Slicing Based Code Parallelization for Minimizing Inter-processor Communication

Mahmut Kandemir
Yuanrui Zhang
Sai Prasanth Muralidhara
Department of Computer Science and Engineering
The Pennsylvania State University
kandemir@cse.psu.edu,
{yuz123,sxm526}@psu.edu

Ozcan Ozturk
Department of Computer Engineering
Bilkent University
ozturk@cs.bilkent.edu.tr

Sri Hari Krishna Narayanan [*]
Department of Computer Science and Engineering
The Pennsylvania State University
snarayan@cse.psu.edu

ABSTRACT

One of the critical problems in distributed memory multi-core architectures is scalable parallelization that minimizes inter-processor communication. Using the concept of iteration space slicing, this paper presents a new code parallelization scheme for data-intensive applications. This scheme targets distributed memory multi-core architectures, and formulates the problem of data-computation distribution (partitioning) across parallel processors using slicing such that, starting with the partitioning of the output arrays, it iteratively determines the partitions of other arrays as well as iteration spaces of the loop nests in the application code. The goal is to minimize inter-processor data communications. Based on this iteration space slicing based formulation of the problem, we also propose a solution scheme. The proposed data-computation scheme is evaluated using six data-intensive benchmark programs. In our experimental evaluation, we also compare this scheme against three alternate data-computation distribution schemes. The results obtained are very encouraging, indicating around 10% better speedup, with 16 processors, over the next-best scheme when averaged over all benchmark codes we tested.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms

Performance

[*]Currently with the Argonne National Laboratory, USA.

Keywords

Parallelizing Compilers, Iteration Space Slicing, Automatic Code Parallelization, Code Analysis and Optimization

1. INTRODUCTION

Multi-core architectures, which combine two or more independent cores on a single chip, are expected to dominate the landscape of computer architecture in the near future. The potential performance gains that can be achieved through the use of multi-cores depend, to a large extent, on how parallelism is exploited by the application being executed.

With the emergence of multi-core systems, it is becoming increasingly important to parallelize sequential applications in an efficient manner. In the context of distributed memory multi-cores, minimizing inter-processor communication can lead to significant improvements in execution latencies. While optimizing compiler literature includes many papers that aim at reducing inter-processor data communication through code-data restructurings, most of these previously-proposed schemes work at a loop nest granularity and do not consider an entire program in making computation-data distribution decisions. Note that different data and computation distributions (partitions) across parallel processors can lead to entirely different inter-processor communication requirements and selecting the best distribution is a challenging task. The fact that computation and data distributions cannot be performed independent of each other makes this problem only harder.

In this work, we present a novel code parallelization scheme based on iteration space slicing [25]. Our proposed scheme targets distributed memory multi-core architectures, and tries to parallelize the input code and distribute computations (loop iterations) as well as data (arrays) such that the total amount of data communications between processors is minimized. To do this, it formulates the problem of data-computation distribution across parallel processors such that, starting with the partitioning of output arrays, it iteratively determines the partitions of other arrays in the application code. During this process, it also partitions computations (loop iteration space) across processors.

Based on this iteration space slicing based formulation of the problem, we also propose a solution scheme. The proposed scheme is evaluated using six data-intensive benchmark programs. Our results show that the proposed scheme is scalable and outperforms three alternate computation-data distribution schemes for all the benchmark programs and all processor counts tested. For example, with 16 cores, our scheme achieves an average speedup of 11.1, which is about 10% better than the next-best scheme.

The rest of this paper is organized as follows. We give a description of the architectural template our scheme targets and explain the impact of data-computation partitioning on inter-processor communication requirements in Section 2. The prior work relevant to this paper is discussed in Section 3. Section 4 provides the basics of iteration space slicing, and the technical details of our proposed scheme are presented in Section 5. Our experimental setup and results are discussed in Section 6 and the paper is concluded in Section 7.

2. ARCHITECTURAL ABSTRACTION

The architecture we focus on in this study is a distributed memory based multi-core system where each core has a separate address space. The communication between two cores in this architecture is through explicit message passing (e.g., using a library such as MPI [22]). Clearly, minimizing inter-processor communication in this architecture is critical for good performance. Along this direction, prior research discussed numerous optimization schemes which can be broadly divided into two categories (as will be discussed in the next section). In the first category are the works that distribute data and computation across processors such that the overall communication requirements are minimized. In comparison, the second category includes studies that try to minimize the number of communication calls in the code by applying schemes such as message vectorization, message aggregation, and message coalescing.

To illustrate how much difference different data-computation distributions can make in communication requirements across parallel processors, consider the following code fragment written in a C-like pseudo-language:

```
for($i_1 = 1; i_1 \le N; i_1++$)
  for($i_2 = 2; i_2 \le N-1; i_2++$)
    $A[i_1, i_2] = (B[i_1, i_2-1] + B[i_1, i_2] + B[i_1, i_2+1])/3$
```

This is a simple code that performs three-point stencil computation, variants of which are used frequently in signal and image processing. Within the body of the loop, each element of array A is updated using three elements of array B. Let us now consider two alternate data partitioning schemes for the arrays involved in this computation, assuming a 4 processor multi-core machine for simplicity. The first one, depicted in Figure 1(a) partitions both the arrays into column-blocks across processors. We see that, in this case, the data elements (from array B) across block-boundaries (one such set of elements are highlighted) should be communicated across neighboring processors. In the second partitioning scheme, in comparison, the data distribution is row-block wise for both arrays, as illustrated in Figure 1(b). In this case, all data accesses are localized, i.e., no inter-processor communication is needed. While one may say that row-block partitioning is an easy choice in this particular scenario, the problem becomes non-trivial in realistic embedded signal and image processing applications where we have tens of loop nests, each processing a subset of tens

