_2, \ldots, n_l)$. For example,

```
$fft factor = (2^4,3^2,5^4),(2^4,5^4,3^2)
```

specifies the factorizations $2^4 \cdot 3^2 \cdot 5^4$ and $2^4 \cdot 5^4 \cdot 3^2$ of 90000.

Usually, DFTs of prime factors of N are used in the computation. However, more efficient implementations can be obtained by using optimized kernels for non-prime DFTs. A radix specification specifies the sizes of the non-prime DFTs to be used in the computation. For example,

```
$fft radix = 4
```

specifies that F_4 is to be used in the computation.

The FFT subsystem can be used to generate tensor product formulas for each algorithm-factorization pair. For example, the algorithm specification shown in the FFT window of Fig. 3 can be used to generate 48 different tensor product formulas (8 bottom-level algorithms $\times$ 6 factorizations). Fig. 4 shows the tensor product formula generated by using ST-DIF for the top-level, PS-DIF for the bottom-level, and the factorization $(5^4, 3^2, 2^4)$. An algorithm-factorization pair can be selected from the selection window (see Fig. 3).

A tensor product file, containing tensor product for-

mulas, is displayed in the tensor product formula window. A tensor product file has five sections: data section, basic operation section, import section, formula section, and main section. The data section contains data allocation specifications for the arrays used in the formula. The basic operation contains the specification of the basic operation in matrix form. The import section is for specifying manually optimized codes for basic computations. The formula section contains the tensor product formulas, and the main section specifies their invocations. The tensor product file in the tensor product formula window of Fig. 3 corresponds to the tensor product formulation shown in Fig. 4.

An architecture subsystem can be used to synthesize a program from the tensor product formulas in a tensor product file. The synthesized program is displayed in the source code window. For example, the source code window in Fig. 3 shows the synthesized code for the Cray Y-MP/8 system with the Cray autotasking directives from the tensor product file displayed in the tensor product formula window.

There are two modes in which a search for an efficient implementation can be conducted: manual and automatic. In the automatic mode, the system automatically executes programs corresponding to each algorithm-factorization pair on the target machine. In the manual mode, the user specifies the next program to be executed by selecting an algorithm-factorization pair. The performance of the best five programs, as well as the performance of the previous three executed programs are displayed in the performance window.

3.2 Strassen Subsystem

The Strassen subsystem is used for developing high-performance programs for dense square matrix multiplication using Strassen's algorithm. It breaks the computation of a larger matrix multiplication into several smaller matrix multiplications and additions by applying a block partitioning. The block partitioning is recursively applied until the block matrix multiplications to be computed can be more efficiently carried out by the conventional matrix multiplication algorithm. A library subroutine, such as **sgemm** from BLAS 3 [6], can be used to perform this block matrix multiplication. The Strassen subsystem also supports Winograd's variation of Strassen's algorithm. This variation reduces the number of additions. Depending on the architecture of the target machine, algorithm, and block size combinations have different performance for a given size.

The Strassen subsystem assists an user in finding an efficient implementation of a matrix multiplication of size $2^n \times 2^n$ on the specified target machine. It takes as an input a file containing design specifications which specify the algorithm, the data allocation scheme, and the block size.

The Strassen subsystem algorithm design specifications have the following format:

```
$str <spec-name> = <spec-parameters>
<spec-name> = algo|alloc|block
```

The algorithm (**algo**) specification specifies which algorithm should be used. Algorithms can be selected from the following list: STR1, STR2, WIN1, and WIN2. The STR1 and WIN1 algorithms require explicit matrix conversion from row/column-major allocation to block-recursive row/column-major allocation. In STR2 and WIN2 algorithms the matrix conversion is distributed over the computation steps. The default algorithm is STR1 algorithm. The allocation (**alloc**) specification specifies the storage of the matrices. It can be one of following four: column-major (COL), row-major (ROW), block recursive column-major (BLK_COL), and block recursive row-major (BLK_ROW). The default allocation is COL. The block (**block**) specification specifies the size of the block matrix for the conventional matrix multiplication.

