Bill Pugh
Chau-Wen Tseng (Eds.)

LNCS 2481

Languages and Compilers for Parallel Computing

15th Workshop, LCPC 2002
College Park, MD, USA, July 2002
Revised Papers

Springer

Bill Pugh Chau-Wen Tseng (Eds.)

Languages and Compilers for Parallel Computing

15th Workshop, LCPC 2002
College Park, MD, USA, July 25-27, 2002
Revised Papers

Springer

Volume Editors

Bill Pugh
Chau-Wen Tseng
University of Maryland, Department of Computer Science
College Park, MD 20814, USA
E-mail: {pugh, tseng}@cs.umd.edu

Library of Congress Control Number: 2005937164

CR Subject Classification (1998): D.3, D.1.3, F.1.2, B.2.1, C.2.4, C.2, E.1

ISSN	0302-9743
ISBN-10	3-540-30781-8 Springer Berlin Heidelberg New York
ISBN-13	978-3-540-30781-5 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11596110 06/3142 5 4 3 2 1 0

Lecture Notes in Computer Science 2481

Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Preface

The 15th Workshop on Languages and Compilers for Parallel Computing was held in July 2002 at the University of Maryland, College Park. It was jointly sponsored by the Department of Computer Science at the University of Maryland and the University of Maryland Institute for Advanced Computer Studies (UMIACS). LCPC 2002 brought together over 60 researchers from academia and research institutions from many countries.

The program of 26 papers was selected from 32 submissions. Each paper was reviewed by at least three Program Committee members and sometimes by additional reviewers. Prior to the workshop, revised versions of accepted papers were informally published on the workshop's website and in a paper proceedings that was distributed at the meeting. This year, the workshop was organized into sessions of papers on related topics, and each session consisted of two to three 30-minute presentations. Based on feedback from the workshop, the papers were revised and submitted for inclusion in the formal proceedings published in this volume. Two papers were presented at the workshop but later withdrawn from the final proceedings by their authors.

We were very lucky to have Bill Carlson from the Department of Defense give the LCPC 2002 keynote speech on "UPC: A C Language for Shared Memory Parallel Programming." Bill gave an excellent overview of the features and programming model of the UPC parallel programming language.

LCPC workshop presentations were held on campus in a spacious 140-person auditorium in the newly constructed Computer Science Instructional Center (CSIC). Workshop participants also enjoyed an afternoon excursion downtown to the Smithsonian's National Museum of Natural History, followed by a banquet held in the wine room of the D.C. Coast restaurant.

The success of LCPC 2002 was due to many people. We would like to thank the Program Committee members for their timely and thorough reviews, and the LCPC Steering Committee (especially David Padua) for providing invaluable advice and continuity for LCPC. We wish to thank Lawrence Rauchwerger and Silvius Rus for providing scripts and templates for formatting the proceedings. We appreciate the hard work performed by Cecilia Khullman, Christina Beal, and Johanna Weinstein (from UMIACS) handling local arrangements and workshop registration. Finally, we would like to thank all the LCPC 2002 authors for their patience in waiting for the long overdue publication of the formal workshop proceedings.

July 2005

Bill Pugh
Chau-Wen Tseng

Organization

The 15th Workshop on Languages and Compilers for Parallel Computing was hosted by the Department of Computer Science at the University of Maryland and the University of Maryland Institute for Advanced Computer Studies (UMIACS).

Steering Committee

Utpal Banerjee	Intel Corporation
David Gelernter	Yale University
Alex Nicolau	University of California at Irvine
David Padua	University of Illinois at Urbana-Champaign

General and Program Co-chairs

Bill Pugh	University of Maryland
Chau-Wen Tseng	University of Maryland

Program Committee

Hank Dietz	University of Kentucky
Manish Gupta	IBM T.J. Watson Research Center
Sam Midkiff	Purdue University
Jose Moreira	IBM T.J. Watson Research Center
Dave Padua	University of Illinois at Urbana-Champaign
Bill Pugh	University of Maryland
Lawrence Rauchwerger	Texas A&M University
Chau-Wen Tseng	University of Maryland

Table of Contents

Memory-Constrained Communication Minimization for a Class of Array Computations

Daniel Cociorva[1], Gerald Baumgartner[1], Chi-Chung Lam[1], P. Sadayappan[1], and J. Ramanujam[2]