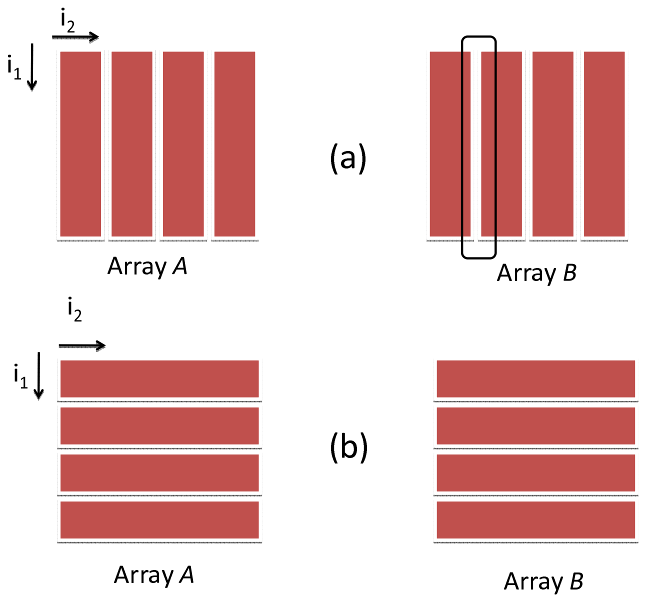

Figure 1: Two different data partitioning schemes: (a) partitions both the arrays into column-blocks across processors, and (b) partitions the arrays in row-blocks across processors.

of multi-dimensional arrays of signals declared in the program.

Most of the prior data-computation distribution techniques targeting distributed memory architectures either operate on a loop nest granularity, i.e., they handle one loop nest at a time, or process multiple (separate) loop nest by carrying data/computation distribution decisions across them using several heuristics. In our opinion, to minimize inter-processor communication, a parallelization scheme should consider an entire application code. Our goal in this work is to present and evaluate a global (program wide) data-computation distribution scheme that minimizes inter-processor communication. We formulate our problem using iteration space slicing theory, discuss a solution strategy, and evaluate it experimentally using several data-intensive, array-based application programs.

3. RELEVANT PRIOR WORK

Program slicing was introduced by Weiser in his seminal paper [31]. A program slice consists of the parts of a program that may affect the values computed at some point of interest. Tip surveys various program slicing techniques and presents a classification of various static and dynamic program slicing techniques for features such as procedures, arbitrary control flow, and interprocess communication [29]. Horwitz et al present a technique to generate a slice of an entire program with the slice crossing the boundaries of procedure calls, also called interprocedural slicing [17]. Reps and Yang describe the semantics of program slicing, which is the relationship between the execution behavior of a program and the execution behavior of its slices [26].

Agrawal and Horgan present the concept of dynamic dependence graph and use it to compute dynamic slices [1]. Harrold and Ci adapt a control-flow based interprocedural slicing algorithm which also reuses slicing information for better efficiency [15]. Liang and Harrold extend this reuse-driven interprocedural technique to an efficient and precise interprocedural algorithm for recursive programs and programs with pointers [21]. Zhang and Gupta present a dynamic dependence graph based program slicing technique which is both cost effective and efficient [32]. Gallagher and Lyle apply the program slicing technique to the software maintenance problem by extending the program slice to a decomposition slice, which captures all computation on a given variable [11]. Komondoor and Horwitz identify the problem of duplicated code and further present a slicing based technique to identify clones and show them to the user [20]. Iteration space slicing, on which we base our work, is introduced by Rosser and Pugh [25]. We will discuss iteration space slicing in more depth later in the paper.

Another area of related work we wish to discuss here regards the compiler support for distributed memory based parallel architectures. The compiler framework discussed in [4] generates a parallel program from a given serial program for efficient execution on a distributed memory machine. Amarasinghe and Lam present techniques to partition the computation across different processors in a machine. They further present algorithms to generate SPMD (single program multiple data) program to be run on each processor [2]. Anderson describes elegant computation and data distribution schemes in her Ph.D. thesis [3]. Chakrabarti et al present a compiler algorithm for global analysis and communication optimization in data parallel programs [7]. They consider all the communication in a procedure body at once instead of considering the loop nests and array references distinctly. Chatterjee et al present ways to generate local addresses and communication sets in distributed memory implementations of data-parallel languages such as High Performance Fortran (HPF) [8]. Gong et al reduce communication overhead and execution delays through optimizations such as avoiding sequentialization caused by communication, and by overlapping communication and computation [12]. Gupta and Banarjee discuss several methods that identify the constraints on data partitioning and present various automatic data partitioning techniques for distributed memory architectures [13]. Kennedy et al show that combining data dependence analysis and data-flow analysis can yield promising results [18]. Hall et al present a set of algorithms to perform interprocedural analysis, optimization, and code generation for Fortran D, where compilation involves only one pass over the procedures [14]. Kennedy and Sethi show the importance of taking the resource constraints into account when deciding about communication placement to ensure correctness [19]. Basumallik et al present a technique to extend the shared memory parallel programming in OpenMP to distributed memory systems thereby exploiting communication pattern reuse [5].

Different schemes have been proposed for automatic code parallelization within different domains. In the context of high-end computing, the relevant studies include [33, 14, 34]. Bondalapati propose techniques for parallelizing nested loops that appear in the digital signal processing domain [35]. To parallelize such loops, they exploit the distributed memory available in the reconfigurable architecture by implementing a data context switching technique. Goumas et al. [36] propose a framework to automatically generate parallel code for tiled nested loops. They have implemented several loop transformations within the proposed approach using MPI [22], the message-passing parallel interface. A modulo scheduling algorithm to exploit loop-level parallelism for coarse-grained reconfigurable architectures has been proposed by Mei et al. [37]. Hogstedt et al. [39] investigate the parallel execution time of tiled loop nests. They use a prediction formula for the execution time of tiled loop nests to aid the compiler. Using this prediction, compiler is able to automatically determine the tiling parameters that minimizes the execution time. Navarro et al. [38] target minimizing communication and load imbalance in parallel programs. Our work is different from most of these efforts as well as prior HPF-related studies since we can handle an entire program code that contains a series of nested loops.

