Efficient Data Parallel Algorithms for Multidimensional Array Operations Based on the *EKMR* Scheme for Distributed Memory Multicomputers

Chun-Yuan Lin, *Student Member*, *IEEE Computer Society*,
Yeh-Ching Chung, *Member*, *IEEE Computer Society*, and Jen-Shiuh Liu

Abstract—Array operations are useful in a large number of important scientific codes, such as molecular dynamics, finite element methods, climate modeling, atmosphere and ocean sciences, etc. In our previous work, we have proposed a scheme *extended Karnaugh map representation* (*EKMR*) for multidimensional array representation. We have shown that sequential multidimensional array operation algorithms based on the *EKMR* scheme have better performance than those based on the *traditional matrix representation* (*TMR*) scheme. Since parallel multidimensional array operations have been an extensively investigated problem, in this paper, we present efficient data parallel algorithms for multidimensional array operations based on the *EKMR* scheme for distributed memory multicomputers. In data parallel programming paradigm, in general, we distribute array elements to processors based on various distribution schemes, do local computation in each processor, and collect computation results from each processor. Based on the row, the column, and the 2D mesh distribution schemes, we design data parallel algorithms for *matrix-matrix addition* and *matrix-matrix multiplication* array operations in both *TMR* and *EKMR* schemes for multidimensional arrays. We also design data parallel algorithms for six Fortran 90 array intrinsic functions, *All*, *Maxval*, *Merge*, *Pack*, *Sum*, and *Cshift*. We compare the time of the data distribution, the local computation, and the result collection phases of these array operations based on the *TMR* and the *EKMR* schemes. The experimental results show that algorithms based on the *EKMR* scheme outperform those based on the *TMR* scheme for all test cases.

Index Terms—Data parallel algorithm, array operation, multidimensional array, data distribution, Karnaugh map.

1 Introduction

Array operations are useful in a large number of important scientific codes, such as molecular dynamics [14], finite-element methods [22], climate modeling [41], atmosphere and ocean sciences [16], etc. In the literature, many methods have been proposed to implement these array operations efficiently. However, the majority of these methods for two-dimensional arrays usually do not perform well when extended to higher dimensional arrays. The reason is that one usually uses the *traditional matrix representation* (*TMR*) [35] to represent higher dimensional arrays. This scheme has two drawbacks [35] for higher dimensional array operations. First, the costs of index computation of array elements for array operations increase as the dimension increases. Second, the cache miss rate for array operations increases as the dimension increases due to more cache lines accessed.

- *C.-Y. Lin and J.-S. Liu are with the Department of Information Engineering, Feng Chia University, Taichung, Taiwan 407, ROC. E-mail: {cylin, liuj}@iecs.fcu.edu.tw.*
- *Y.-C. Chung is with the Department of Computer Science, National Tsing-Hua University, Hsinchu, Taiwan 300, ROC. E-mail: ychung@cs.nthu.edu.tw.*

Manuscript received 4 Dec. 2000; revised 26 June 2002; accepted 20 Oct. 2002.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 113234.

In our previous work [35], we have proposed a new scheme called *extended Karnaugh map representation* (*EKMR*) for multidimensional array representation. Since parallel multidimensional array operations [7], [10], [11], [16] have been an extensively investigated problem, in this paper, we present efficient data parallel algorithms for multidimensional array operations based on the *EKMR* scheme for distributed memory multicomputers. In data parallel programming paradigm, in general, we distribute array elements to processors based on various distribution schemes (the data distribution phase), do local computation in each processor (the local computation phase), and collect computation results from each processor (the result collection phase).

For one-dimensional array, the distributed scheme can be block, cyclic, or block-cyclic [23]. For higher dimensional array, these distribution schemes can be applied to each dimension. Therefore, there are many combinations of array distribution schemes for higher dimensional arrays. In this paper, we focus on the design based on the row, the column, and the 2D mesh distribution schemes [7], [10], [11], [16], [29], [30], [31], [44]. For two-dimensional arrays, the row, the column, and the 2D mesh distribution schemes are similar to (Block, *), (*, Block), and (Block, Block) used in Fortran 90 [1], respectively. For three or higher dimensional arrays, the row, the column, and the 2D mesh distribution

schemes are similar to (*, ..., Block, *), (*,..., *, Block), and (*, ..., Block, Block), respectively.

Based on the distribution schemes described above, we design data parallel algorithms for *matrix-matrix addition* and *matrix-matrix multiplication* array operations based on the *TMR* and the *EKMR* schemes for multidimensional arrays. We also design data parallel algorithms for six Fortran 90 array intrinsic functions, *All*, *Maxval*, *Merge*, *Pack*, *Sum*, and *Cshift*. To evaluate the performance of the designed data parallel algorithms, we compare the time of the data distribution, the local computation, and the result collection phases of these array operations based on the *TMR* and the *EKMR* schemes. The experimental results show that the execution time of array operations based on the *EKMR* scheme is less than that of those based on the *TMR* scheme in the data distribution, the local computation, and the result collection phases. For the data distribution and the result collection phases, the reason is that the cost of packing/unpacking array elements for the *EKMR* scheme is less than that for the *TMR* scheme since the number of noncontinuous data blocks in the *EKMR* scheme is less than that in the *TMR* scheme. For the local computation phase, the reasons are two-fold. First, the *EKMR* scheme can reduce the costs of index computation of array elements for array operations because it uses a set of two-dimensional arrays to represent a higher dimensional array. Second, the cache miss rate for array operations based on the *EKMR* scheme is less than that based on the *TMR* scheme because the number of cache lines accessed by array operations based on the *EKMR* scheme is less than that based on the *TMR* scheme.

This paper is organized as follows: In Section 2, a brief survey of related work will be presented. We will briefly describe the *EKMR* scheme for multidimensional array representation in Section 3. In Section 4, we will discuss the implementations of data parallel algorithms of multidimensional array operations based on the *TMR* and *EKMR* schemes with various distribution schemes. The performance comparisons and experimental results of these data parallel algorithms will be given in Section 5.

2 Related Work

Many methods for improving array computation have been proposed in the literature. Carr et al. [5] and McKinley et al. [36] presented a comprehensive approach to improving data locality by using loop transformations such as loop permutation, loop reversal, etc. They demonstrated that these transformations are useful for optimizing many array programs. They also proposed an algorithm called *LoopCost* to analyze and construct the cost models for variable loop orders of array operations. Kandemir et al. [24], [25] proposed a compiler technique to perform loop and data layout transformations to solve the global optimization problem on sequential and multiprocessor machines. They use loop transformations to find the best loop order of an array operation to solve the global optimization problems. O'Boyle and Knijnenburg [37] presented a new algebraic framework to combine loop and data layout transformations. By integrating loop and data layout transformations, any poor spatial locality and expensive array subscripts can be eliminated. Sularycke and Ghose [39] proposed a simple sequential loop interchange algorithm that can produce a better performance than existing algorithms for array multiplication.

Chatterjee et al. [9] examined two nonlinear data layout functions (*4D* and *Morton*) for two-dimensional arrays with the tiling scheme that promises improved performance at low cost. In [8], they further examined the combination of five recursive data layout functions (various forms of *Morton* and *Hilbert*) with the tiling scheme for three parallel matrix multiplication algorithms. Coleman and McKinley [13] presented a new algorithm *TSS* for choosing problem-size dependent tile size based on the cache size and cache line size for a direct-mapped cache. Wolf and Lam [42] proposed an algorithm that improves the locality of a loop nest by transforming the code via interchange, reversal, skewing, and tiling. In [33], they also presented a comprehensive analysis of the performance of blocked code on machines with caches. Frens et al. [21] presented a simple recursive algorithm with the quad-tree decomposition of matrices that has outperformed hand-optimized BLAS3 matrix multiplication. The use of quad-trees or oct-trees is known in parallel computing [3] for improving both load balance and locality. Carter et al. [6] focused on using hierarchical tiling to exploit superscalar-pipelined processor. The hierarchical tiling is a framework for applying known tiling methods to ease the burden on several compiler phases that are traditionally treated separately.

Kotlyar et al. [29], [30], [31] presented a relational algebra based framework for compiling efficient sparse array code from dense DO-Any loops and a specified sparse array. Fraguela et al. [17], [18], [19], [20] analyzed the cache effects for the array operations. They established the cache probabilistic model and modeled the cache behavior for sparse array operations. Kebler and Smith [26] described a system, SPARAMAT, for concept comprehension that is particularly suitable for sparse array codes. Ziantz et al. [44] proposed a runtime optimization technique that can be applied to a compressed row storage array for array distribution and off-processor data fetching in order to reduce both the communication and computation time. Chang et al. [7] and Chung et al. [10], [11] presented an efficient libraryfor parallel sparse computations with Fortran 90 array intrinsic operations. They provide a new data distribution scheme for multidimensional sparse arrays. Their scheme is similar to (*, ..., Block, Block) data distribution scheme used in Fortran 90.

3 The *EKMR* Scheme

In [35], we have proposed the *EKMR* scheme for multidimensional array representation. Before presenting efficient data parallel algorithms based on the *EKMR* scheme, we briefly describe the *EKMR* scheme for multidimensional arrays. Details of the *EKMR* scheme can be found in [35]. We describe the *EKMR* and *TMR* schemes based on the row-major data layout [12]. The idea of the *EKMR* scheme is based on the Karnaugh map. Fig. 1 shows examples of n-input Karnaugh maps for $n = 1, \ldots, 4$. When n is less than or equal to 4, an n-input Karnaugh map can be drawn on a plane easily, that is, it can be represented

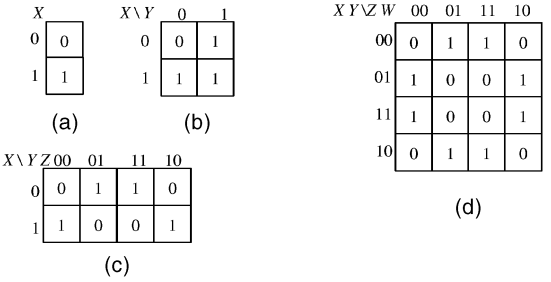

Fig. 1. Examples of the Karnaugh map. (a) 1-input for $f = X$. (b) 2-input for $f = X + Y$. (c) 3-input for $f = XZ' + X'Z$. (d) 4-input for $f = YW' + Y'W$.

by a two-dimensional array. Consequently, when $n = 1$ and 2, the *EKMR*(1) and the *EKMR*(2) is simply a one-dimensional array and the traditional two-dimensional array, respectively.

Let *A*[*k*][*i*][*j*] denote a three-dimensional array based on the *TMR*(3) with a size of $3 \times 4 \times 5$. The corresponding *EKMR*(3) of array *A*[3][4][5] with the size of 4×15, is shown in Fig. 2. In the *EKMR*(3), the index variable i' is the same as the index variable *i* andthe index variablej' is a combination of the index variables *j* and *k*. A more concrete example based on the row-major data layout is given in Fig. 3.