4 Program Generation

This section describes how programs suitable for various machine architectures can be synthesized from tensor product formulation of an algorithm.

4.1 Program Generation for Sequential Processors

The tensor product formulation of block-recursive algorithms usually involves certain basic computations. Consider for example, the computation corresponding to F_2 in the formulation of the radix-2 FFT algorithm. The following code can be synthesized for $Y^2 = F_2(X^2)$:

$$Y(0) = X(0) + X(1)$$
$$Y(1) = X(0) - X(1)$$

To achieve high performance for the synthesized code, sometimes it is necessary to use manually optimized codes for these basic computations.

Efficient programs can be synthesized from tensor product formulas by exploiting the regular structure in a tensor product. For example, $Y^{mp} = (I_m \otimes B^{p,n})(X^{mn})$ can be interpreted as m copies of $B^{p,n}$ acting in parallel on m disjoint segments of X^{mn}:

do $i = 0, m-1$
$Y^{mp}(i*p : (i+1)*p-1) =$
$B^{p,n}(X^{nm}(i*n : (i+1)*n-1))$
enddo .

It can be noted from the tensor product formulation of the FFT algorithms [18, 24] and the Strassen matrix multiplication algorithm [16, 17] that formulas for block recursive algorithms have the following generic form:

$$\prod_{j=1}^{k} T_j \text{ where } T_j = \begin{cases} (I_{l_j} \otimes A^{m_j,n_j} \otimes I_{r_j}) \\ (I_{l_j} \otimes L_{n_j}^{m_j \times n_j} \otimes I_{r_j}) \end{cases},$$

where A^{m_j,n_j} are computation matrices. The generic tensor product $(I_{l_j} \otimes A^{m_j,n_j} \otimes I_{r_j})$ can be implemented as a doubly nested loop. Therefore, the sequential code for the above formula will have the following loop structure:

do $j = 1, k$
do $s = 0, l_j - 1$
do $t = 0, r_j - 1$
$A^{m_j,n_j}(input, output, j, s, t)$
enddo enddo enddo

The j loop is a sequential loop but the s and t loops can be parallelized. Different implementations can be obtained by changing the order of the two inner loops as they are fully permutable. However, different orderings of the inner loops will result in different data access patterns. This will give rise to codes with different performance characteristics on systems with hierarchical and/or interleaved memories.

4.2 Program Generation for Shared-Memory Vector Multiprocessors

On a vector processor, $Y^{mp} = (B^{p,n} \otimes I_m)(X^{nm})$ can be implemented by using vector operations on n vectors obtained by splitting X^{nm} into consecutive segments of size m. The length of each vector is determined by the dimensions of the identity matrix and each vector is accessed at a unit stride. Hence, $Y^{pm} = (B^{p,n} \otimes I_m)(X^{nm})$ can be implemented as a sequence of vector assignment statements. Implementation of $Y^{pm} = (I_m \otimes B^{p,n})(X^{nm})$ on a vector processor can be obtained by noting that $Y^{pm} = L_m^{pm}(B^{p,n} \otimes I_m)L_n^{nm}(X^{nm})$. Although the vector lengths are still the same as above, the vectors are now accessed at non-unit strides. If the stride distance is a factor or a multiple of the number of the memory modules, this will have an adverse effect on the performance on a vector machine like the Cray Y-MP, which uses memory interleaving to match the memory bandwidth with processor speed.

On a shared-memory multiprocessor system, both $Y^{mp} = (I_m \otimes B^{p,n})(X^{nm})$ and $Y^{mp} = (B^{p,n} \otimes I_m)(X^{nm})$ can be implemented by converting the do loop in the sequential implementation to a parallel doall loop. If each processor has vector functional units, such as in the Cray Y-MP/8 system, then both the vector and multiprocessing capabilities should be exploited. In order to achieve both vectorization and parallelization in both formulas, I_m can be rewritten as $I_{m_1} \otimes I_{m_2}$, where m_1 and m_2 are positive integers such that $m = m_1 \cdot m_2$. The loop corresponding to I_{m_1} can be parallelized and the loop corresponding to I_{m_2} can be vectorized. This can be specified as follows:

$$\begin{aligned} Y^{mp} &= (|[I_{m_1}]| \otimes |<I_{m_2}>| \otimes B^{p,n})(X^{nm}) \text{ and} \\ Y^{mp} &= (B^{p,n} \otimes |[I_{m_1}]| \otimes |<I_{m_2}>|)(X^{nm}), \end{aligned}$$

where encapsulating an identity matrix within $|[\ ]|$ or $|<\ >|$ is used to denote that in the generated code the loop corresponding to that identity matrix should be parallelized or vectorized, respectively. For a given m, the value of m_2 chosen will depend on the size of vector registers.

To synthesize a shared-memory vector program for the radix-2 Cooley-Tukey DIT FFT algorithm, the following specification (ignoring the multiplication by twiddle factors and initial bit-reversal permutation) can be used:

$$F_{2^n} = \prod_{i=1}^{n} (|[I_{2^{i-1}}]| \otimes F_2 \otimes |<I_{2^{n-i}}>|).$$

However, to obtain a high vector length and a fixed degree of parallelism, the following formulation can be used:

$$F_{2^n} = (I_{2^r} \otimes F_{2^s})(F_{2^r} \otimes I_{2^s}),$$

where r and s are positive integers such that $n = r+s$. Therefore, the above formula can be expressed as:

$$F_{2^n} = (|[I_{2^{r_1}}]| \otimes |<I_{2^{r_2}}>| \otimes F_{2^s})(F_{2^r} \otimes |[I_{2^{s_1}}]| \otimes |<I_{2^{s_2}}>|).$$

If 2^{s_2} and 2^{r_2} are sufficiently large, good performance can be expected. However, access of vectors at a stride of 2^s can be detrimental to the performance. One way to achieve unit vector stride is to rewrite $(|[I_{2^{r_1}}]| \otimes |< I_{2^{r_2}}>| \otimes F_{2^s})$ in the above formula using the commutation rule as follows: $L_{2^r}^{2^n}(F_{2^s} \otimes |[I_{2^{r_1}}]| \otimes |<I_{2^{r_2}}>|)L_{2^s}^{2^n}$. Now, unit stride can be achieved at the cost of two permutation steps. Whether or not performance gains are achieved will depend on the target machine characteristics.

We now present performance results for some FFT programs synthesized by EXTENT for the Cray Y-MP/8 shared-memory vector multiprocessor. Table 1

Table 1: Performance of FFT on the Cray Y-MP/8 system.

	2^{18}			3^{11}			5^7		
FFT Form	time (sec.)	CPU%	VL	time (sec.)	CPU%	VL	time (sec.)	CPU%	VL
CT-DIF	0.089	32.6	64.0	0.065	33.1	36.68	0.054	31.9	32.16
ST-DIF	0.068	26.7	64.0	0.057	27.2	36.68	0.053	30.3	34.59
KL-DIF	0.093	30.6	64.0	0.065	24.4	36.68	0.056	22.0	34.59
PS-DIF	0.100	28.8	64.0	0.067	24.8	36.68	0.058	24.6	34.59

shows performance results for FFTs of size 2^{18}, 3^{11}, and 5^7 for programs generated by EXTENT. These programs are synthesized from two-level tensor tensor formulations of an FFT algorithm. The Stockham DIF FFT is used as the top-level FFT algorithm for all the programs. The performance of the programs are shown according to the algorithm used for the bottom-level FFT. The following factorizations were used: $(2^9, 2^9)$ for 2^{18}, $(3^5, 3^6)$ for 3^{11}, and $(5^3, 5^4)$ for 5^7. The table shows the wallclock time (in sec.), percentage of the eight CPUs allocated to the program (CPU% depends upon the system load), and the average vector length for the floating-point operations. The execution times for the best case are within 10% of the execution times obtained for the Cray SciLib FFT routine.