[1] Department of Computer and Information Science, The Ohio State University, Columbus, OH 43210, USA
{cociorva, gb, clam, saday}@cis.ohio-state.edu
[2] Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803, USA
jxr@ece.lsu.edu

Abstract. The accurate modeling of the electronic structure of atoms and molecules involves computationally intensive tensor contractions involving large multidimensional arrays. The efficient computation of complex tensor contractions usually requires the generation of temporary intermediate arrays. These intermediates could be extremely large, but they can often be generated and used in batches through appropriate loop fusion transformations. To optimize the performance of such computations on parallel computers, the total amount of interprocessor communication must be minimized, subject to the available memory on each processor. In this paper, we address the memory-constrained communication minimization problem in the context of this class of computations. Based on a framework that models the relationship between loop fusion and memory usage, we develop an approach to identify the best combination of loop fusion and data partitioning that minimizes inter-processor communication cost without exceeding the per-processor memory limit. The effectiveness of the developed optimization approach is demonstrated on a computation representative of a component used in quantum chemistry suites.

1 Introduction

The development of high-performance parallel programs for scientific applications is usually very time consuming. The time to develop an efficient parallel program for a computational model can be a primary limiting factor in the rate of progress of the science. Our overall goal is to develop a program synthesis system to facilitate the rapid development of high-performance parallel programs for a class of scientific computations encountered in quantum chemistry. The domain of our focus is electronic structure calculations, as exemplified by coupled cluster methods [4], in which many computationally intensive components are expressible as a set of tensor contractions. We are developing a synthesis system that will transform an input specification expressed in a high-level notation into efficient parallel code tailored to the characteristics of the target architecture.

A number of compile-time optimizations are being incorporated into the program synthesis system. These include algebraic transformations to minimize the number

B. Pugh and C.-W. Tseng (Eds.): LCPC 2002, LNCS 2481, pp. 1–15, 2005.

of arithmetic operations [8,13], loop fusion and array contraction for memory space minimization [13,12], tiling and data locality optimization [1,2], space-time trade-off optimization [3], and data partitioning for communication minimization [9,10]. Since the problem of determining the set of algebraic transformations to minimize operation count was found to be NP-complete, we developed a pruning search procedure [8] that is very efficient in practice. The operation-minimization procedure results in the creation of intermediate temporary arrays. Often, these intermediate arrays that help in reducing the computational cost create a problem with the memory required. Loop fusion was found to be effective in significantly reducing the total memory requirement. However, since some fusions could prevent other fusions, the choice of the optimal set of fusion transformations is important. So we addressed the problem of finding the choice of fusions for a given operator tree that minimizes the space required for all intermediate arrays after fusion [12,11].

We have also previously addressed the problem of communication optimization in the context of the operator trees [9,10]. An efficient polynomial-time dynamic programming algorithm was developed for the determination of optimal distributions of the various arrays through the evaluation of the operator tree so as to minimize interprocessor communication overhead. However, that model did not consider the effects of loop fusion for memory minimization. As we elaborate later with examples, it is not feasible to simply apply the previously developed loop fusion algorithm and the previous communication minimization algorithm (in either order) to optimize for the parallel context when memory size constraints are severe. For many computations of interest to quantum chemists, the unoptimized form of the computation could require in excess of hundreds of terabytes of memory. Therefore, the following optimization problem is of great interest: given a set of computations expressed as a sequence of tensor contractions (explained later on), an empirically derived measure of the communication cost for a given target computer, and a specified limit on the amount of available memory on each processor, re-structure the computation so as to minimize the total execution time while staying within the available memory. In this paper, we present a framework that we have developed to address this problem. The memory-constrained communication minimization algorithm we develop here will be incorporated into the synthesis system being developed.

The computational structures that we target arise in scientific application domains that are extremely compute-intensive and consume significant computer resources at national supercomputer centers. They are present in various computational chemistry codes such as ACES II, GAMESS, Gaussian, NWChem, PSI, and MOLPRO. In particular, they comprise the bulk of the computation with the coupled cluster approach to the accurate description of the electronic structure of atoms and molecules [14,15]. Computational approaches to modeling the structure and interactions of molecules, the electronic and optical properties of molecules, the heats and rates of chemical reactions, etc., are very important to the understanding of chemical processes in real-world systems.