Let *A*[*l*][*k*][*i*][*j*] denote a four-dimensional array based on the *TMR*(4) with a size of $2 \times 3 \times 4 \times 5$. Fig. 4 illustrates a corresponding *EKMR*(4) of array *A*[2][3][4][5] with a size of 8×15. In the *EKMR*(4), the index variable i' is a combination of the index variables *l* and *i* and the index variable j' is a combination of the index variables *j* and *k*.

Based on the *EKMR*(4), we can generalize our results to the *n*-dimensional array. Assume that there is an *n*-dimensional array based on the *TMR*(*n*) with a size of *m* along each dimension. Since the *EKMR*(*n*) can be represented by m^{n-4} *EKMR*(4), we use a one-dimensional array *X* with a size of m^{n-4} to link these *EKMR*(4). Fig. 5 shows the corresponding *EKMR*(6), represented by six arrays based on the *EKMR*(4) each with a size of 8×15, of array *A*[*n*][*m*][*l*][*k*][*i*][*j*]. In Fig. 5, a one-dimensional array *X* with a size of six is used to link these *EKMR*(4).

4 The Design of Data Parallel Algorithms of Multidimensional Array Operations Based on the *TMR* and the *EKMR* Schemes

The design of a data parallel program, in general, can be divided into three phases: data distribution, local computation, and result collection. In the following, we examine the design issues of these three phases for data parallel

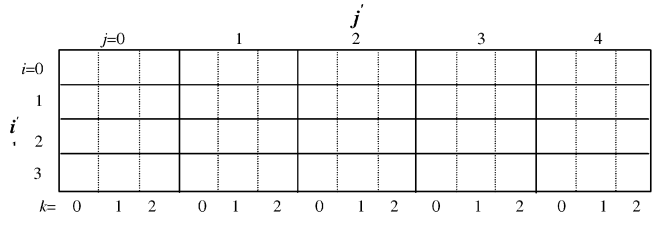

Fig. 2. The *EKMR*(3) scheme.

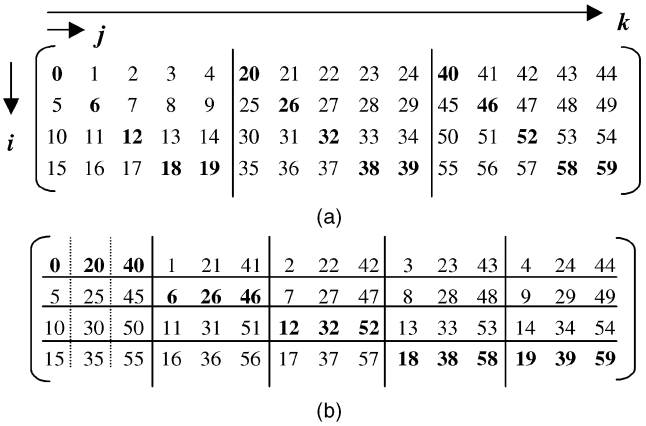

Fig. 3. (a) A three-dimensional array in the *TMR*(3). (b) The corresponding *EKMR*(3).

algorithms of multidimensional array operations based on the *TMR* and the *EKMR* schemes. The *TMR* and the *EKMR* both are representation schemes for multidimensional arrays. Different data layout functions can be applied to them to get different data layouts. In the following, we design data parallel algorithms of multidimensional array operations according to the row-major data layout. We do not consider algorithms based on the recursive data layout [8], [9], [21]. The reason is that how to select a recursive data layout such that a multidimensional array operation algorithm based on the *TMR* scheme having the best performance is an open question [8], [9]. Since there are many array operations, we use the *matrix-matrix multiplication* array operation as the design example. Other array operations can be implemented in a similar manner. Since the distribution of two or more arrays for an array operation will result in a complex data parallel algorithm design, we only consider the case for the distribution of a single array.

4.1 The Data Distribution Phase for the *TMR* and the *EKMR* Schemes

In this paper, we focus on the implementations based on the row, the column, and the 2D mesh distribution schemes. Below, we describe the implementations of the data distribution phase for the *TMR* and *EKMR* schemes based on these three distribution schemes. To distribute an array to processors, it consists of three steps. In the first step, an array needs to be partitioned into chucks based on a distribution scheme. In the second step, array elements in a chuck may need to be packed before distributing to the

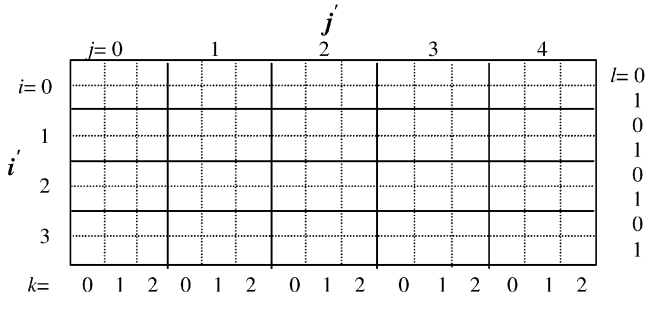

Fig. 4. The *EKMR*(4) scheme.

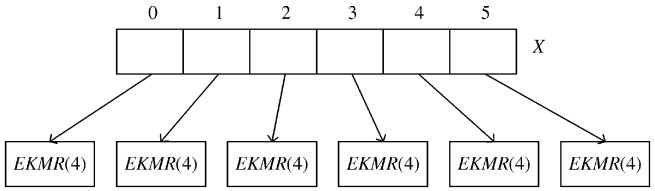

Fig. 5. An example of the *EKMR*(6).

corresponding processor. In the third step, a packed array is distributed to the corresponding processor. Since the first and third steps require the same cost for the row, the column, and the 2D mesh distribution schemes, to evaluate the performance of these distribution schemes, we will focus on the packing cost of the second step. We assume that an $n \times n \times n$ array, $A[k][i][j]$, and P processors are given.

4.1.1 The Row Distribution Scheme

For *TMR*(3), $A[k][i][j]$ is distributed to processors by partitioning the array into P three-dimensional arrays along the i direction. More precisely, let *row_size* denote the number of rows assigned to each processor. The value of *row_size* is equal to $\lceil n/p \rceil$ for the first r processors and $\lfloor n/p \rfloor$ for the remaining $P - r$ processors, where r is the remainder of n divided by P. Since array elements of the partial arrays assigned to each processor are not stored in consecutive memory locations, they need to be packed before distributing to processors. An example of the row distribution scheme for *TMR*(3) is shown in Fig. 6a. Assume that the buffer D is used to pack array elements. The algorithm of the row distribution scheme for *TMR*(3) is shown in Fig. 7.

We have mentioned that array elements of the partial arraysneedto be packed before distributing to processors since they are not stored in consecutive memory locations. We use the number of noncontinuous data blocks to evaluate the cost of packing array elements. The number of noncontinuous data blocks indicates how many jumps are needed during the packing, which affects cache usage or disk seeking time. From Fig. 6a and the algorithm of the row distribution scheme based on *TMR*(3), the number of noncontinuous data blocks on a processor is n. For P processors, in *TMR*(3), the number of noncontinuous data blocks is $P \times n$. For the *TMR*(n), where $n > 3$, the distribution is similar to that of *TMR*(3). The number of noncontinuous data blocks in the *TMR*(n) on P processors is $P \times n^{n-2}$.

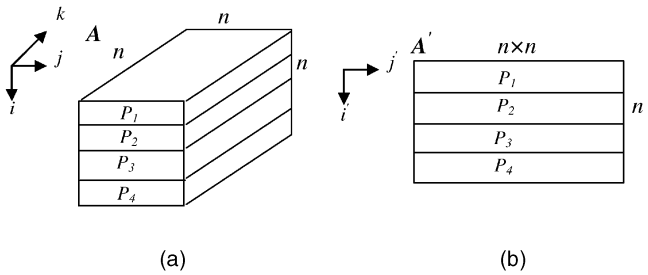

Fig. 6. The row distribution scheme for arrays A and A' to four processors. (a) The array A based on *TMR*(3). (b) The array A' based on *EKMR*(3).

```
Algorithm row_distribution_scheme_TMR(3)
    1.  for ( p_id = 0 ; p_id < P ; p_id ++)
    2.      l = 0 ;
    3.      offset = p_id × row_size ;   /*Pack array elements into buffer D*/
    4.      for ( k = 0 ; k < n ; k++)
    5.          for ( i = 0 ; i < row_size ; i++)
    6.              for ( j = 0 ; j < n ; j++)
    7.                  D[l] = A[k][ i + offset ][j] ;
    8.                  l++;
    9.      Distribute D[] to appropriate processor;
end_of_row_distribution_scheme_TMR(3)
```

Fig. 7. Algorithm of the row distribution scheme for *TMR*(3).

Let $A'[i'][j']$ be the corresponding *EKMR*(3) of $A[k][i][j]$. From the definition of *EKMR*(3), array A' consists of n rows and each row contains n^2 array elements. In the row distribution scheme, $A'[i'][j']$ is distributed to processors by partitioning the array into P two-dimensional arrays along the i' direction. The value of *row_size* is the same as that in the case of *TMR*(3). Since array elements in the same row are stored in consecutive memory locations in *EKMR*(3), they do not need to be packed before distributing to processors. An example of the row distribution scheme for *EKMR*(3) is shown in Fig. 6b. The algorithm of the row distribution scheme for *EKMR*(3) is shown in Fig. 8.

From Fig. 6b and the algorithm of the row distribution scheme based on *EKMR*(3), the numbers of noncontinuous data blocks on a processor and P processors are both 0. For *EKMR*(4), the numbers of noncontinuous data blocks on a processor and P processors are both 0. For *EKMR*(n), where $n > 4$, array elements of the partial arrays assigned to each processor are not stored in consecutive memory locations. They need to be packed before distributing to processors. An example is shown in Fig. 9.

Let $A[m_{n-4}][m_{n-3}] \ldots [m_1][l][k][i][j]$ be an n^n n-dimensional array for the *TMR*(n), where $n > 4$. Let array A' be the corresponding *EKMR*(n) of array A. Let $A'_{(m_{n-4},m_{n-3},\ldots m_1)}$ be a corresponding *EKMR*(4) with the size of $n^2 \times n^2$. The algorithm of the row distribution scheme for *EKMR*(n) is shown in Fig. 10.

From Fig. 9 and the algorithm of the row distribution scheme based on *EKMR*(n), the number of noncontinuous data blocks on P processors is $P \times n^{n-4}$. Therefore, in the row distribution scheme, the number of noncontinuous data blocks in the *EKMR* scheme is less than that in the *TMR* scheme.