Finally, our work is also related to the efforts that employ polyhedral arithmetic and integer linear programming for code analysis and optimization purposes. Pouchet et al [24] consider iterative optimization in the context of a polyhedral model. They focus on the class of loop transformation, which can be expressed as one-dimensional affine schedules, and define a systematic exploration method to enumerate the space of all legal, distinct transformations in this class. Feautrier et al [10] describe a method for solving systems of affine inequalities, which can be used for different optimization purposes. Verdoolaege et al [30] study enumeration of integer projections of parametric polytope.

Our work is different from these studies as we use program slicing in a systematic fashion to derive data and computation partitions for minimizing inter-processor communications. In our experiments, we also compare our strategy with three alternate data-computation distribution techniques.

4. BASICS

Slicing, proposed originally by [31], is a program analysis technique that computes a *slice* of a program based on some slicing criterion. A program slice consists of the parts of a program that may affect or maybe affected by the values computed at some point of interest in the code. Important applications of slicing include debugging, code maintenance, program analysis and program optimization, as discussed in Section 3. Pugh and Rosser [25] proposed a strategy that extends conventional slicing to array-intensive applications and referred to this version as the *iteration space slicing.* Using iteration space slicing, one can answer questions such as "Which iterations of which statements might affect the values of a given set of elements from array A?" In this section, we summarize how iteration space slicing can be used for array-based applications constructed from nested loops and data arrays.

```
for($i_1 = 1; i_1 < N; i_1 + +$)
  for($i_2 = 1; i_2 < N; i_2 + +$)
  $\cdots A[i_1 + 1, i_2 + 1] \cdots$
```

Figure 2: An example code fragment.

We start by discussing the mappings between loop iterations and array elements. Consider the code fragment in Figure 2. We use affine sets to represent the set of loop iterations and array elements. For example, the iteration space of the loop nest in this fragment can be expressed as $Y_1 = \{(i_1 \quad i_2)^T \mid (1 \le i_1 \le N-1) \;\&\&\; (1 \le i_2 \le N-1)\}$. Similarly, the set of array elements accessed by the reference

$(1\ \ 1)^T \leq \vec{I} \leq (N-1\ \ N-1)^T$
$\quad\cdots A[h_1(\vec{I})]\cdots$

Figure 3: Compact form (representation) of the fragment in Figure 2.

shown can be expressed as $X_1 = \{(d_1 \quad d_2)^T \mid \exists(i_1 \quad i_2) \in Y_1\ (d_1 = i_1 + 1\ \&\&\ d_2 = i_2 + 1)\}$. We use mappings from iteration space to data space to represent array references. For instance, for the reference shown in this code fragment, we can define a mapping (h_1) as follows:

$$\begin{aligned} h_1 \quad &: \quad Y_1 \longrightarrow X_1 \\ h_1 \quad &= \quad \{(i_1\ \ i_2)^T \longrightarrow (d_1\ \ d_2)^T \mid \\ & \qquad (i_1\ \ i_2)^T \in Y_1 \quad \&\& \quad (d_1\ \ d_2)^T \in X_1\}. \end{aligned}$$

As a result, we can write $X_1 = h_1(Y_1)$ and $Y_1 = h_1^{-1}(X_1)$ for convenience.[1] We can also re-write the code fragment in Figure 2 as shown in Figure 3. In this *compact form,* $\vec{I}$, which is the same as $(i_1 \quad i_2)^T$, is referred to as the *iteration vector,* a vector of loop iterators which takes its values from the iterations space of the loop nest to which it belongs. Note that the first expression (the upper one) in the compact form indicates the bounds of the iteration vector, and the second expression is the same as the original loop body, except that it is written using our mapping h_1. We would like to say that all the (data/iteration) sets we use in our problem formulation and solution strategy (like X_1 and Y_1) are Presburger sets (i.e., sets that are constructed from expressions formed using Presburger arithmetic) and can therefore be manipulated using polyhedral tools such as the Omega Library from University of Maryland [23].

Consider now the sample code fragment (compact form) depicted in Figure 4. In this fragment, array A is updated in the first loop nest and is used to update array B in the second loop nest. Let us focus on a set of elements of array B assigned to a processor (e.g., as a result of data distribution of arrays across processors) and denote this set using X. Now, based on our discussion above, the set of iterations that access (assign value to) these data elements can be represented using $h^{-1}(X)$. Going further, the set of data elements accessed, by the right-hand-side reference (to array A) can be expressed using $g \circ h^{-1}(X)$. In this paper, we use the operator "$\circ$" as a means of composing an iteration set from a data set, or vice versa. Continuing with the first loop nest now (i.e., going from the second loop nest to the first one), the set of iterations that access (assign values) the data elements in $g \circ h^{-1}(X)$ can be written as $f^{-1} \circ g \circ h^{-1}(X)$. This discussion clearly shows that, by starting with a subset of data elements from array B assigned in the second loop nest and using (backward) iteration space slicing, we are able to determine the set of iterations in the first loop nest that assign values to the data elements of array A, which affect these elements of B (accessed in the second loop nest). In short, we determined the set of loop iterations in the first nest that affect the values of the elements of array B in the second nest. In the rest of this paper, we demonstrate how this concept of (backward) iteration space slicing can be used for minimum-communication computation-data partitioning in distributed memory machines.

[1]Note that, clearly, some mappings do not have inverse in mathematical sense. However, using a polyhedral tool, we can compute the elements in $h_1^{-1}(X_1)$.

$\cdots \leq \vec{I} \leq \cdots$
$\quad A[f(\vec{I})] = \cdots$

$\cdots \leq \vec{I} \leq \cdots$
$\quad B[h(\vec{I})] = A[g(\vec{I})]$

Figure 4: An example code fragment in compact form.