There has been some recent work on using loop fusion for memory reduction for sequential execution. Fraboulet et al. [5] use loop alignment to reduce memory requirement between adjacent loops by formulating the one-dimensional version of the prob-

lem as a network flow problem; they did look at the effect of their solution on cache behavior or communication. Song et al. [17,18] present a different network flow formulation of the memory reduction problem and they include a simple model of cache misses as well. They do not consider trading off memory for recomputation or the impact of data distribution on communication costs while meeting per-processor memory constraints in a distributed memory machine. There has been much less work investigating the use of loop fusion as a means of reducing memory requirements [6,16]. To the best of our knowledge, loop fusion transformation for memory reduction, in combination with data partitioning for communication minimization in the parallel context, has not been previously considered.

The paper is organized as follows. In the next section, we elaborate on the computational context of interest and the pertinent optimization issues. Section 3 presents our multi-dimensional processor model, discusses the interaction between distribution of arrays and loop fusion, and describes our algorithm for the memory-constrained communication minimization problem. Section 4 presents results from the application of the new algorithm to an example abstracted from NWChem [7]. Conclusions are provided in Section 5.

2 Elaboration of Problem

In the class of computations considered, the final result to be computed can be expressed as multi-dimensional summations of the product of several input arrays. Due to commutativity, associativity, and distributivity, there are many different ways to obtain the same final result and they could differ widely in the number of floating point operations required. Consider the following example:

$$S(t) = \sum_{i,j,k} A(i,j,t) \times B(j,k,t).$$

If implemented directly as expressed above, the computation would require $2N_iN_jN_kN_t$ arithmetic operations to compute. However, assuming associative reordering of the operations and use of distributive law of multiplication over addition is acceptable for the floating-point computations, the above computation can be rewritten in various ways. One equivalent form that only requires $N_iN_jN_t + N_jN_kN_t + 2N_jN_t$ operations is as shown in Fig. 1(a).

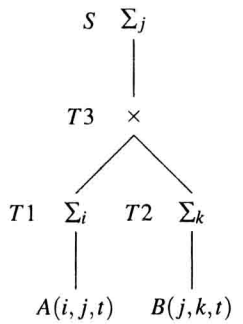
Generalizing from the above example, we can express multi-dimensional integrals of products of several input arrays as a sequence of formulae. Each formula produces some intermediate array and the last formula gives the final result. A formula is either:

- a multiplication formula of the form: $Tr(\ldots) = X(\ldots) \times Y(\ldots)$, or
- a summation formula of the form: $Tr(\ldots) = \sum_i X(\ldots)$,

where the terms on the right hand side represent input arrays or intermediate arrays produced by a previously defined formula. Let IX, IY and ITr be the sets of indices in $X(\ldots)$, $Y(\ldots)$ and $Tr(\ldots)$, respectively. For a formula to be well-formed, every index in $X(\ldots)$ and $Y(\ldots)$, except the summation index in the second form, must appear in $Tr(\ldots)$. Thus $IX \cup IY \subseteq ITr$ for any multiplication formula, and $IX - \{i\} \subseteq ITr$ for any

$$T1(j,t) = \sum_i A(i,j,t)$$
$$T2(j,t) = \sum_k B(j,k,t)$$
$$T3(j,t) = T1(j,t) \times T2(j,t)$$
$$S(t) = \sum_j T3(j,t)$$

(a) Formula sequence



(b) Binary tree representation

Fig. 1. A formula sequence and its binary tree representation

summation formula. Such a sequence of formulae fully specifies the multiplications and additions to be performed in computing the final result.

A sequence of formulae can be represented graphically as a binary tree to show the hierarchical structure of the computation more clearly. In the binary tree, the leaves are the input arrays and each internal node corresponds to a formula, with the last formula at the root. An internal node may either be a multiplication node or a summation node. A multiplication node corresponds to a multiplication formula and has two children which are the terms being multiplied together. A summation node corresponds to a summation formula and has only one child, representing the term on which summation is performed. As an example, the binary tree in Fig. 1(b) represents the formula sequence shown in Fig. 1(a).