4.1.2 The Column Distribution Scheme

For *TMR*(3), $A[k][i][j]$ is distributed to processors by partitioning array A into P three-dimensional arrays along the j direction. More precisely, let *column_size* denote the number of columns assigned to each processor. The value of *column_size* is equal to $\lceil n/p \rceil$ for the first r processors and

```
Algorithm row_distribution_scheme_EKMR(3)
    1.  for ( p_id =0 ; p_id < P ; p_id ++ )
    2.      offset = p_id × row_size ;
    3.      Distribute (A' + offset) → (A' + 2×offset-1) to appropriate processor;
end_of_row_distribution_scheme_EKMR(3)
```

Fig. 8. Algorithm of the row distribution scheme for *EKMR*(3).

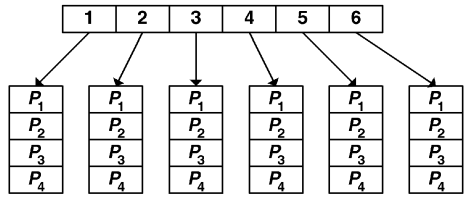

Fig. 9. The row distribution scheme for array A' to four processors based on *EKMR*(6).

```
Algorithm row_distribution_scheme_EKMR(n)
    1. for ( p_id = 0 ; p_id < P ; p_id ++)
    2.     l = 0 ;
    3.     offset = p_id × row_size ;
    4.     for ( x = 0 ; x < n^(n-4) ; x++)
    5.         for ( i = 0 ; i < row_size ; i++)
    6.             for ( j = 0 ; j < n^2 ; j++)
    7.                 D[l] = A'x[ i + offset ][j] ;
    8.                 l++;
    9.     Distribute D[] to appropriate processor;
end_of_row_distribution_scheme_EKMR(n)
```

Fig. 10. Algorithm of the row distribution scheme for *EKMR*(*n*).

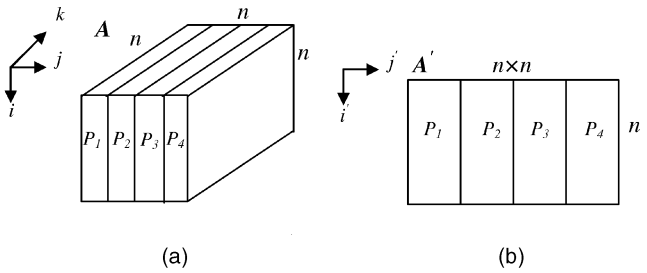

Fig. 11. The column distribution scheme for arrays A and A' to four processors. (a) The array A based on *TMR(3)*. (b) The array A' based on *EKMR(3)*.

$\lfloor n/p \rfloor$ for the remaining $P - r$ processors, where r is the remainder of n divided by P. Since array elements of the partial arrays assigned to each processor are not stored in consecutive memory locations, they need to be packed before distributing to processors. An example of the column distribution scheme for *TMR*(3) is shown in Fig. 11a. The algorithm of the column distribution scheme for *TMR*(3) is shown in Fig. 12.

For *TMR*(*n*), where $n > 3$, the distribution is similar to that of *TMR*(3). From Fig. 11a and the algorithm of the column distribution scheme based on *TMR*(3), the number of noncontinuous data blocks on a processor and P processors is n^2 and $P \times n^2$, respectively. For *TMR*(*n*), where $n > 3$, the distribution is similar to that of *TMR*(3). The number of noncontinuous data blocks in *TMR*(*n*) on P processors is $P \times n^{n-1}$.

For *EKMR*(3), in the column distribution scheme, $A'[i'][j']$ is distributed to processors by partitioning the array into P two-dimensional arrays along the j' direction. The value of *column_size* for *EKMR*(3) is equal to $\lceil n^2/p \rceil$ for the first r processors and $\lfloor n^2/p \rfloor$ for the remaining $P - r$ processors. If n can not be divided by P, there are only $\lceil n^2/p \rceil n$ or $\lfloor n^2/p \rfloor n$ array elements in each processor for *EKMR*(3). However, there are $\lceil n/p \rceil n^2$ or $\lfloor n/p \rfloor n^2$ array elements in each processor for *TMR*(3). The load-balancing of data parallel

```
Algorithm column_distribution_scheme_TMR(3)
    1. for ( p_id = 0 ; p_id < P ; p_id ++)
    2.     l = 0 ;
    3.     offset = p_id × column_size ;   /*Pack array elements into buffer D*/
    4.     for ( k = 0 ; k < n ; k++)
    5.         for ( i = 0 ; i < n ; i++)
    6.             for ( j = 0 ; j < column_size ; j++)
    7.                 D[l] = A[k][i][ j + offset ] ;
    8.                 l++;
    9.     Distribute D[] to appropriate processor;
end_of_column_distribution_scheme_TMR(3)
```

Fig. 12. Algorithm of the column distribution scheme for *TMR*(3).

```
Algorithm column_distribution_scheme_EKMR(3)
    1. for ( p_id = 0 ; p_id < P ; p_id ++)
    2.     l = 0 ;
    3.     offset = p_id × column_size ;
    4.     for ( i = 0 ; i < n ; i++)
    5.         for ( j = 0 ; j < column_size ; j++)
    6.             D[l] = A'[i][ j + offset ] ;
    7.             l++;
    8.     Distribute D[] to appropriate processor;
end_of_column_distribution_scheme_EKMR(3)
```

Fig. 13. Algorithm of the column distribution scheme for *EKMR*(3).

```
Algorithm column_distribution_scheme_EKMR(n)
    1. for ( p_id = 0 ; p_id < P ; p_id ++)
    2.     l = 0 ;
    3.     offset = p_id ×column_size ;
    4.     for ( x = 0 ; x < n^(n-4) ; x++)
    5.         for ( i = 0 ; i < n ; i++)
    6.             for ( j = 0 ; j < column_size ; j++)
    7.                 D[l] = A'x[i][ j + offset ] ;
    8.                 l++;
    9.     Distribute D[] to appropriate processor;
end_of_column_distribution_scheme_EKMR(n)
```

Fig. 14. Algorithm of the column distribution scheme for *EKMR*(*n*).

algorithms based on *EKMR*(3) is better than those based on *TMR*(3). Since array elements of the partial arrays assigned to each processor are not stored in consecutive memory locations, they need to be packed before distributing to processors. An example of the column distribution scheme for *EKMR*(3) is shown in Fig. 11b. The algorithms of the column distribution scheme for *EKMR*(3) is described in Fig. 13.

From Fig. 11b and the algorithm of the column distribution scheme based on *EKMR*(3), the number of noncontinuous data blocks on a processor and P processors is n and $P \times n$, respectively. For *EKMR*(*n*), where $n > 3$, the distribution is similar to that of *EKMR*(3). The algorithm of the column distribution scheme for *EKMR*(*n*) is Fig. 14.

An example of the column distribution scheme for *EKMR*(6) is shown in Fig. 15. From Fig. 15 and the algorithm of the column distribution scheme based on *EKMR*(*n*), the number of noncontinuous data blocks on P processors is $P \times n^{n-2}$. Therefore, in the column distribution scheme, the number of noncontinuous data blocks in the *EKMR* scheme is less than that in the *TMR* scheme.

4.1.3 The 2D Mesh Distribution Scheme

For *TMR*(3) and *EKMR*(3), the 2D mesh distribution scheme inherits the characteristics of the row and the column

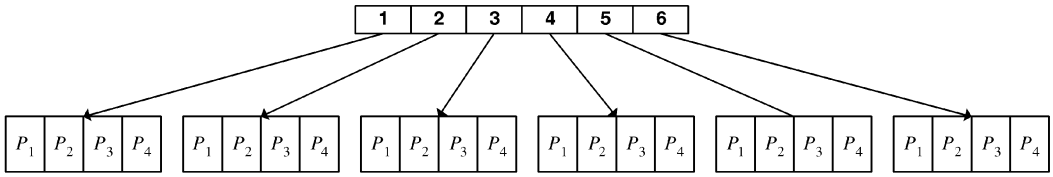

Fig. 15. The column distribution scheme for array A' to four processors based on *EKMR*(6).

distribution schemes. Since array elementsof the partial arrays assigned to each processor are not stored in consecutive memory locations, they need to be packed before distributing to processors. Examples of the 2D mesh distribution scheme for *TMR*(3) and *EKMR*(3) are shown in Fig. 16a and Fig. 16b, respectively.

Assume that an $n \times n \times n$ array, $A[k][i][j]$, and $P \times Q$ processors are given. The algorithms of the 2D mesh distribution scheme for *TMR*(3) and *EKMR*(3) are shown in Fig. 17.

From Fig. 16a and the algorithm of the 2D distribution scheme based on *TMR*(3), the number of noncontinuous data blocks on a processor and $P \times Q$ processors is n^2/P and $Q \times n^2$, respectively. From Fig. 16b and the algorithm of the 2D distribution scheme based on *EKMR*(3), the number of noncontinuous data blocks on a processor and $P \times Q$ processors is n/P and $Q \times n$, respectively. For *TMR*(*n*) and *EKMR*(*n*), the distributions are similar to those of *TMR*(3) and *EKMR*(3), respectively. The algorithm of the 2D mesh distribution scheme for *EKMR*(*n*) is shown in Fig. 18.

An example of the 2D mesh distribution scheme for *EKMR*(6) is shown in Fig. 19. The number of noncontinuous data blocks in *TMR*(*n*) and *EKMR*(*n*) in $P \times Q$ processors is $Q \times n^{n-1}$ and $Q \times n^{n-2}$, respectively. Therefore, in the 2D mesh distribution scheme, the number of noncontinuous data blocks in the *EKMR* scheme is less than that in the *TMR* scheme.

4.2 The Local Computation Phase for the *TMR* and *EKMR* Schemes

After distributing single array to processors, the next step is to perform the computation based on the distributed arrays. In general, the work in the local computation phase is the same as the sequential algorithm with some changes in the scope of operated data, i.e., changes the scope of loop indices for array operations. For the row or the column distribution schemes, we change the scope of the row or the column loop index, respectively. For the 2D mesh distribution scheme, we change the scope of the row and the column loop indices.

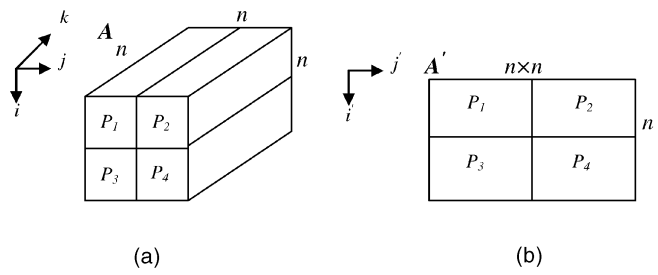

Fig. 16. The 2D mesh distribution scheme for arrays A and A' to four processors. (a) The array A based on *TMR*(3). (b) The array A' based on *EKMR*(3).