LN(1): $\cdots \leq \vec{I} \leq \cdots$
$\quad A_1[f_{1,1}(\vec{I})] = A_0[f_{0,1}(\vec{I})]$
LN(2): $\cdots \leq \vec{I} \leq \cdots$
$\quad A_2[f_{2,2}(\vec{I})] = A_1[f_{1,2}(\vec{I})]$
LN($s-1$): $\cdots \leq \cdots$
LN(s): $\cdots \leq \vec{I} \leq \cdots$
$\quad A_s[f_{s,s}(\vec{I})] = A_s[f_{s-1,s}(\vec{I})]$
LN($s+1$): $\cdots \leq \cdots$
LN($n-1$): $\cdots \leq \vec{I} \leq \cdots$
$\quad A_{n-1}[f_{n-1,n-1}(\vec{I})] = A_{n-2}[f_{n-2,n-1}(\vec{I})]$
LN(n): $\cdots \leq \vec{I} \leq \cdots$
$\quad A_n[f_{n,n}(\vec{I})] = A_{n-1}[f_{n-1,n}(\vec{I})]$

Figure 5: A generic code template in its compact form. LN(s) indicates the sth loop nest—the loop nest in which array A_s is updated, and A_n is the output array.

5. PROBLEM FORMULATION AND PROPOSED SOLUTION

5.1 Determining Data and Computation Partitions

Our goal is to derive data and computation partitions across the processors of a distributed memory multi-core architecture to minimize inter-processor communication. For each loop nest, we want to determine a computation (loop iteration) distribution across processors and, for each array, we want to determine a distribution of its elements across (the local memories of) the processors. We consider the generic application program in Figure 5, written in its compact form. In this program, $n+1$ different data arrays (A_0 through A_n) are accessed using n (independent) loop nests. There are two main reasons why we focus on this generic code form. First, it represents the program structure of many array-intensive, loop-based application codes. In these codes, each loop nest computes new values for a set of array elements and the computation ends when the values of the elements of the output array (in this case, A_n) are computed. The second reason is that it is simple enough to help us illustrate and explain how our iteration space slicing based approach operates. *We discuss later how we handle the cases where the input code deviates from this generic format.*

We assume that $X_{p,n}$ is the set of data elements from array A_n (the output array) assigned to processor p where $1 \leq p \leq P$, that is, we start with the partitioning of the output array A_n across the parallel processors (how this partitioning is determined will be discussed later) and this is the starting point of our backward data-computation partitioning scheme. Using this data partitioning, the set of loop iterations assigned to any processor p in loop nests 1 through n can be expressed as follows using our "$\circ$" operator:

$$\begin{aligned}
Y_{p,n} &= f_{n,n}^{-1}(X_{p,n}) \\
Y_{p,n-1} &= f_{n-1,n-1}^{-1} \circ f_{n-1,n} \circ f_{n,n}^{-1}(X_{p,n}) \\
Y_{p,n-2} &= f_{n-2,n-2}^{-1} \circ f_{n-2,n-1} \circ f_{n-1,n-1}^{-1} \\
&\quad \circ f_{n-1,n} \circ f_{n,n}^{-1}(X_{p,n}) \\
\cdots &= \cdots \\
Y_{p,1} &= f_{1,1}^{-1} \circ f_{1,2} \circ \cdots \circ f_{n-2,n-1} \circ f_{n-1,n-1}^{-1} \\
&\quad \circ f_{n-1,n} \circ f_{n,n}^{-1}(X_{p,n})
\end{aligned}$$

In these expressions, $Y_{p,s}$ captures the set of iterations assigned to processor p from loop nest s. Notice that this process operates in a backward fashion on the input program code. More specifically, using the distribution of the elements of array A_n across processors, we first determine the iteration partitioning for the nth loop nest (LN(n)). Then, we move to loop nest $(n-1)$ and determine the partitioning of its iteration space across parallel processors, and so on.

For ease of discussion, let us define $Q_{p,s}$ as $f_{s,s}^{-1} \circ f_{s,s+1} \circ \cdots \circ f_{n-1,n-1}^{-1} \circ f_{n-1,n} \circ f_{n,n}^{-1}$. As a result, we have $Y_{p,s} = Q_{p,s}(X_{p,n})$. Note that $Q_{p,s}$ summarizes the function that returns us the set of loop iterations to be assigned to processor p from loop nest s, given as input $X_{p,n}$, the set of data elements from array A_n assigned to processor p (the partitioning of the output data array). In this way, for each loop nest s and processor p, we can calculate $Y_{p,s}$ and this gives us distribution of computation (loop iterations) across available processors.

Similarly, we can determine the set of data elements accessed by different processors as follows (note that the calculations for array distribution and those for computation distribution share a lot of computations between them, and so, efficient implementations are possible):

$$\begin{aligned}
Z_{p,n} &= X_{p,n} \\
Z_{p,n-1} &= f_{n-1,n} \circ f_{n,n}^{-1}(X_{p,n}) \\
Z_{p,n-2} &= f_{n-2,n-1} \circ f_{n-1n-1}^{-1} \\
&\quad \circ f_{n-1,n} \circ f_{n,n}^{-1}(X_{p,n}) \\
\cdots &= \cdots \\
Z_{p,1} &= f_{1,2} \circ \cdots \circ f_{n-2,n-1} \circ f_{n-1,n-1}^{-1} \\
&\quad \circ f_{n-1,n} \circ f_{n,n}^{-1}(X_{p,n})
\end{aligned}$$

$Z_{p,r}$ is the set of data elements accessed by processor p from array A_r. As before, defining $R_{p,r}$ as $f_{r,r+1} \circ f_{r+1,r+1}^{-1} \circ \cdots \circ f_{n-1,n-1}^{-1} \circ f_{n-1,n} \circ f_{n,n}^{-1}$, we have $Z_{p,r} = R_{p,r}(X_{p,n})$. Note that $R_{p,r}$ is a function that gives the set of data elements from array A_r assigned to processor p, given the partitioning of the output array A_n across parallel processors (i.e., given the $X_{p,n}$ sets).