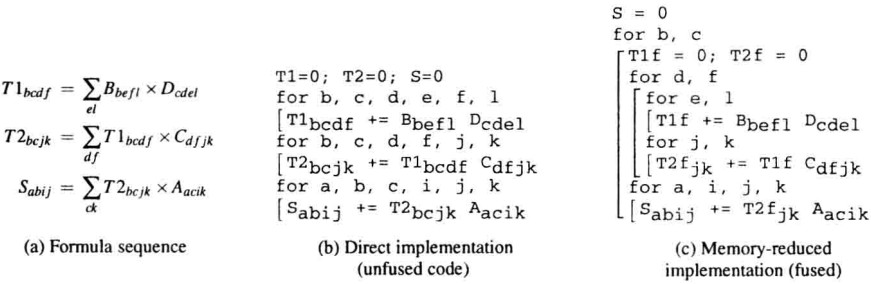
The operation-minimization procedure discussed above usually results in the creation of intermediate temporary arrays. Sometimes these intermediate arrays that help in reducing the computational cost create a problem with the memory capacity required. For example, consider the following expression:

$$S_{abij} = \sum_{cdefkl} A_{acik} \times B_{befl} \times C_{dfjk} \times D_{cdel}$$

If this expression is directly translated to code (with ten nested loops, for indices $a-l$), the total number of arithmetic operations required will be $4N^{10}$ if the range of each index $a-l$ is N. Instead, the same expression can be rewritten by use of associative and distributive laws as the following:

$$S_{abij} = \sum_{ck} \left(\sum_{df} \left(\sum_{el} B_{befl} \times D_{cdel} \right) \times C_{dfjk} \right) \times A_{acik}$$

This corresponds to the formula sequence shown in Fig. 2(a) and can be directly translated into code as shown in Fig. 2(b). This form only requires $6N^6$ operations. However, additional space is required to store temporary arrays $T1$ and $T2$. Often, the space requirements for the temporary arrays poses a serious problem. For this example,



(a) Formula sequence

(b) Direct implementation (unfused code)

(c) Memory-reduced implementation (fused)

Fig. 2. Example illustrating use of loop fusion for memory reduction

abstracted from a quantum chemistry model, the array extents along indices $a - d$ are the largest, while the extents along indices $i - l$ are the smallest. Therefore, the size of temporary array $T1$ would dominate the total memory requirement.

We have previously shown that the problem of determining the operator tree with minimal operation count is NP-complete, and have developed a pruning search procedure [8,9] that is very efficient in practice. For the above example, although the latter form is far more economical in terms of the number of arithmetic operations, its implementation will require the use of temporary intermediate arrays to hold the partial results of the parenthesized array subexpressions. Sometimes, the sizes of intermediate arrays needed for the "operation-minimal" form are too large to even fit on disk.

A systematic way to explore ways of reducing the memory requirement for the computation is to view it in terms of potential loop fusions. Loop fusion merges loop nests with common outer loops into larger imperfectly nested loops. When one loop nest produces an intermediate array which is consumed by another loop nest, fusing the two loop nests allows the dimension corresponding to the fused loop to be eliminated in the array. This results in a smaller intermediate array and thus reduces the memory requirements. For the example considered, the application of fusion is illustrated in Fig. 2(c). By use of loop fusion, for this example it can be seen that $T1$ can actually be reduced to a scalar and $T2$ to a 2-dimensional array, without changing the number of arithmetic operations.

For a computation comprised of a number of nested loops, there will generally be a number of fusion choices, that are not all mutually compatible. This is because different fusion choices could require different loops to be made the outermost. In prior work, we have addressed the problem of finding the choice of fusions for a given operator tree that minimizes the total space required for all arrays after fusion [13,12,11].

A data-parallel implementation of the unfused code for computing S_{abij} would involve a sequence of three steps, each corresponding to one of the loops in Fig. 2(b). The communication cost incurred will clearly depend on the way the arrays A, B, C, D, $T1$, $T2$, and S are distributed. We have previously considered the problem of minimization of communication with such computations [13,9]. However, the issue of memory space requirements was not addressed. In practice, many of the computations of interest in quantum chemistry require impractically large intermediate arrays in the unfused operation-minimal form. Although the collective memory of parallel machines is

very large, it is nevertheless insufficient to hold the full intermediate arrays for many computations of interest. Thus, array contraction through loop fusion is essential in the parallel context too. However, it is not satisfactory to first find a communication-minimizing data/computation distribution for the unfused form, and then apply fusion transformations to minimize memory for that parallel form. This is because 1) fusion changes the communication cost, and 2) it may be impossible to find a fused form that fits within available memory, due to constraints imposed by the chosen data distribution on possible fusions. In this paper we address this problem of finding suitable fusion transformations and data/computation partitioning that minimize communication costs, subject to limits on available per-processor memory.