Given P processors, let $A[m_{n-4}][m_{n-3}]\ldots[m_1][l][k][i][j]$ and $B[m_{n-4}][m_{n-3}]\ldots[m_1][l][k][i][j]$ be two n-dimensional arrays with size of n^n for the *TMR*(*n*); arrays A' and B' be the corresponding *EKMR*(*n*) of arrays *A* and *B*, respectively; $A'_{(m_{n-4},m_{n-3},\ldots,m_1)}$ and $B'_{(m_{n-4},m_{n-3},\ldots,m_1)}$ be two corresponding *EKMR*(4); C and C' be local arrays for *TMR*(*n*) and *EKMR*(*n*) in each processor, respectively. The algorithms of the local computation phase for a *matrix-matrix multiplication* array operation based on *TMR*(*n*) and *EKMR*(*n*) with the row distribution scheme is shown in Fig. 20. In order to exploit advantages for the structure of the *EKMR* scheme, we also use the concepts provided by O'Boyle et al. [37] to design algorithms of the local computation phase for *matrix-matrix multiplication* array operation based on the *EKMR* scheme.

For the column and the 2D mesh distribution schemes, algorithms of the local computation phase for the *matrix-matrix multiplication* array operation based on *TMR*(*n*) and *EKMR*(*n*) are similar to that based on the row distribution scheme. Since the page limitation, they are omitted in this paper.

To evaluate the algorithms of the local computation phase for array operations based on *TMR*(*n*) and *EKMR*(*n*), we analyze the theoretical performance in two aspects, the cost of addition/subtraction/multiplication operators and the cache effect. For the cost of addition/subtraction/multiplication operators, we analyze the numbers of addition/subtraction/multiplication operators for the index computation of array elements and array operations in these algorithms. In this aspect, we use the full indexing cost for each array element to analyze the performance of algorithms based on the *TMR* and *EKMR* schemes. It is no doubt that the compiler optimization techniques do achieve incremental addressing. However, we do not consider any compiler optimization technique in the theoretical analysis. The reason is that it is difficult to analyze the effects of compiler optimization techniques since the effects of the optimization may depend on the addressing mode of a machine, the way to write the programs, the efficiency of the optimization techniques, etc. To analyze the cache effect, an algorithm called *LoopCost* that was proposed by Carr et al. [5], [36] is used to compute the costs of various loop orders of an array operation. In the algorithm, *LoopCost*(*l*) is the number of cache line accessed by the innermost loop *l*. The value of *LoopCost*(*l*) reflects the cache miss rate. The smaller the *LoopCost*(*l*), the smaller the cache miss rate. The detail theoretical analysis can be found in [35]. In the following,

```
Algorithm 2D mesh_distribution_scheme_TMR(3)
    1.  for ( p_id = 0 ; p_id < P ; p_id ++)
    2.      offset = p_id × row_size ;
    3.    for ( q_id = 0 ; q_id < Q ; q_id ++)
    4.      l = 0 ;
    5.      offset1 = q_id × column_size ;   /*Pack array elements into buffer D*/
    6.      for ( k = 0 ; k < n ; k++)
    7.        for ( i = offset ; i < offset + row_size ; i++)
    8.          for ( j = offset1 ; j < offset1 + column_size ; j++)
    9.            D[l] = A[k][i][j] ;
    10.           l++;
    11.     Distribute D[] to appropriate processor;
end_of_2D mesh_distribution_scheme_TMR(3)

Algorithm 2D mesh_distribution_scheme_EKMR(3)
    1.  for ( p_id = 0 ; p_id < P ; p_id ++)
    2.      offset = p_id × row_size ;
    3.    for ( q_id = 0 ; q_id < Q ; q_id ++)
    4.      l = 0 ;
    5.      offset1 = q_id × column_size ;   /*Pack array elements into buffer D*/
    6.      for ( i = offset ; i < offset + row_size ; i++)
    7.        for ( j = offset1 ; j < offset1 + column_size ; j++)
    8.          D[l] = A [i][j] ;
    9.          l++;
    10.     Distribute D[] to appropriate processor;
end_of_2D mesh_distribution_scheme_EKMR(3)
```

Fig. 17. Algorithms of the 2D mesh distribution scheme for *TMR*(3) and *EKMR*(3).

we summarize the costs of the index computation and the cache effect shown in [35].

Assume that thecache line size used in algorithm *LoopCost* is r and the cost for an addition/subtraction operator and a multiplication operator is β and α, respectively. Let A and B be two $m \times m \times m$ three-dimensional local arrays based on the *TMR*(3) and A' and B' are the corresponding local arrays of A and B based on *EKMR*(3) with the size of $m \times m^2$ in each processor. The costs of index computation of array elements and array operations for algorithms of the *matrix-matrix multiplication* array operation based on *TMR*(3) and *EKMR*(3) is $(10\alpha + 7\beta)m^4$ and $(4\alpha + 7\beta)m^4 + \alpha m^3 + \alpha m^2$, respectively. The *LoopCost*(*l*) for algorithms of the *matrix-matrix multiplication* array operation based on *EKMR*(3) and *TMR*(3) with the innermost loop index *J* is

$$\left(2\left\lceil\frac{m^2}{r}\right\rceil + \left\lceil\frac{m}{r}\right\rceil\right) \times m^2$$

and $(2\lceil m/r\rceil + 1) \times m^3$, respectively. The *LoopCost*(*l*) for algorithms of the *matrix-matrix multiplication* array operation based on *TMR*(3) with the innermost loop index *J* is less than that with other innermost loop order.

Let $A[m_{n-4}][m_{n-3}]\ldots[m_1][l][k][i][j]$ and

$$B[m_{n-4}][m_{n-3}]\ldots[m_1][l][k][i][j]$$

be two m^n n-dimensional local arrays and

$$A'_{(m_{n-4},m_{n-3},\ldots,m_1)}[i'][j'] \text{ and } B'_{(m_{n-4},m_{n-3},\ldots,m_1)}[i'][j']$$

are two corresponding *EKMR*(*n*) whose *EKMR*(4) has a size of m^4 in each processor. For *TMR*(*n*) and *EKMR*(*n*), the costs of index computation of array elements and array operations for algorithms of the *matrix-matrix multiplication* is

$$\left(\frac{3n^2 - 3n + 2}{2}\right)\alpha m^{n+1} + (3n-2)\beta m^{n+1}$$

and $(7\alpha + 10\beta)m^{n+1} + \alpha m^n + 2\beta m^{n-1} + \alpha m^{n-2}\alpha m^{n-3}$, respectively. The *LoopCost*(*l*) for algorithms of the *matrix-matrix multiplication* array operation based on *EKMR*(*n*) and *TMR*(*n*) with the innermost loop index *J* is

$$\left(2\left\lceil\frac{m^2}{r}\right\rceil + \left\lceil\frac{m}{r}\right\rceil\right) \times m^{n-1}$$

and $(2\lceil m/r\rceil + 1) \times m^n$, respectively. The *LoopCost*(*l*) for algorithms of *matrix-matrix multiplication* array operation based on *TMR*(*n*) with the innermost loop index *J* is less than that with other innermost loop order.

From the above results, first, we can see that the cost of index computation of array elements based on the *EKMR* scheme is less than that based on the *TMR* scheme. Second, we can see that the number of cache lines accessed by array operations based on the *EKMR* scheme is less than that based on the *TMR* scheme.

```
Algorithm 2D mesh_distribution_scheme_EKMR(n)
    1.  for ( p_id = 0 ; p_id < P ; p_id ++)
    2.      offset = p_id × row_size ;
    3.    for ( q_id = 0 ; q_id < Q ; q_id ++)
    4.      l = 0 ;
    5.      offset1 = q_id × column_size ;
    6.      for ( x = 0 ; x < n^(n-4) ; x++)
    7.        for ( i = offset ; i < offset + row_size ; i++)
    8.          for ( j = offset1 ; j < offset1 + column_size ; j++)
    9.            D[l] = A'x[i][j] ;
    10.           l++;
    11.     Distribute D[] to appropriate processor;
end_of_2D mesh_distribution_scheme_EKMR(n)
```

Fig. 18. Algorithm of the 2D mesh distribution scheme for *EKMR*(*n*).

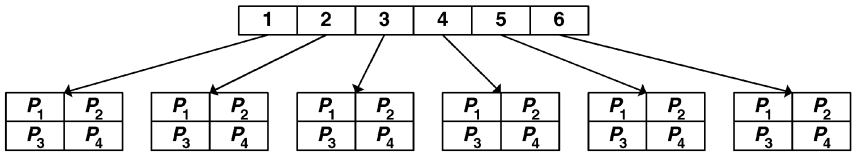

Fig. 19. The 2D mesh distribution scheme for array A' to four processors based on *EKMR*(6).

4.3 The Result Collection Phase for the *TMR* and *EKMR* Schemes

Results computed and scattered among processors must be collected to form a final result. In general, the processor (host processor) that distributes data is responsible for the results collection. To collect partial results to form a final result, the host processor uses different ways to process the partial results for different array operations. For example, for the logical array operation, such as *All*, the host processor collects the *True* or *False* information from each processor and decides the result is *True* or *False*. In this case, only logical operations are applied to the collected partial results. For some array operations, such as *matrix-matrix multiplication*, after collecting partial results from each processor, the host processor needs to unpack partial results into appropriate locations to obtain the final result. The phase to unpack partial results into appropriate array positions to obtain the final result is similar to the data distribution phase. Different array operations may have different implementations for the result collection phase. The algorithms of the result collection phase for the *matrix-matrix multiplication* array operation based on *TMR*(*n*) and *EKMR*(*n*) with the row distribution scheme is shown in Fig. 21. For the column and the 2D mesh distribution schemes, algorithms of the result collection phase for the *matrix-matrix multiplication* array operation based on *TMR*(*n*) and *EKMR*(*n*) are similar to those for the row distribution scheme. Since the page limitation, they are omitted in this paper.

5 Experimental Results

To evaluate the performance of data parallel algorithms of multidimensional array operations based on the *EKMR* scheme, we implement the *matrix-matrix addition* and *matrix-matrix multiplication* array operations based on *TMR*(3) and *EKMR*(3) with the row, the column, and the 2D mesh distribution schemes on an IBM SP2 parallel machine. The IBM SP2 parallel machineis located at National Center of High Performance Computing (NCHC) in Taiwan. This super-scalar architecture uses an IBM RISC System/6000 POWER2 CPU with a clock rate of 66.7 MHz. There are 40 IBM POWER2 nodes in this system and each node has a 128KB first-level data cache, a 32KB first-level instruction cache, and 128MB of memory space. We compare the time of the data distribution phase, the local computation phase, and the result collection phase of these data parallel algorithms. We also implemented data parallel algorithms for six Fortran 90 array intrinsic functions, *All*, *Maxval*, *Merge*, *Pack*, *Sum*, and *Cshift* in both *TMR*(3) and *EKMR*(3). These array operations based on *TMR*(4) and *EKMR*(4) with the row distribution scheme were implemented as well. All data parallel algorithms were implemented in *C + Message Passing Interface* (MPI).