4.3 Program Generation for Distributed-Memory Machines

On distributed-memory multiprocessors, such as the Intel Paragon and the Cray T3D, data-parallelism is exploited by distributing the shared data across the local memories of interconnected processors. Various data distribution schemes can be used to distribute data among processors. The common distributions used are the block, cyclic, and block-cyclic distributions [7]. In the course of the computation, communication is needed when a processor requires data from another processor's local memory. In most distributed-memory multiprocessors the cost of communicating a data value is considerably larger than the cost of a primitive arithmetic computation on the data element. It is therefore important to reduce the communication overhead to achieve high-performance.

A block-cyclic distribution *cyclic*(b) of X^N on P processors partitions the array into blocks of b consecutive elements, and then maps these blocks to the P processors in a cyclic manner. The *block* and *cyclic* distributions are equivalent to *cyclic*($\lceil N/P \rceil$) and *cyclic*(1), respectively. Let us first consider the block distribution of X^{mn} on m processors. Under the block distribution, X^{mn} is split into m segments consisting of n consecutive elements and the i-th segment $X^{mn}(in : (i+1)n-1)$ is assigned to processor i. Next consider the block distribution of the vector obtained by applying L^{mn}_m to X^{mn}. This will result in allocation of $X^{mn}(0 : mn-1 : m)$ to the first processor, $X^{mn}(1 : mn-1 : m)$ to the second processor, and finally $X^{mn}(m-1 : mn-1 : m)$ to the last processor. This corresponds exactly to a cyclic distribution of X^{mn} on m processors. In general, a *cyclic*(b) distribution of X^{mn} can be denoted by the application of $(L^{nm/b}_m \otimes I_b)$ to X^{nm}. Under any block-cyclic distribution of X^{mn} on m processors, n elements are allocated to each processor, which will be denoted by x^n.

For implementing a tensor product formula on a distributed-memory multiprocessor, the first step is to identify data distributions which can be used to implement it in a communication-free manner. If no such distribution exists, then data distributions which lead to the least communication overhead should be chosen. If different components of a formula are communication-free with respect to different data distributions, then the formula can be implemented with communication-free phases separated by redistribution steps. Consider the formula $F \equiv Y^{mp} = (I_m \otimes B^{p,n})(X^{nm})$. We have already seen that F can be implemented using a doall loop corresponding to I_m. If processor i is assigned to perform the computation of the i-th iteration, then it performs $Y^{mp}(ip : (i+1)p-1) = B^{p,n}(X^{nm}(in : (i+1)n-1))$. Under a block distribution of X^{mn} and Y^{mp}, both $X^{nm}(in : (i+1)n-1)$ and $Y^{mp}(ip : (i+1)p-1)$ will be allocated to processor i, implying a communication-free implementation of F. In general, to implement F on q processors in a communication-free manner, I_m in the formula can be rewritten as $I_{k_1} \otimes I_q \otimes I_{k_2}$, where $m = qk_1k_2$. Then, since $F \equiv (L^{k_1q}_q \otimes I_{k_2p})(Y^{mp}) = (I_q \otimes (I_{k_1} \otimes I_{k_2} \otimes B^{p,n}))(L^{k_1q}_q \otimes I_{k_2n})(X^{nm})$, distributing X^{mn} according to a cyclic(k_2n) distribution and Y^{mp} according to a cyclic(k_2p) distribution results in a communication-free implementation with each processor performing $y^{k_1k_2p} = (I_{k_1} \otimes I_{k_2} \otimes B^{p,n})(x^{k_1k_2n})$ in its local space.

Table 2: Execution times (sec.) for FFT on the Cray T3D system with 32 nodes.

	block distribution			cyclic distribution		
FFT Form	2^{18}	3^{11}	5^7	2^{18}	3^{11}	5^7
CT-DIF	0.175	0.109	0.045	0.146	0.102	0.028
ST-DIF	0.168	0.083	0.035	0.141	0.044	0.016
KL-DIF	0.206	0.103	0.043	0.180	0.101	0.027
PS-DIF	0.206	0.103	0.043	0.180	0.101	0.027

Now consider the following tensor product formula which represents the core computation in the Cooley-Tukey FFT:

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

$Z^{2^n} = F_{2^n}(X^{2^n})$ can be implemented on a 2^q processor distributed-memory machine as follows:

1. $L^{2^n}_{2^q}(Y) = (I_q \otimes (\prod_{i=1}^{n-q}(I_{2^{i-1}} \otimes F_2 \otimes I_{2^{n-i-q}})))L^{2^n}_{2^q}(X)$
2. $Z = (I_q \otimes \prod_{i=n-q+1}^{n}(I_{2^{i-1-q}} \otimes F_2 \otimes I_{2^{n-i}}))(Y)$.