3 Memory-Constrained Communication Minimization

Given a sequence of formulae, we now address the problem of finding the optimal partitioning of arrays and operations among the processors and the loop fusions on each processor in order to minimize inter-processor communication and computational costs while staying within the available memory in implementing the computation on a message-passing parallel computer. Section 3.1 introduces a multi-dimensional processor model used to represent the computational space. Section 3.2 discusses the combined effects of loop fusions and array/operation partitioning on communication cost, computational cost, and memory usage. An integrated algorithm for solving this problem is presented in Section 3.3.

3.1 Preliminaries: A Multi-dimensional Processor Model

A logical view of the processors as a multi-dimensional grid is used, where each array can be distributed or replicated along one or more of the processor dimensions. As will be clear later on, the logical view of the processor grid does not impose any restriction on the actual physical interconnection topology of the processor system since empirical characterization of the cost of redistribution between different distributions is performed on the target system.

Let p_d be the number of processors on the d-th dimension of an n-dimensional processor array, so that the number of processors is $p_1 \times p_2 \times \ldots \times p_n$. We use an n-tuple to denote the partitioning or *distribution* of the elements of a data array on an n-dimensional processor array. The d-th position in an n-tuple α, denoted $\alpha[d]$, corresponds to the d-th processor dimension. Each position may be one of the following: an index variable distributed along that processor dimension, a '*' denoting replication of data along that processor dimension, or a '1' denoting that only the first processor along that processor dimension is assigned any data. If an index variable appears as an array subscript but not in the n-tuple, then the corresponding dimension of the array is not distributed. Conversely, if an index variable appears in the n-tuple but not in the array, then the data are replicated along the corresponding processor dimension, which is the same as replacing that index variable with a '*'.

As an example, suppose 128 processors form a 4-dimensional $2 \times 2 \times 4 \times 8$ array. For the array $B(b,e,f,l)$ in Fig. 2(a), the 4-tuple $\langle b,e,*,1\rangle$ specifies that the first and the

second dimensions of B are distributed along the first and second processor dimensions respectively (the third and fourth dimensions of B are not distributed), and that data are replicated along the third processor dimension and are assigned only to processors whose fourth processor dimension equals 1. Thus, a processor whose id is P_{z_1,z_2,z_3,z_4} will be assigned a portion of B specified by $B(myrange(z_1,N_b,p_1),myrange(z_2,N_e,p_2),1:N_f,1:N_l)$ if $z_4 = 1$ and no part of B otherwise, where $myrange(z,N,p)$ is the range $(z-1)\times N/p+1$ to $z\times N/p$.

We assume the data-parallel programming model and do not consider distributing the computation of different formulae on different subsets of processors. A child array (or a part of it) is redistributed before the evaluation of its parent if their distributions do not match. For instance, suppose the arrays $B(b,e,f,l)$ and $D(c,d,e,l)$ have distributions $\langle b,e,*,1\rangle$ and $\langle c,d,*,1\rangle$ respectively. If we want $T1$ to have the distribution $\langle c,d,f,1\rangle$ when evaluating $T1(b,c,d,f) = \sum_{e,l} B(b,e,f,l)\times D(c,d,e,l)$, B would have to be redistributed from $\langle b,e,*,1\rangle$ to $\langle *,*,f,1\rangle$ because the two distributions do not match. But since for $D(c,d,e,l)$, the distribution $\langle c,d,*,1\rangle$ is the same as $\langle c,d,f,1\rangle$, D is not redistributed.

3.2 Interaction Between Array Partitioning and Loop Fusion

The partitioning of data arrays among the processors and the fusions of loops on each processor are inter-related. Although in our context there are no constraints to loop fusion due to data dependences (there are never any fusion preventing dependences), there are constraints and interactions with array distribution: (*i*) both affect memory usage, by fully collapsing array dimensions (fusion) or by reducing them (distribution), (*ii*) loop fusion does not change the communication volume, but increases the number of messages, and therefore the start-up communication cost, and (*iii*) fusion and communications patterns may conflict, resulting in mutual constraints. We discuss these issues next.