5.1 Performance Comparisons for the Time of the Data Distribution Phase

Since the data distribution phase is independent to array operations, the comparisons presented in this subsection are

```
Algorithm local_computation_row_distribution_scheme_TMR(n)
1.  for ( p_id = 0 ; p_id < P ; p_id ++)
2.    for ( m_{n-4} = 0 ; m_{n-4} < n ; m_{n-4}++)
3.      for ( m_{n-3} = 0 ; m_{n-3} < n ; m_{n-3}++)
4.        /*From loop m_{n-4} to loop m_1*/
5.        for ( l = 0 ; l < n ; l++)
6.          for ( k = 0 ; k < n ; k++)
7.            for ( i = 0 ; i < row_size ; i++)
8.              for (m = 0 ; m < n ; m++)
9.                for (j = 0 ; j < n ; j++)
10.                 C[m_{n-4}][m_{n-3}]...[m_1][l][k][i][j] = C[m_{n-4}][m_{n-3}]...[m_1][l][k][i][j] +
                    A[m_{n-4}][m_{n-3}]...[m_1][l][k][l][m] × B[m_{n-4}][m_{n-3}]...[m_1][l][k][m][j];
end_of_local_computation_row_distribution_scheme_TMR(n)

Algorithm local_computation_row_distribution_scheme_EKMR(n)
1.  for ( p_id = 0 ; p_id < P ; p_id ++)
2.    for ( x = 0 ; x < n^{n-4} ; x++)
3.      for ( i = 0 ; i < row_size ; i++)
4.        t = i × n;
5.        for ( j = 0 ; j < n ; j++)
6.          v = j × n;
7.          for ( l = 0 ; l < n ; l++)
8.            w = t + l;
9.            u = l + v;
10.           for ( m = 0 ; m < n ; m++)
11.             r = m × n;
12.             for ( k = 0 ; k < n ; k++)
13.               C'_x[w][k+r]=C'_x[w][k+r]+A'_x[w][k+v]×B'_x[u][k+r];
end_of_local_computation_row_distribution_scheme_EKMR(n)
```

Fig. 20. Algorithms of the local computation phase for a *matrix-matrix multiplication* array operation based on *TMR*(*n*) and *EKMR*(*n*) with the row distribution scheme.

```
Algorithm result_collection_row_distribution_scheme_TMR(n)
1.  for ( p_id = 0 ; p_id < P ; p_id ++)
2.    Receive the return buffer D_p_id[];
3.    l=0;
4.    offset = p_id × row_size;
5.    for ( m_n-4 = 0 ; m_n-4 < n ; m_n-4++)
6.      for ( m_n-3 = 0 ; m_n-3 < n ; m_n-3++)
7.        /*From loop m_n-4 to loop m_1*/
8.        for ( l = 0 ; l < n ; l++)
9.          for ( k = 0 ; k < n ; k++)
10.           for ( i = 0 ; i < row_size ; i++)
11.             for ( j = 0 ; j < n ; j++)
12.               C[m_n-4][m_n-3]...[m_1][l][k][ i + offset ][j] = D_p_id [l] ;
13.               l++;
end_of_result_collection_row_distribution_scheme_TMR(n)

Algorithm result_collection_row_distribution_scheme_EKMR(n)
1.  for ( p_id = 0 ; p_id < P ; p_id ++)
2.    Receive the buffer D_p_id[] from each processor;
3.    l=0;
4.    offset = p_id × row_size;
5.    for (x = 0 ; x < n^(n-4) ; x++)
6.      for ( i = 0 ; i < row_size ; i++)
7.        for ( j = 0 ; j < n^2 ; j++)
8.          C_x[ i + offset ][j] = D_p_id [l];
9.          l++;
end_of_result_collection_row_distribution_scheme_EKMR(n)
```

Fig. 21. Algorithms of the result collection phase for the *matrix-matrix multiplication* array operation based on *TMR*(*n*) and *EKMR*(*n*) with the row distribution scheme.

applicable to all array operations. Fig. 22 shows the time of the data distribution phase of the row, the column, and the 2D mesh distribution schemes for *TMR*(3) and *EKMR*(3) on 16 processors. From Fig. 22, we can see that the time of the data distribution phase with different distribution schemes for *EKMR*(3) is less than that for *TMR*(3). The reason is that the numbers of noncontinuous data blocks with different distribution schemes for *EKMR*(3) are all less than those for *TMR*(3). Therefore, the packing time for *EKMR*(3) is less than that for *TMR*(3).

Fig. 23 shows the time of the data distribution phase with various distribution schemes for *TMR*(3) and *EKMR*(3) with $200 \times 200 \times 200$ array size. From Fig. 23, we have similar observations as those of Fig. 22. Fig. 24 shows the time of the data distribution phase of the row distribution scheme for *TMR*(4) and *EKMR*(4) on 16 processors. Fig. 25 shows the time of the data distribution phase of the row distribution scheme for *TMR*(4) and *EKMR*(4) with $50 \times 50 \times 50 \times 50$ array size. From Figs. 24 and 25, we can see that the time of the data distribution phase of the row distribution scheme for *EKMR*(4) is less than that for *TMR*(4).

5.2 Performance Comparisons for the Time of the Local Computation Phase

For the local computation phase of algorithms, the loop repermutation can be applied to reorder the memory accesses for array elements of array operations to obtain

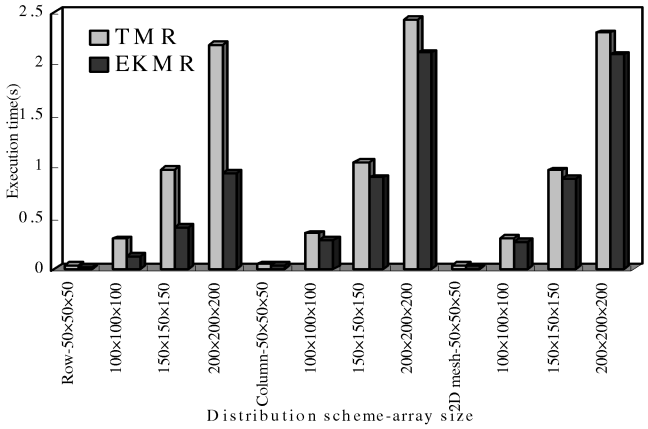

Fig. 22. The time of the data distribution phase with various distribution schemes.

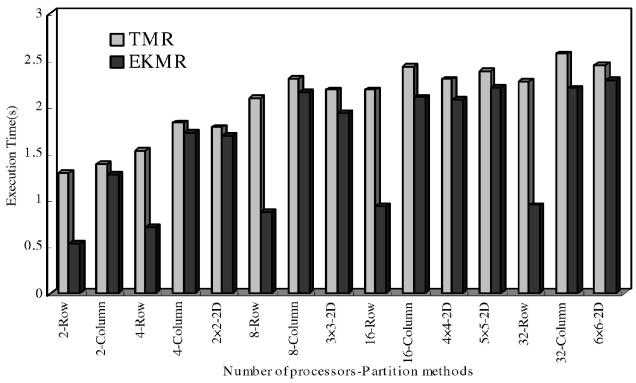

Fig. 23. The time of the data distribution phase with various distribution schemes.

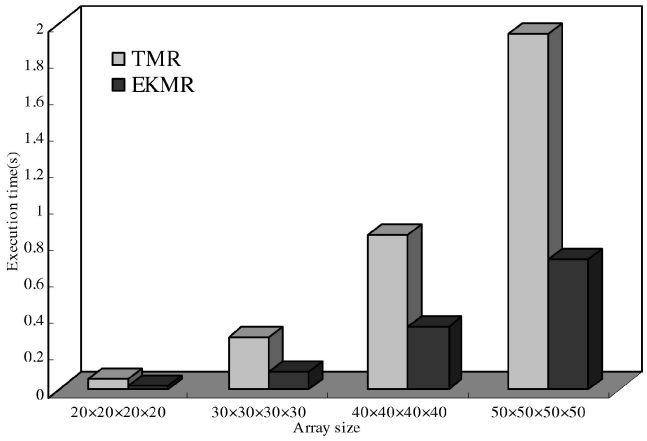

Fig. 24. The time of the data distribution phase for *TMR*(4) and *EKMR*(4).

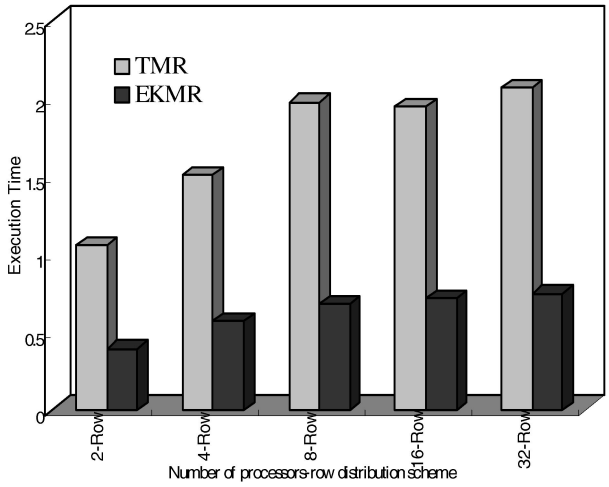

Fig. 25. The time of the data distribution phase for *TMR*(4) and *EKMR*(4).

better performance. Therefore, for the *matrix-matrix addition* and *matrix-matrix multiplication* array operations based on *TMR*(3), there are six and 24 loop orders can be used to design different algorithms, respectively. For the *matrix-matrix addition* and *matrix-matrix multiplication* array operations based on *TMR*(4), there are 24 and 120 loop orders can be used to design different algorithms. In [38], we have shown that sequential algorithms of *matrix-matrix addition* and *matrix-matrix multiplication* array operations based on *TMR*(3) and *TMR*(4) whose innermost loop index is *J* outperform those whose innermost loop index is not *J*. Therefore, in this paper, we implemented the algorithms of the local computation phase for the *matrix-matrix addition* and *matrix-matrix multiplication* array operations based on