This implies that if both the input and the output arrays are distributed cyclically on 2^q processors, then the computation for the first part can be performed without any need for interprocessor communication. Each processor will perform $y^{2^{n-q}} = (\prod_{i=1}^{n-q}(I_{2^{i-1}} \otimes F_2 \otimes I_{2^{n-i-q}}))(x^{2^{n-q}})$ in its local memory. Similarly, the second part of the computation can be implemented in a communication-free manner by distributing Z^{2^n} and Y^{2^n} in a block-wise fashion on 2^q processors.

The two parts of the computation have different distribution requirements for array Y^{2^n}. There are two obvious implementations possible: 1) Let all three arrays have the same fixed distribution (either block or cyclic distribution) 2) Initially distribute array X and Y cyclically and array Z block-wise, perform the first part of the computation, change the distribution of Y to a block distribution, and finally perform the second part of the computation. With the first implementation, either the first or the second part of the computation will be communication-free and the other part of the computation would require communication, depending upon whether cyclic or block distribution is chosen. On the other hand, the second implementation requires communication to redistribute array Y from a cyclic to block distribution, i.e., $L^{2^n}_{2^q}(Y^{2^n})$ should be transformed to Y^{2^n}. This can be achieved by applying $L^{2^n}_{2^{n-q}}$ to $L^{2^n}_{2^q}(Y^{2^n})$. Assuming that $n \geq 2q$, this requires an all-to-all communication, where each processor sends $y^{2^{n-q}}(j2^{n-2q} : (j+1)2^{n-2q}-1)$ to processor j. Assuming the time to communicate a message of size m is $t_s + mt_p$, where t_s is the message latency and t_p is the link transfer time per data element, the communication-overhead is: $2^q(t_s + 2^{n-2q}t_p) = 2^q t_s + 2^{n-q}t_p$. It can be easily determined that the first implementation will require q communication steps with the total communication cost of $q(t_s + 2^{n-q}t_p) = qt_s + q2^{n-q}t_p$. Depending upon the target machine parameters t_s and t_p, one implementation will have a better performance than the other.

We now present performance results for FFT programs synthesized by EXTENT for the Cray T3D. Table 2 shows the wallclock time (in sec.) for some FFT programs. Timings are shown for programs with either block or cyclic distribution of the data. All these programs use the shared-memory communication primitives. It can be seen that the performance variation between different forms can vary by more than a factor of two.

5 Conclusions

In this paper, we have presented the EXTENT programming environment for automatic generation of parallel/vector programs from tensor product formulas. It greatly aids in the search of the best implementation for a block recursive algorithm on several different parallel machines such as the Cray Y-MP, Cray T3D, and Intel Paragon. For certain important applications, EXTENT provides subsystems for automatic generation of tensor product formulas from problem specifications. Thus, for developing programs for these applications no knowledge of tensor products is required. However, only little training in tensor products is needed to fine tune the performance of the generated code by modifying the parallelization and data distribution directives in the generated tensor product formulas.

Acknowledgments

We are grateful to the Ohio Supercomputer Center for providing access to the Cray Y-MP/8 system and the Cray T3D system. We are also grateful to the San Diego Supercomputer Center for providing access to the Intel Paragon.

References

[1] J. R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Trans. Programming Languages and Systems*, 9:491–542, 1987.

[2] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr. Implementing a parallel C++ runtime system for scalable parallel systems. In *Supercomputing '93*, pages 588–597, Nov. 1993.

[3] J. W. Brewer. Kronecker Products and matrix calculus in system theory. *IEEE Transaction on Circuits and Systems*, 25:772–781, 1978.