Algorithm 1 Integrated data-computation distribution algorithm based on slicing.

```
1: partition($\mathcal{J}$, $X_{p,n}, \forall p$)
2: for ($k := n; k \geq 1; k--$) do
3:     for ($p := 1; p \leq P; p++$) do
4:         $Q_{p,k} := f_{k,k}^{-1} \circ f_{k,k+1} \circ \cdots \circ f_{n-1n-1}^{-1} \circ f_{n-1,n} \circ f_{n,n}^{-1}$
5:         $Y_{p,k} := Q_{p,k}(X_{p,n})$
6:         $R_{p,k} := f_{k,k+1} \circ f_{k+1,k+1}^{-1} \circ \cdots \circ f_{n-1,n} \circ f_{n,n}^{-1}$
7:         $Z_{p,k} := R_{p,k}(X_{p,n})$
8:     end for
9: end for
10: return {$Y_{p,k}$ and $Z_{p,k}, \quad \forall p, \forall k$}
```

Algorithm I summarizes our slicing based computation and data partitioning scheme explained above ($\mathcal{J}$, the first input parameter, represents the input program to be partitioned). At each step, this algorithm determines $Y_{p,k}$ and $Z_{p,k}$ for all p. We want to emphasize that, at this point, nothing can be said about optimality of these iteration space and data space partitionings. It is clear that the quality of the resulting iteration/data space partitioning depends on the partitioning of A_n across processors and, consequently, different partitionings of this array can lead to different iteration space and data space partitionings (and consequently different inter-processor communication requirements). In the next subsection, we discuss how we determine a good partitioning for A_n that minimizes the amount of inter-processor communication (when all loop nests and all arrays in the code are considered).

5.2 Minimizing Inter-processor Communication

Let us now consider a partitioning of elements of array A_n across P processors: $X_{1,n}, X_{2,n}, \cdots, X_{P,n}$. We define *sharing factor* induced by this partition as:

$$SF(X_{1,n}, X_{2,n}, \cdots, X_{P,n}) = \\ \textstyle\sum_{1 \leq p \leq P} \sum_{1 \leq q \leq P, p \neq q} \sum_{1 \leq r \leq n} |Z_{p,r} \cap Z_{q,r}|,$$

where "$\cap$" denotes set intersection, and $Z_{p,r}$ is as defined in the previous subsection. Sharing factor captures the sum of the differences between the common data elements accessed by pairs of processors. We search for a partitioning that minimizes the sharing fact. In the ideal case, we do not want the processors to share any data and this would lead to no communication. Clearly, in this case, each loop nest can be executed in parallel without requiring any communication. However, in practice, it may not be possible to obtain a communication-free partitioning; instead, we try to obtain a partitioning that minimizes communication. Note also that we can also have a weighted version of the sharing factor defined above:

$$SF(X_{1,n}, X_{2,n}, .., X_{P,n}) = \\ \textstyle\sum_{1 \leq p \leq P} \sum_{1 \leq q \leq P} \sum_{1 \leq r \leq n} \alpha_{p,q,r} |Z_{p,r} - Z_{q,r}|,$$

$\alpha_{p,q,r}$ is a weight capturing the importance of reducing data sharing between processors p and q regarding loop nest r.

Before defining our problem formally, let us make another definition: *partitioning factor.* A partitioning factor indicates how a dimension of a multi-dimensional array is divided across processors. For example, the partitioning of array A shown in Figure 1(b) can be expressed using the partitioning factors (4,1), indicating that the first dimension of the array is divided across 4 processors and the second dimension is not distributed (i.e., it is sequential). In a similar fashion, the distribution of the same array in Figure 1(a) can be expressed as using partitioning factors (1,4). For an a-dimensional array, if $(b_1 b_2 \cdots b_a)$ are the partitioning factors, we have $b_1 b_2 \cdots b_a = P$, where P is the total number of cores in the multi-core architecture.

Recall that, in the iteration space slicing based approach described in the previous subsection, all computation and data partitionings depend on (are determined by) the partitioning of the output array A_n. Therefore, to reach the minimum communication partitions, we need to determine a suitable partitioning of array A_n that leads to it. Based on this discussion, we can formulate the problem of determining the optimal partitioning of an output array A_n across P processors as follows:

Let A_n has a dimensions. Find partitioning factors $(b_1, b_2, \cdots, b_a)$ $(b_1 b_2 \cdots b_a = P)$ such that, for any other partitioning factor $(c_1, c_2, \cdots, c_a)$ $(c_1 c_2 \cdots c_a = P)$, we have

$$SF(Xb_{1,n}, Xb_{2,n}, \cdots, Xb_{P,n}) \leq SF(Xc_{1,n}, Xc_{2,n}, \cdots, Xc_{P,n}),$$

where $Xb_{p,n}$ and $Xc_{p,n}$ are allocations for processor p $(1 \leq p \leq P)$ from array A_n under partitioning factors $(b_1, b_2, \cdots, b_a)$ and $(c_1, c_2, \cdots, c_a)$, respectively.

Informally, we want to determine the partitioning factor that minimizes inter-processor data sharing. Our approach to this problem is to build *Presburger sets* for sharing factor (SF) in terms of unknowns (b_is) and determine the values of these unknowns such that the overall value of the sharing factor is minimized. For example, assuming all b_is are unknowns, we can express SF in terms of these unknowns and then try to determine the best values for these unknowns to minimize SF. Unfortunately, this is easier said than done, as Presburger sets can contain existential and universal quantifiers and complex expressions. Instead, we take a rather indirect approach to this problem which exploits the code generation utility provided by the Omega Library and other similar polyhedral toolsets. The Omega Library has a function called code-gen(.) which takes as input an arbitrarily complex Presburger set and generates as output a code (a series of potentially nested loops) that enumerates the elements in that Presburger set. We first generate this code and then add an outer loop to this code which goes over the potential values on unknowns (b_is). We then execute this code (at compile time), and for each potential values of unknowns, print out the number of times the loops iterate (this can be determined by increasing a counter variable inside the generated loops). The values of the unknowns that give the minimum number (counter value) are the values that also minimize the amount of inter-processor communication. Algorithm II gives the pseudo-code for our algorithm that employs this strategy.