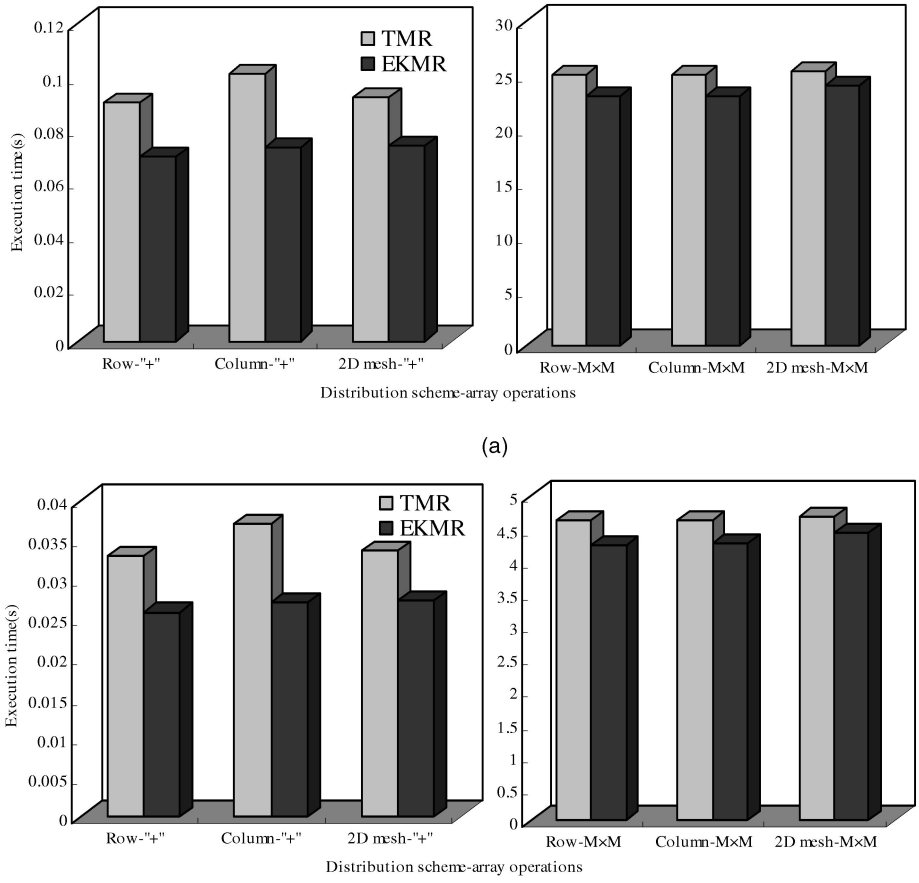

Fig. 26. The time of the local computation phase for *TMR*(3) and *EKMR*(3). (a) Without the compiler optimization. (b) With the compiler optimization.

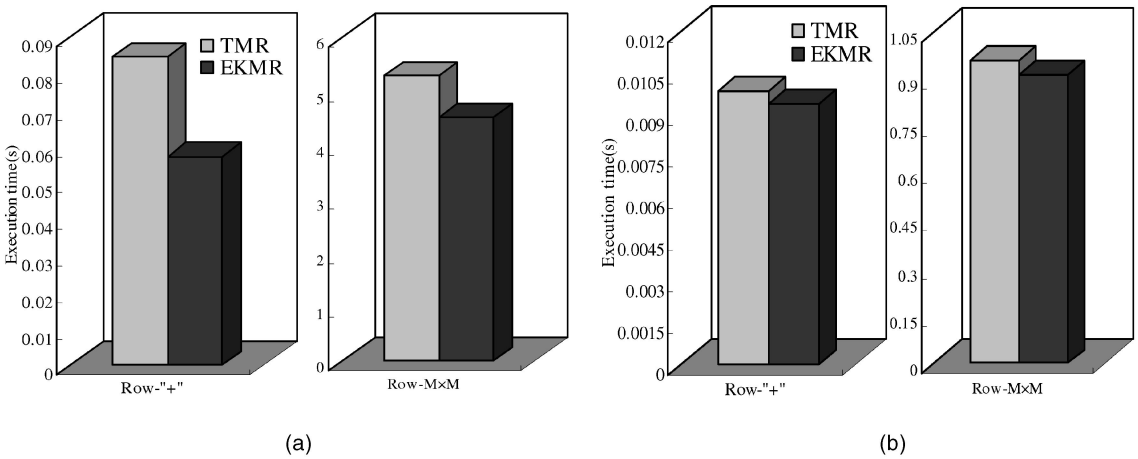

Fig. 27. The time of the local computation phase for *TMR*(4) and *EKMR*(4). (a) Without the compiler optimization. (b) With the compiler optimization.

TMR(3) with *KIJ* and *KIMJ* loop orders and *TMR*(4) with *LKIJ* and *LKIMJ* loop orders, respectively.

Fig. 26 shows the time of the local computation phase for the *matrix-matrix addition* and *matrix-matrix multiplication* array operations with/without the -O3 compiler optimization option based on the *TMR*(3) and the *EKMR*(3) with $200 \times 200 \times 200$ array size on 16 processors. From Fig. 26, with the same distribution scheme, we can see that the time of the local computation phase for data parallel algorithms based on *EKMR*(3) is less than that of based on *TMR*(3). The reason is that the cost of index computation of array elements and the number of cache lines accessed for array operations in the *EKMR* scheme are less than those in the *TMR* scheme.

Fig. 27 shows the time of the local computation phase for the *matrix-matrix addition* and *matrix-matrix multiplication* array operations with/without the -O3 compiler optimization option based on *TMR*(4) and *EKMR*(4) with the row distribution scheme and $50 \times 50 \times 50 \times 50$ array size on 16 processors. From Fig. 27, we obtain similar observations as those of Fig. 26.

5.3 Performance Comparisons for the Time of the Result Collection Phase

Fig. 28 shows the time for the result collection phase of the *matrix-matrix addition* and *matrix-matrix multiplication* array operations based on*TMR*(3) and*EKMR*(3) with $200 \times 200 \times 200$ array size on 16 processors. From Fig. 28, with the same distribution scheme, we can see that the time of the result collection phase for data parallel algorithms based on *EKMR*(3) is less than that of based on *TMR*(3). The reason is that the unpacking time for *EKMR*(3) is less than that for *TMR*(3).

Fig. 29 shows the time for the result collection phase of the *matrix-matrix addition* and *matrix-matrix multiplication* array operations based on *TMR*(4) and *EKMR*(4) with the row distribution scheme and $50 \times 50 \times 50 \times 50$ array size on 16 processors. From Fig. 29, we have similar observations as those of Fig. 28.

We also compare the speedups of data parallel algorithms of the *matrix-matrix multiplication* array operation based on the *TMR* and the *EKMR* schemes. Fig. 30 shows the speedups of data parallel algorithms of the *matrix-matrix multiplication* array operation based on *TMR*(3) and *EKMR*(3) with $200 \times 200 \times 200$ array size. From Fig. 30, we can see that the speedups of data parallel algorithms based on *EKMR*(3) are better than those based on *TMR*(3).

Fig. 31 shows the speedups of data parallel algorithms of *matrix-matrix multiplication* array operation based on *TMR*(4) and *EKMR*(4) with the row distribution scheme and $50 \times 50 \times 50 \times 50$ array size. From Fig. 31, we have similar observations as those of Fig. 30.

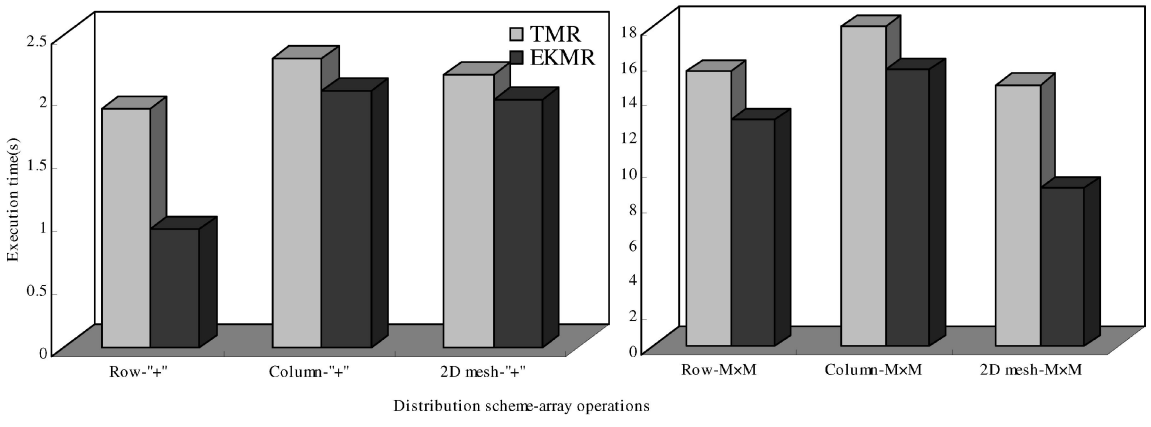

Fig. 28. The time of the result collection phase for *TMR*(3) and *EKMR*(3).

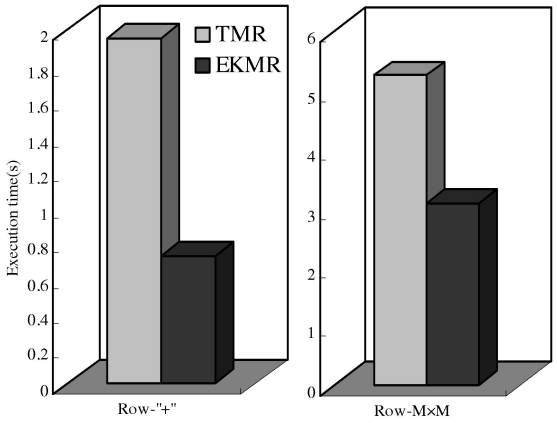

Fig. 29. The time of the result collection phase for *TMR*(4) and *EKMR*(4).

5.4 Performance Comparison for the Total Execution Time of Data Parallel Algorithms of Fortran 90 Array Intrinsic Functions

In this section, we compare the total execution time of data parallel algorithms for six Fortran 90 array intrinsic functions, *All*, *Maxval*, *Merge*, *Pack*, *Sum*, and *Cshift* in both *TMR* and *EKMR* schemes. Since there are several parameters in those array intrinsic functions, in this paper, we implemented the general case of each array intrinsic function. For the *All* array intrinsic function, we implemented the case $All(A > d)$. This operation is used to identify whether the values of array elements in A are all larger than a constant d or not. For the *Maxval* array intrinsic function, we implemented the case $e = Maxval(A)$. This operation is used to find the maximal value of array elements in A. For the *Merge* array intrinsic function, we implemented the case $C = Merge(A, B, A > B)$. This operation is used to identify those array elements in A whose values are larger than the corresponding array elements in B. For the *Pack* array intrinsic function, we implemented the case $C = Pack(A, A > d)$. This operation is used to identify array elements in A whose values are larger than a constant d. For the *Sum* array intrinsic function, we implemented the case $e = Sum(A)$. This operation is used to sum up the values of array elements in A. For the *Cshift* array intrinsic function, we implemented the case $B = Cshift(A, 2)$. This operation is used to left-shift array A two times along the column direction and stores the results to array B.