[4] B. M. Chapman, P. Mehrotra, and H. P. Zima. Vienna Fortran – a Fortran language extension for distributed memory multiprocessors. In J. Saltz and P. Mehrotra, editors, *Language, Compilers and Runtime Environments for Distributed Memory Machines*, pages 39–62. North-Holland, Amsterdam, 1992.

[5] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. ScaLAPACK: A scalable linear algebra library for distributed-memory concurrent computers. In *Proc. of Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127, Oct. 1992.

[6] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Math. Software*, 16(1):1–17, 1990.

[7] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Rice University, 1993.

[8] H. M. Gerndt and H. P. Zima. Optimizing communication in SUPERB. In *CONPAR 90*, pages 300–311, 1990. LNCS 457.

[9] A. Graham. *Kronecker Products and Matrix Calculus: With Applications*. Ellis Horwood Limited, 1981.

[10] J. Granta, M. Conner, and R. Tolimieri. Recursive fast algorithms and the role of the tensor product. *IEEE Transaction on Signal Processing*, 40(12):2921–2930, Dec. 1992.

[11] F. A. Graybill. *Matrices, with applications in Statistics*. Wadsworth International Group, Belmont, CA, 1983.

[12] S. K. S. Gupta, S. D. Kaushik, C.-H. Huang, J. R. Johnson, R. W. Johnson, and P. Sadayappan. A methodology for the generation of data distributions to optimize communication. In *Fourth IEEE Symposium on Parallel and Distributed Processing*, pages 436–441, Dec. 1992.

[13] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran-D for MIMD distributed-memory machines. *Communication of the ACM*, 35(8):66–80, Aug. 1992.

[14] R. A. Horn and C. R. Johnson. *Topics in Matrix Analysis*. Cambridge University Press, Cambridge, 1991.

[15] E. N. Houstis and J. R. Rice. Parallel ELLPACK: A development and problem solving environment for high performance computing machines. In P. W. Gaffney and E. N. Houstis, editors, *Programming environments for high-level scientific problem solving*, pages 229–241, 1992.

[16] C.-H. Huang, J. R. Johnson, and R. W. Johnson. A tensor product formulation of Strassen's matrix multiplication algorithm. *Appl. Math. Letters*, 3(3):67–71, 1990.

[17] C.-H. Huang, J. R. Johnson, and R. W. Johnson. Generating parallel programs from tensor product formulas: A case study of Strassen's matrix multiplication algorithm. In *Proc. International Conference on Parallel Processing 1992*, volume III, pages 104–108, Aug. 1992.

[18] J. R. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri. A methodology for designing, modifying and implementing fourier transform algorithms on various architectures. *Circuits Systems Signal Process*, 9(4):450–500, 1990.

[19] S. D. Kaushik, S. Sharma, and C.-H. Huang. An algebraic theory for modeling multistage interconnection networks. *Journal of Information Science and Engineering*, 9:1–26, 1993.

[20] S. D. Kaushik, S. Sharma, C.-H. Huang, J. R. Johnson, R. W. Johnson, and P. Sadayappan. An algebraic theory for modeling direct interconnection networks. In *Supercomputing '92*, Nov. 1992.

[21] C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, Bruce Leung, and D. Schouten. Parafrase-2: An environment for parallelizing, partitioning synchronizing, and scheduling programs on multiprocessors. In *Proc. of International Conference on Parallel Processing*, volume II, pages 39–48, 1989.

[22] P. A. Regalia and S. K. Mitra. Kronecker products, unitary matrices and signal processing applications. *SIAM Reviews*, 31(4):586–613, Dec. 1989.

[23] G. X. Ritter and P. D. Gader. Image algebra techniques and parallel image processing. *J. Parallel and Distributed Computing*, 4:7–44, 1987.

[24] C. Van Loan. *Computational frameworks for the fast Fourier transform*. SIAM, 1992.

[25] H. P. Zima, H.-J. Bast, and H. M. Gerndt. SUPERB - a tool for semi-automatic MIMD/SIMD parllelization. *Parallel Computing*, 6:1–18, 1988.