Algorithm 2 Algorithm to determine data-computation partitions with minimum communication.

```
1: **min-comm**($\mathcal{J}$)
2: let $(b_1, b_2, \cdots, b_a)$ be the partitioning factors (a vector
   of unknowns)
3: compute $Xb_{1,n}, Xb_{2,n}, \cdots, Xb_{P,n}$
4: call **partition**($\mathcal{J}$, $Xb_{p,n}, \forall p$)
5: compute $SF(Xb_{1,n}, Xb_{2,n}, \cdots, Xb_{P,n})$ using $Z_{p,k}$ sets
6: $\mathcal{T}$:= call code-gen ($SF(Xb_{1,n}, Xb_{2,n}, \cdots, Xb_{P,n})$)
7: $\mathcal{W}$:= put an outer loop (iterating over potential values
   of $(b_1, b_2, \cdots, b_a)$ enclosing $\mathcal{T}$
8: execute $\mathcal{W}$ for each $(b_1, b_2, \cdots, b_a)$ with $b_1 b_2 \cdots b_a = P$
9: return $(b_1, b_2, \cdots, b_a)$ that minimizes number of times
   the loops in $\mathcal{W}$ iterate
```

5.3 Discussion

In this section, we first discuss what happens if a loop in our target application is not parallelizable. After that, we elaborate on what happens if the target application deviates from the template shown in Figure 5.

Recall that our main target code is of the type shown in Figure 5. We assume that the code parallelization decisions have already been made before our approach is invoked and the slicing-based computation distribution we propose is mainly for parallel loops (i.e., loops whose iterations can be executed in parallel). If a loop is to be executed sequentially (due to dependences), our approach can process that loop by simply assuming that all iterations will be executed on a single processor (and this actually happened in two of our benchmarks, presented later in Section 6). As a result, our approach will be still applicable (but be less effective) when majority of the loops cannot be executed in parallel.

We now discuss what happens if the input program we want to optimize does not exactly fit in the generic code template shown in Figure 5. Specifically, we elaborate on several sample scenarios and briefly explain how our approach handles these scenarios. Let us first consider the scenario where an array has more than one reference on the right hand side. As an example, consider the following statement in a loop nest:

$$A_r[f_{r,r}(\vec{I})] = A_{r-1}[f_{r-1,r}(\vec{I})] + A_{r-1}[g_{r-1,r}(\vec{I})],$$

where array A_{r-1} is referenced twice on the right hand side. In this case, we can compute $Z_{p,r-1}$ as

$$\begin{aligned} Z_{p,r-1} &= f_{r-1,r} \circ \cdots \circ f^{-1}_{n-1,n-1} \circ f_{n-1,n} \circ f^{-1}_{n,n}(X_{p,n}) \\ &\cup \;\; g_{r-1,r} \circ \cdots \circ f^{-1}_{n-1,n-1} \circ f_{n-1,n} \circ f^{-1}_{n,n}. \end{aligned}$$

That is, in determining the set of elements assigned to processor p from array A_{r-1}, we take into account both the references it has. The other cases can be handled similarly. For example, if there are references to different arrays on the right hand side, to determine $Z_{p,r}$, we need to consider all the references to array A_r that occur in all different right

Parameter	Value
Processor	1GHz
Issue width	4 per cycle
On-chip memory	512KB/processor, 8 cycles access latency
Off-chip memory latency	220 cycles
On-chip network	Dual-bus, 3 cycles average bus contention

Table 1: Our simulation parameters and their default values.

Benchmark	Brief Description	Data Size (MB)
ADPCM	Adaptive diff. pulse code modulation (MediaBench)	8.21
CELP	Linear prediction based speech coder	6.73
FACEREC	Face recognition algorithm (SpecOMP)	4.78
ART	Image recognition using neural networks (SpecOMP)	6.92
LU	LU solver (NAS)	3.54
SP	Pentadiagonal solver (NAS)	5.44

Table 2: Benchmark codes used in our evaluation.

hand sides. Finally, it is possible that we may have more than one output array (like A_n). This case is also easy to handle as we can consider each output array in isolation. Although we do not discuss here in detail, we can use similar strategies to compute the $Y_{p,s}$ sets under these situations. We want to mention that our current implementation handles any program code shape, including those that exhibit the scenarios discussed above.

6. EXPERIMENTAL EVALUATION

Our goal in this section is to present an experimental evaluation of the proposed approach. In particular, we are interested in answering two important questions. First, how scalable is this approach? And second, how does it compare against existing data-computation partitioning strategies? Table 1 gives the default values of our simulation parameters, and Table 2 lists the benchmark codes used in our evaluation. The last column of this table gives the total data size processed by each benchmark program. We used SUIF [28] from Stanford University to implement our scheme and the other schemes against which we compare our approach. In SUIF, independently developed compilation passes work together by using a common intermediate format to represent programs. We implemented our code-data partitioning step as an independent pass within SUIF. To do this, we connected the Omega Library to SUIF, i.e., the library worked as a sub-module of the SUIF infrastructure. All our simulations have been performed using the SIMICS tool-set [27], which is a full-system simulator used to run unchanged production binaries of the target hardware at high-performance speeds. For conducting our experiments, we first enhanced SIMICS with accurate timing models. Then, using this enhanced version, we modeled a distributed-memory multicore system and executed the programs compiled for the SPARC-V9 ISA. For codes that are not in the MPI form originally (e.g., CELP), we hand-coded their MPI versions. We noted that the additional increase in compilation time due to our approach was at most 74% (occurred in compiling the FACEREC program).