Fig. 32 shows the total execution time for data parallel algorithms of six Fortran 90 array intrinsic functions based on *TMR*(3) and *EKMR*(3) with $200 \times 200 \times 200$ array size on 16 processors. From Fig. 32, we can see that the execution time for data parallel algorithms of six Fortran 90 array intrinsic functions based on *EKMR*(3) is less than that based on *TMR*(3) for all test cases.

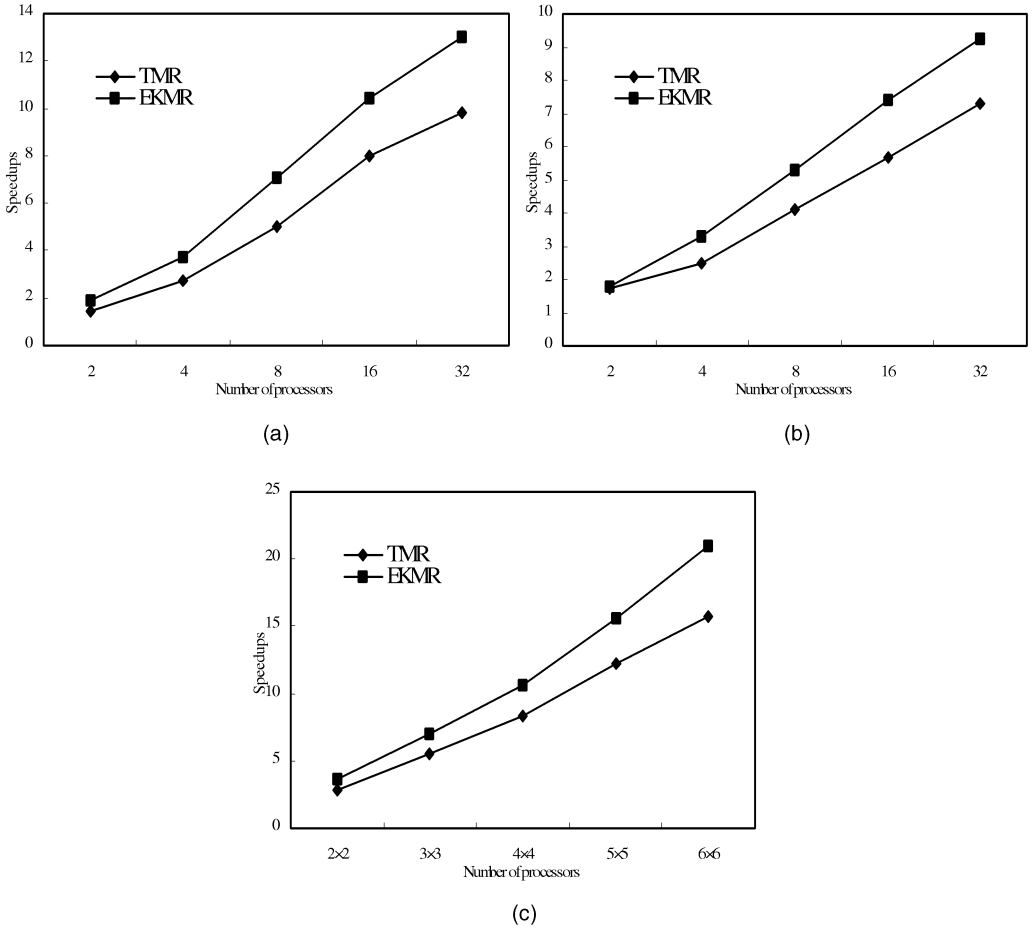

Fig. 30. The speedups of the *matrix-matrix multiplication* array operation based on *TMR*(3) and *EKMR*(3). (a) The row distribution scheme. (b) The column distribution scheme. (c) The 2D mesh distribution scheme.

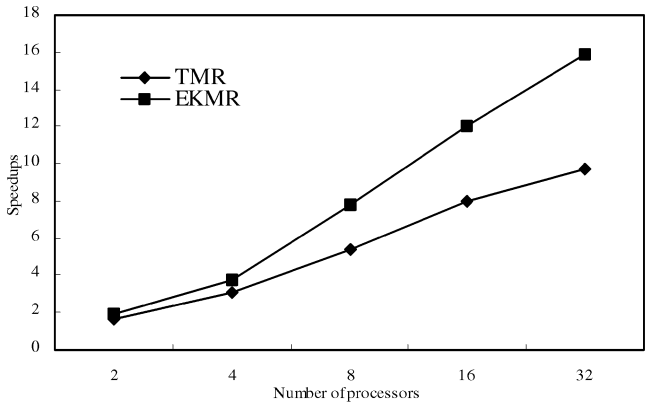

Fig. 31. The speedups of the *matrix-matrix multiplication* array operation based on *TMR*(4) and *EKMR*(4).

Fig. 33 shows the execution time for data parallel algorithms of Fortran 90 array intrinsic functions based on *TMR*(4) and *EKMR*(4) with the row distribution scheme and $50 \times 50 \times 50 \times 50$ array size on 16 processors. From Fig. 33, we have similar observations as those of Fig. 32.

6 Conclusions and Future Work

In this paper, based on the row, the column, and the 2D mesh distribution schemes, we have designed data parallel algorithms for the *matrix-matrix addition* and *matrix-matrix multiplication* array operations based on the *EKMR* scheme for multidimensional arrays on distributed memory multicomputers. We also presented data parallel algorithms for six Fortran 90 array intrinsic functions, *All*, *Maxval*, *Merge*, *Pack*, *Sum*, and *Cshift*, based on the *EKMR* scheme. To evaluate these algorithms, we compared the time of the data distribution, the local computation, and the result collection phases of these array operations based on the *EKMR* scheme with those based on the *TMR* scheme. The experimental results show that the execution time of array operations based on the *EKMR* scheme is less than that based on the *TMR* scheme in the data distribution, the local computation, and the result collection phases for all test cases. The results encourage us to use the *EKMR* scheme for multidimensional array representation on distributed memory multicomputers.

In this paper, we used the loop repermutation and the concepts proposed by O'Boyle et al. [37] to design algorithms for multidimensional array operations. All programs of array operations based on the *TMR* and *EKMR* schemes are derived by hand. However, Kennedy et al. [4], [15], [27], Ancourt et al. [2], [43], and Kodukula et al. [28] have proposed some automated methods to generate efficient parallel codes for two-dimensional array operations based on the *TMR* scheme. It is interesting to see if their methods can be applied to multidimensional array operations. In the future, we will try to extend their work to multidimensional array operations based on the *TMR* and *EKMR* schemes.

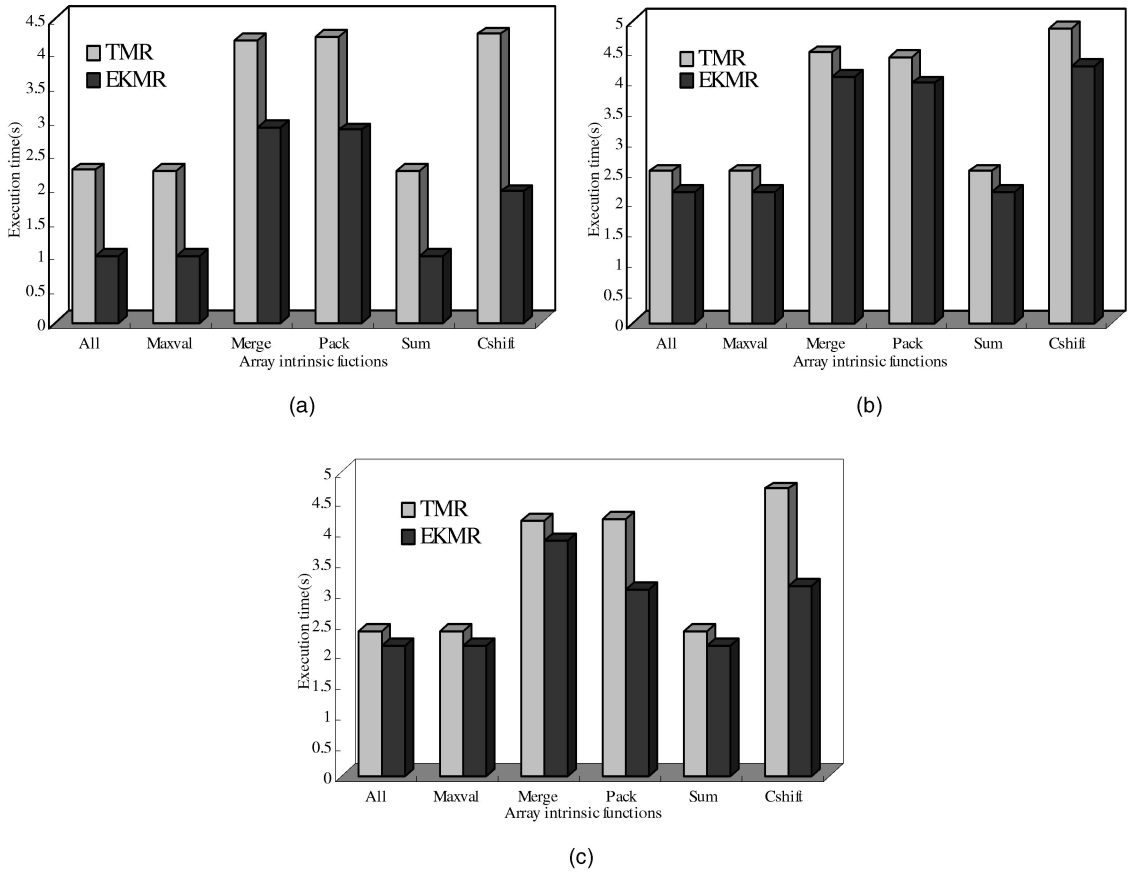

Fig. 32. The execution time for six Fortran 90 array intrinsic functions based on the *TMR*(3) and the *EKMR*(3). (a) Row distribution scheme. (b) Column distribution scheme. (c) Two-dimension mesh distribution scheme.

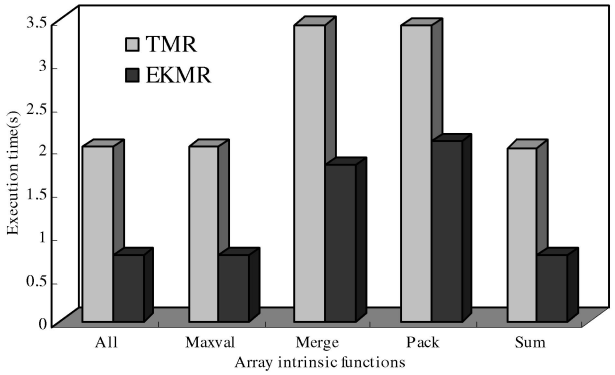

Fig. 33. The execution time for six Fortran 90 array intrinsic functions based on *TMR*(4) and *EKMR*(4).