(*i*) **Memory Usage and Array Distribution.** The memory requirements of the computation depend on both loop fusion and array distribution. Fusing a loop with index t between a node v and its parent eliminates the t-dimension of array v. If the t-loop is not fused but the t-dimension of array v is distributed along the d-th processor dimension, then the range of the t-dimension of array v on each processor is reduced to N_t/p_d. Let $DistSize(v,\alpha,f)$ be the size on each processor of array v, which has fusion f with its parent and distribution α. We have

$$DistSize(v,\alpha,f) = \prod_{i\in v.dimens} DistRange(i,v,\alpha,Set(f))$$

where $v.dimens = v.indices - \{v.sumindex\}$ is the array dimension indices of v before loop fusions, $v.indices$ is the set of loop indices for v including the summation index $v.sumindex$ if v is a summation node, $Set(f)$ is the set of fused indices for fusion f, and

$$DistRange(i,v,\alpha,x) = \begin{cases} 1 & \text{if } i\in x \\ N_i/p_d & \text{if } i\notin x \text{ and } i=\alpha[d] \\ N_i & \text{if } i\notin x \text{ and } i\notin \alpha \end{cases}$$

In our example, assume that $N_a = N_b = N_c = N_d = 1000$, $N_e = N_f = 70$, and $N_j = N_k = N_l = 30$. These are index ranges typical of the quantum chemistry calculations

$$C(i,k) = \sum_j A(i,j) \times B(j,k)$$
$$E(i,l) = \sum_k C(i,k) \times D(k,l)$$

(a) Formula sequence

```
for i = 1, Ni
 for k = (z-1) * Nk/4 + 1, z * Nk/4
  for j = 1, Nj
   C(i,k) += A(i,j) * B(j,k)
Redistribute C(i,k) from <k> to <l>=<*>
for i = 1, Ni
 for l = (z-1) * Nl/4 + 1, z * Nl/4
  for k = 1, Nk
   E(i,l) += C(i,k) * D(k,l)
```

(b) Before loop fusion

```
for i = 1, Ni
 Initialize C(k) to zero
 for k = (z-1) * Nk/4 + 1, z * Nk/4
  for j = 1, Nj
   C(k) += A(i,j) * B(j,k)
 Redistribute C(k) from <k> to <l>=<*>
 for  l = (z-1) * Nl/4 + 1, z * Nl/4
  for  k = 1, Nk
   E(i,l) += C(k) * D(k,l)
```

(c) After loop fusion

Fig. 3. An example of the increase in communication cost due to loop fusion

of interest, and are used elsewhere in the paper in relation to this example. If the array $B(b,e,f,l)$ has distribution $\langle b,e,*,1\rangle$ and fusion $\langle bf\rangle$ with T_2, then the size of B on each processor whose fourth dimension equals one would be $N_e/2 \times N_l = 1050$ words, since the e and l dimensions are the only unfused dimensions, and the e dimension is distributed onto 2 processors. Note that if array v undergoes redistribution from α to β, the array size on each processor after redistribution is $DistSize(v,\beta,f)$, which could be different from $DistSize(v,\alpha,f)$, the size before redistribution.

(*ii*) **Loop Fusion Increases Communication Cost.** The initial and final distributions of an array v determines the communication pattern and whether v needs redistribution, while loop fusions change the number of times array v is redistributed and the size of each message. Let v be an array that needs to be redistributed. If node v is not fused with its parent, array v is redistributed only once. Fusing a loop with index t between node v and its parent puts the collective communication code for redistribution inside the loop. Thus, the number of redistributions is increased by a factor of N_t/p_d if the t-dimension of v is distributed along the d-th processor dimension and by a factor of N_t if the t-dimension of v is not distributed. In other words, loop fusions cannot reduce communication cost. Instead, the number of messages increases with loop fusion, while the total volume of communication stays the same. Therefore, the communication cost increases, due to higher start-up costs. Consider the computation sequence presented in Fig. 3(a), where the array $C(i,k)$ is first "produced" from $A(i,j)$ and $B(j,k)$, and then "consumed" to produce $E(i,l)$. For this simple example, we assume that the computation is executed in parallel on 4 processors, with a one-dimensional logical processor view. Figure 3(b) shows the pseudo-code in the absence of fusion: the array $C(i,k)$ is re-distributed from $\langle k\rangle$ to $\langle l\rangle$ only once. In the presence of fusion, where the i-loop is the outermost loop, the dimensionality of the array C is reduced to $C(k)$, but the redistribution is performed N_i times. The pseudo-code in Fig. 3(c) illustrates this effect.

(*iii*) **Potential Conflict Between Array Distribution and Loop Fusion. Solution of the Conflict by Virtual Partitioning.** For the fusion of a loop between nodes u and v to be possible, the loop must either be undistributed at both u and v, or be distributed onto