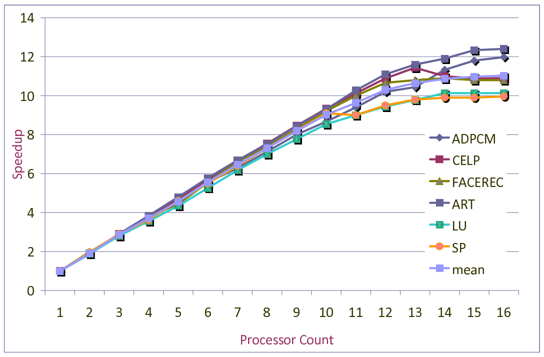

Figure 6: Scalability of our scheme (speedup over sequential codes) when the number of processors is modulated between 1 and 16.

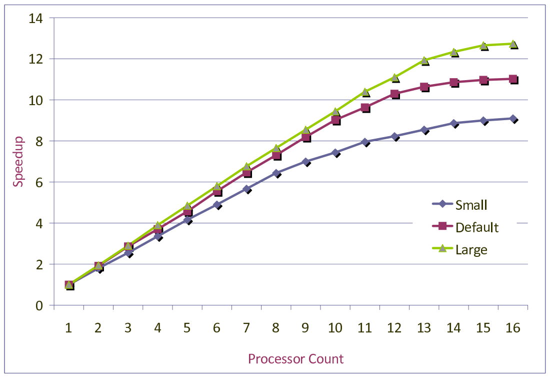

Figure 7: Speedup values with different dataset sizes.

The first set of results we collected are presented in Figure 6 and shows the scalability of our scheme (speedup over sequential codes) when the number of processors is modulated between 1 and 16. The curve marked as "mean" represents the mean value across all applications. Our observation is that the proposed strategy scales quite well. For example, with 8 and 16 processors, we achieve average speedup values of 7.4 and 11.1, respectively. Maybe more importantly, as illustrated in Figure 7, the speedup values we obtain increase as the dataset sizes are increased. In this plot, each curve represents the mean value when considering all six benchmarks we have. The curve marked using "Default" represents the default dataset sizes used in our applications (given in the last column of Table 2), and "Small" and "Large" represent, respectively, the dataset sizes that are roughly half and roughly twice of the default size. These results are encouraging considering the trend that dataset sizes of parallel applications increase much faster than the increase in on-chip memory capacities.

In addition to these experiments, we also compared our approach to three state-of-the-art data and computation distribution strategies. We can summarize these three alternate schemes as follows (note that we do not have direct comparison against [25] as it uses iteration space slicing for the purposes of different optimizations—optimizing fused loops, tolerating message latency, and enhancing message aggregation):

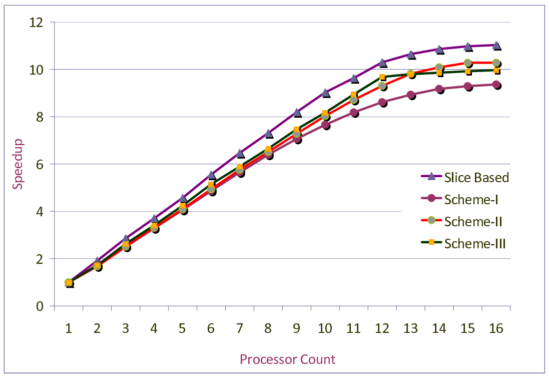

Figure 8: Speedup values for Scheme-I, Scheme-II, and Scheme-III.

• Scheme-I: This is a data locality-aware parallelization strategy, where each loop nest is first restructured to maximize data reuse in innermost loop positions. This, as a side effect, also helps remove data dependences from outer loop positions. The outermost loop that does not carry any data dependence is then parallelized. After these parallelizations (computation distributions) have been performed, the most frequently demanded data distribution for each array (when considering all loop nests in the program) is selected as its data distribution across processors.

• Scheme-II: This is similar to the scheme discussed in [3]. This scheme processes loop nests one-by-one and carries parallelization and data distribution decisions made in currently-processed nest to the next ones. It is essentially a global approach (like ours) to code parallelization for distributed memory message-passing architectures.

• Scheme-III: This is a data distribution oriented code parallelization scheme. In this scheme, which is semi-automatic, first, the elements of data arrays declared in the code are distributed across the local memories of the involved processors. This distribution is carried out by hand using a programmer's expert knowledge (in our implementation we carried out a profile-driven distribution of data to have a fair comparison against our scheme). Based on this data distribution, then the loop nests are parallelized one-by-one and for each loop its iterations are distributed across processors. In a sense, this strategy is similar to the used in HPF and other data parallel computing paradigms [16].

Figure 8 plots the speedup curves for these three schemes (the results with our proposed scheme are reproduced for ease of comparison). Each curve represents the mean value across all our benchmarks. We can summarize our observations as follows. First, our iteration space slicing based approach generates better results than all three alternate schemes tested in all processor counts. Second, Scheme-I does not generate good results, as it is more suitable for shared memory machines in which management of shared cache space is of primary importance, rather than minimizing inter-processor data sharing. Third, up to 12 processors, Scheme-III generates results better than other two alternate schemes, indicating the potential of careful data distribution by hand. Fourth, with larger number of cores, Scheme-II generates results that are closer to our scheme (among all schemes). That is, it is important to consider the data access patterns of all the loop nests in parallelizing application programs. The main difference between our scheme and that in [3] is that our approach explores a much larger search space than the one in [3]. In fact, in our experiments, the loops formed using the code-gen(.) utility of the Omega Library tested all partitioning factors $(b_1, b_2, \cdots, b_a)$, where $b_1 b_2 \cdots b_a = P$, P being the number of processors used, and we selected the vector that minimizes the total inter-processor communication volume. In comparison, the scheme in [3] processes the loop nests one by one and for each loop nest makes some decisions before moving to the next one. During this process, the accumulated conditions restrict the search space for the loop nests yet to be processed. The performance difference between our scheme and [3] shown in Figure 8 is primarily due to this difference in search space sizes.