ACKNOWLEDGMENTS

The work of this paper was partially supported by the National Science Council under contract NSC89-2213-E-035-007.

REFERENCES

[1] J.C. Adams, W.S. Brainerd, J.T. Martin, B.T. Smith, and J.L. Wagener, *FORTRAN 90 Handbooks.* Intertext Publications/McGraw-Hill, 1992.

[2] C. Ancourt and F. Irigoin, "Scanning Polyhedra with DO Loops," *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* pp. 39-50, 1991.

[3] I. Banicescu and S.F. Hummel, "Balancing Processor Loads and Exploiting Data Locality in N-Body Simulations," *Proc. ACM/IEEE Supercomputing Conf.,* 1995.

[4] D. Callahan, S. Carr, and K. Kennedy, "Improving Register Allocation for Subscripted Variables," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 53-65, 1990.

[5] S. Carr, K.S. McKinley, and C.-W. Tseng, "Compiler Optimizations for Improving Data Locality," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 252-262, 1994.

[6] L. Carter, J. Ferrante, and S.F. Hummel, "Hierarchical Tiling for Improved Superscalar Performance," *Proc. Int'l Symp. Parallel Processing,* pp. 239-245, 1995.

[7] R.-G. Chang, T.-R. Chung, and J.K. Lee, "Parallel Sparse Supports for Array Intrinsic Functions of Fortran 90," *J. Supercomputing,* vol. 18, no. 3, pp. 305-339, Mar. 2001.

[8] S. Chatterjee, A.R. Lebeck, P.K. Patnala, and M. Thottethodi, "Recursive Array Layouts and Fast Parallel Matrix Multiplication," *Proc. ACM Symp. Parallel Algorithms and Architectures,* pp. 222-231, 1999.

[9] S. Chatterjee, V.V. Jain, A.R. Lebeck, S. Mundhra, and M. Thottethodi, "Nonlinear Array Layouts for Hierarchical Memory Systems," *Proc. ACM Int'l Conf. Supercomputing,* pp. 444-453, 1999.

[10] T.-R. Chung, R.-G. Chang, and J.K. Lee, "Sampling and Analytical Techniques for Data Distribution of Parallel Sparse Computation," *Proc. SIAM Conf. Parallel Processing for Scientific Computing,* 1997.

[11] T.-R. Chung, R.-G. Chang, and J.K. Lee, "Efficient Support of Parallel Sparse Computation for Array Intrinsic Functions of Fortran 90," *Proc. ACM Int'l Conf. Supercomputing,* pp. 45-52, 1998.

[12] M. Cierniak and W. Li, "Unifying Data and Control Transformations for Distributed Shared Memory Machines," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 205-217, 1995.

[13] S. Coleman and K.S. McKinley, "Tile Size Selection Using Cache Organization and Data Layout," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 279-290, 1995.

[14] J.K. Cullum and R.A. Willoughby, *Lanczos Algorithms for Large Symmetric Eignenvalue Computations.* Boston: Birkhauser, 1985.

[15] C. Ding and K. Kennedy, "Improving Cache Performance in Dynamic Applications through Data and Computation Reorganization at Run Time," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 229-241, 1999.

[16] C.H.Q. Ding, "An Optimal Index Reshuffle Algorithm for Multidimensional Arrays and Its Applications for Parallel Architectures," *IEEE Trans. Parallel and Distributed Systems,* vol. 12, no. 3, pp. 306-315, Mar. 2001.

[17] B.B. Fraguela, R. Doallo, and E.L. Zapata, "Cache Misses Prediction for High Performance Sparse Algorithms," *Proc. Int'l Euro-Par Conf.,* pp. 224-233, 1998.

[18] B.B. Fraguela, R. Doallo, and E.L. Zapata, "Cache Probabilistic Modeling for Basic Sparse Algebra Kernels Involving Matrices with a Non-Uniform Distribution," *Proc. IEEE Euro-Micro Conf.,* pp. 345-348, 1998.

[19] B.B. Fraguela, R. Doallo, and E.L. Zapata, "Modeling Set Associative Caches Behaviour for Irregular Computations," *Proc. ACM Int'l Conf. Measurement and Modeling of Computer Systems,* pp. 192-201, 1998.

[20] B.B. Fraguela, R. Doallo, and E.L. Zapata, "Automatic Analytical Modeling for the Estimation of Cache Misses," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* pp. 221-231, 1999.

[21] J.D. Frens and D.S. Wise, "Auto-Blocking Matrix-Multiplication or Tracking BLAS3 Performance from Source Code," *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* pp. 206-216, 1997.

[22] G.H. Golub and C.F. Van Loan, *Matrix Computations,* second ed. Baltimore, Md.: John Hopkins Univ. Press, 1989.

[23] High Performance Fortran Forum, *High Performance Fortran Language Specification,* second ed. Rice Univ., 1997.

[24] M. Kandemir, J. Ramanujam, and A. Choudhary, "Improving Cache Locality by a Combination of Loop and Data Transformations," *IEEE Trans. Computers,* vol. 48, no. 2, pp. 159-167, Feb. 1999.

[25] M. Kandemir, J. Ramanujam, and A. Choudhary, "A Compiler Algorithm for Optimizing Locality in Loop Nests," *Proc. ACM Int'l Conf. Supercomputing,* pp. 269-276, 1997.

[26] C.W. Kebler and C.H. Smith, "The SPARAMAT Approach to Automatic Comprehension of Sparse Matrix Computations," *Proc. Int'l Workshop Program Comprehension,* pp. 200-207, 1999.

[27] K. Kennedy and K.S. McKinley, "Optimizing for Parallelism and Data Locality," *Proc. ACM Int'l Conf. Supercomputing,* pp. 323-334, 1992.

[28] I. Kodukula, N. Ahmed, and K. Pingali, "Data-Centric Multilevel Blocking," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 346-357, 1997.

[29] V. Kotlyar, K. Pingali, and P. Stodghill, "Compiling Parallel Sparse Code for User-Defined Data Structures," *Proc. SIAM Conf. Parallel Processing for Scientific Computing,* 1997.

[30] V. Kotlyar, K. Pingali, and P. Stodghill, "A Relation Approach to the Compilation of Sparse Matrix Programs," *Proc. European Conf. Parallel Processing,* pp. 318-327, 1997.

[31] V. Kotlyar, K. Pingali, and P. Stodghill, "Compiling Parallel Code for Sparse Matrix Applications," *Proc. Supercomputing Conf.,* pp. 20-38, Aug. 1997.

[32] B. Kumar, C.-H. Huang, R.W. Johnson, and P. Sadayappan, "A Tensor Product Formulation of Strassen's Matrix Multiplication Algorithm with Memory Reduction," *Proc. Int'l Parallel Processing Symp.,* pp. 582-588, 1993.

[33] M.S. Lam, E.E. Rothberg, and M.E. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 63-74, 1991.

[34] W. Li and K. Pingali, "A Singular Loop Transformation Framework Based on Non-Singular Matrices," *Proc. Workshop Languages and Compilers for Parallel Computing,* pp. 391-405, 1992.

[35] C.-Y. Lin, J.-S. Liu, and Y.-C. Chung, "Efficient Representation Scheme for Multi-Dimensional Array Operations," *IEEE Trans. Computers,* vol. 51, no. 3, pp. 327-345, Mar. 2002.

[36] K.S. McKinley, S. Carr, and C.-W. Tseng, "Improving Data Locality with Loop Transformations," *ACM Trans. Programming Languages and Systems,* vol. 18, no. 4, pp. 424-453, July 1996.

[37] M.F.P. O'Boyle and P.M.W. Knijnenburg, "Integrating Loop and Data Transformations for Global Optimization," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* pp. 12-19, 1998.

[38] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, *Numerical Recipes in Fortran 90: The Art of Parallel Scientific Computing.* Cambridge Univ. Press, 1996.

[39] P.D. Sulatycke and K. Ghose, "Caching Efficient Multithreaded Fast Multiplication of Sparse Matrices," *Proc. Merged Int'l Parallel Processing Symp. and Symp. Parallel and Distributed Processing,* pp. 117-124, 1998.

[40] M. Thottethodi, S. Chatterjee, and A.R. Lebeck, "Tuning Strassen's Matrix Multiplication for Memory Efficiency," *Proc. SC: of High Performance Networking and Computing,* 1998.
[41] M. Ujaldon, E.L. Zapata, S.D. Sharma, and J. Saltz, "Parallelization Techniques for Sparse Matrix Applications," *J. Parallel and Distribution Computing,* vol. 38, no. 2, pp. 256-266, 1996.
[42] M. Wolf and M. Lam, "A Data Locality Optimizing Algorithm," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 30-44, 1991.
[43] Y.Q. Yang, C. Ancourt, and F. Irigoin, "Minimal Data Dependence Abstractions for Loop Transformations," *Proc. Workshop Languages and Compilers for Parallel Computing,* pp. 201-216, 1994.
[44] L.H. Ziantz, C.C. Ozturan, and B.K. Szymanski, "Run-Time Optimization of Sparse Matrix-Vector Multiplication on SIMD Machines," *Proc. Int'l Conf. Parallel Architectures and Languages,* pp. 313-322, 1994.

Chun-Yuan Lin received the BS and MS degrees in computer science from Feng Chia University in 1999 and 2000, respectively. He is currently a PhD student in the Department of Information Engineering at Feng Chia University. His research interests are in the areas of parallel and distributed computing, parallel algorithms, array operations, and bioinformatics. He is a student member of the IEEE Computer Society and ACM.

Yeh-Ching Chung received the BS degree in information engineering from Chung Yuan Christian University in 1983, and the MS and PhD degrees in computer and information science from Syracuse University in 1988 and 1992, respectively. He joined the Department of Information Engineering at Feng Chia University as an associate professor in 1992 and became a full professor in 1999. From 1998 to 2001, he was the chairman of the department. In 2002, he joined the Department of Computer Science at National Tsing Hua University as a full professor. His research interests include parallel and distributed processing, pervasive computing, embedded software, and system software for SOC design. He is a member of the IEEE Computer Society and ACM.

Jen-Shiuh Liu received the BS and MS degrees in nuclear engineering from National Tsing Hua University and the MS and PhD degrees in computer science from Michigan State University in 1979, 1981, 1987, and 1992, respectively. Since 1992, he has been an associate professor in the Department of Information Engineering and Computer Science at Feng Chia University, Taiwan. His research interests include parallel and distributed processing, computer networks, and computer system security.