7. CONCLUDING REMARKS

The main contribution of this paper is a new data and computation distribution scheme for array-intensive application programs. The goal of this scheme is to minimize the amount of inter-processor communication across processors in a distributed-memory multi-core architecture. This is achieved using iteration space slicing which, starting with output arrays and processing loop nests in a reverse order, determines optimal partitionings for the manipulated arrays across involved processors. We used a mix of six data-intensive applications to test the success of this scheme and compared it, in our experiments, to three alternate data-computation distribution strategies. Our results show that the proposed scheme has good scalability and outperforms the alternate schemes tested for all benchmark programs and all processor counts. For example, with 16 cores, our scheme achieves an average speedup of 11.1, this result is about 10%, 12% and 17% better than the speedups obtained using the three alternate strategies on the same number of processors. Our ongoing work includes integrating this data-computation partitioning strategy with existing parallelism-related compiler optimizations and collecting results with control-dominated applications.

8. ACKNOWLEDGMENTS

This work is supported in part by NSF grants 0702519, 0720645, 0720749, 0811687, and 0821527.

9. REFERENCES

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 1990.

[2] S. P. Amarasinghe and M. S. Lam. Communication optimization and code generation for distributed memory machines. In *ACM SIGPLAN Notices*, 1993.

[3] J. M. Anderson. Automatic computation and data decomposition for multiprocessors. *Ph.D Thesis,* Stanford University, March 1997.

[4] P. Banerjee, J. A. Chandy, M. Gupta, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The paradigm compiler for distributed-memory multicomputers. In *IEEE Computer*, 1995.

[5] A. Basumallik, S.-J. Min, and R. Eigenmann. Programming distributed systems using Openmp. In *Proceedings of HIPS*, 2007.

[6] R. Brightwell, R. Riesen, K. D. Underwood. Analyzing the impact of overlap, offload, and independent progress for message passing interface applications. In *International Journal of High Performance Computing Applications*, May 2005.

[7] S. Chakrabarti, M. Gupta, and J. deok Choi. Global communication analysis and optimization. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 1996.

[8] S. Chatterjee, J. R. Gilbert, F. J. E. Long, R. Schreiber, S. Hua Teng, D. John, and R. Gilbert. Generating local addresses and communication sets for data-parallel programs. In *Journal of Parallel and Distributed Computing*, 1995.

[9] A. Danalis , K.-Y. Kim , L. Pollock , M. Swany. Transformations to parallel codes for communication-computation overlap. In *Proceedings of the ACM/IEEE Conference on Supercomputing,* 2005.

[10] P. Feautrier, J.F. Collard, C. Bastoul. Solving systems of affine (in)equalities. *Technical Report,* PRiSM,Versailles University, France, 2002.

[11] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. In *IEEE Transactions on Software Engineering*, 17:751–761, 1991.

[12] C. Gong, R. Gupta, and R. Melhem. Compilation techniques for optimizing communication on distributed-memory systems. In *In Proc. International Conference on Parallel Processing, Volume II*, 1993.

[13] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3:179–193, 1992.

[14] M. W. Hall, M. W. Hall, S. Hiranandani, S. Hiranandani, K. Kennedy, K. Kennedy, C. wen Tseng, and C. wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35:66–80, 1992.

[15] M. J. Harrold and N. Ci. Reuse-driven interprocedural slicing. In *In Proceedings of the 20th International Conference on Software Engineering*, 1998.

[16] High Performance Fortran. http://www.netlib.org/hpf/

[17] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *ACM Transactions on Programming Languages and Systems*, 12:26–60, 1990.

[18] K. Kennedy and K. Kennedy. Combining dependence and data-flow analyses to optimize communication. In *Proceedings of the 9th International Parallel Processing Symposium*, 1995.

[19] K. Kennedy and A. Sethi. Resource-based communication placement analysis. In *Proceedings of the 9th Workshop on Language and Compilers for Parallel Computing*, 1996.

[20] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, 2001.

[21] D. Liang and M. J. Harrold. Reuse-driven interprocedural slicing in the presence of pointers and recursion. In *Proceedings of International Conference on Software Maintenance*, 1999.

[22] MPI Standard. http://www-unix.mcs.anl.gov/mpi/

[23] The Omega Project. http://www.cs.umd.edu/projects/omega/

[24] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *Proceedings of CGO,* pp. 144–156, 2007.

[25] E. Rosser and W. Pugh. Iteration space slicing and its application to communication optimization. In *Proceedings of the International Conference on Supercomputing,*,1997.

[26] T. Reps and W. Yang. The semantics of program slicing. Technical Report, University of Wisconsin, 1988.

[27] http://www.virtutech.com/

[28] http://suif.stanford.edu/

[29] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[30] S. Verdoolaege, K. Beyls, M. Bruynooghe, and F. Catthoor. Experiences with Enumeration of Integer Projections of Parametric Polytopes. In *Proceedings of the Compiler Construction Symposium,* pp. 91–105, 2005.

[31] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, 1981.

[32] X. Zhang and R. Gupta. Cost effective program slicing. In *Proceedings of the Programming Languages Design and Implementation*, 2004.

[33] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pp.112–125, 1993.

[34] M. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distrib. Syst. 2,* 4:452–471,1991.

[35] K. Bondalapati. Parallelizing DSP nested loops on reconfigurable architectures using data context switching. In *Proc. of the 38th Design Automation Conference*, pp.273–276, 2001.

[36] G. Goumas, N. Drosinos, M. Athanasaki and N. Koziris. Automatic parallel code generation for tiled nested loops. In *Proc. of the ACM Symposium on Applied Computing*, pp. 1412–1419, 2004.

[37] B. Mei, S. Vernalde, D. Verkest, H. Man and R. Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *Proc. of the Conference on Design, Automation and Test in Europe*, pp. 10296–10301, 2003.

[38] A. Navarro, E. Zapata and D. Padua. Compiler techniques for the distribution of data and computation. *IEEE Transactions on Parallel Distributed Systems 14,* 6:545–562, 2003.

[39] K. Hogstedt, L. Carter and J. Ferrante. On the parallel execution time of tiled loops. *IEEE Transactions on Parallel Distributed Systems 14,* 3:307–321, 2003.