Effective Automatic Data Allocation for Parallelization of Affine Loop Nests

IISc-CSA-TR-2014-2
March 2014

Chandan Reddy
Indian Institute of Science
chandan.g@csa.iisc.ernet.in

Uday Bondhugula
Indian Institute of Science
uday@csa.iisc.ernet.in

ABSTRACT

This paper proposes techniques for data allocation and computation mapping when compiling affine loop nest sequences for distributed-memory clusters. Techniques for transformation and detection of parallelism, and generation of communication sets relying on the polyhedral framework already exist. However, these recent approaches used a simple strategy to map computation to nodes – typically block or block-cyclic. These mappings may lead to excess communication volume for multiple loop nests. In addition, the data allocation strategy used did not permit efficient weak scaling. We address these complementary problems by proposing automatic techniques to determine computation placements for identified parallelism and allocation of data. Our approach for data allocation is driven by tiling of data spaces along with a scheme to allocate and deallocate tiles on-demand and reuse them. We show that our approach for computation mapping is able to come up with more effective mappings than those that can be developed using vendor-supplied libraries. Experimental results on some sequences of BLAS calls demonstrate a mean speedup of 1.82× over versions written with ScaLAPACK. Besides enabling weak scaling for distributed memory, data tiling also improves locality for shared-memory parallelization. Experimental results on a 32-core shared-memory SMP system shows a mean speedup of 2.67× over code that is not data tiled.

1. INTRODUCTION AND MOTIVATION

A significant amount of work has been done in the past two decades on parallelizing for distributed-memory. A majority of this work was done in developing compiler technology for high performance Fortran. However, even in domains where it was suitable, namely programs with regular accesses, there was limited success. Several steps involved in achieving good performance remained manual. The quality of communication code as well as the ability to automatically apply complex transformations was a big limitation. Hence, even for programs that involve regular accesses such as sequences of linear algebra kernels, the approach currently used to obtain the best performance is to rely on highly tuned libraries. In addition, none of the previous approaches on automatic distributed-memory parallelization and code generation have been directly employed so far even in domain-specific language compilation. MPI still happens to be the dominant and de facto programming model due to the lack of any compiler support. The objective of our paper is to further improve automatic compiler and runtime support for distributed-memory clusters of multicores with emphasis of exploiting locality.

Some of the limitations in parallelization and code generation for regular programs, in particular, affine loop nests, have been addressed in recent years [3, 5, 4, 8]. These works provide techniques for transformation and detection of parallelism, and generation of communication sets relying on the polyhedral framework. However, these works use a simple strategy to map identified parallelism – typically block or block-cyclic. Previous automatic data distribution works [12, 6, 11, 13] also employed only block or block-cyclic mappings for loop nests with the possibility to re-distribute in between. In spite of a good choice of loop transformations and structure, these strategies to map identified parallelism significantly impact communication volume and load balance. Some specialized mappings such as multipartitioning [15, 7] were known and implemented in dHPF, but these works did not provide any automatic way to determine such mappings. In addition to this, there has been significant room for improvement in the way data allocation was handled – to better exploit locality in conjunction with compute transformations. This paper provides an effective solution to these missing steps.

Our approach for data allocation works by tiling of data spaces. A *data tile* is the granularity at which data is allocated and it is itself contiguous in main memory. Data local to a node as well as that which is received from remote nodes is accessed by first addressing a data tile and then indexing into it. A compiler-based approach with light-weight runtime helper functions handles on-demand allocation and deallocation of tiles, and their reuse. The approach can work in conjunction with either static or dynamic scheduling of compute tiles. Besides enabling weak scaling for distributed memory, data tiling improves locality for shared-memory parallelization – by reducing cache conflict misses, data TLB misses, and false sharing, and allowing better prefetching.

Although manual distributed-memory parallelization as seen by a programmer often starts with the step of data decomposition followed by computation decomposition, we show that this seemingly natural approach is not the efficient one when designing flexible and automatic compiler support. We argue that emphasis should first be placed on determining the right computation transformation and a placement. If good computation distributions are found, the initial data distribution only impacts "first-read" and "last write" communication. Determining a data distribution and then a compute distribution as done by some previous approaches may even prevent certain computation distributions where the owner of data has to change in order to exploit locality. Our approach neither has the notion of an owner for data, nor that of fixed distribution, nor re-distribution. Instead data that is accessed for a piece of computation is allocated on demand (if not already allocated), with communication data flowing from one node to another as dictated by computation mappings. In summary, the contributions of our work are that of:

- developing a technique to map identified parallelism after transformation encompassing block, block-cyclic and other specialized and arbitrary mappings,
- devising a data allocation technique based on data tiling to provide improved locality and enable weak scaling for distributed memory parallelization,
- and demonstration through experiments that our approach is significantly better than previous approaches and the code generated outperforms that which can be written even using vendor supplied BLAS libraries.

More specifically, for some sequences of BLAS calls, the code we automatically generate beats code manually written using Intel ScaLAPACK library by a mean factor of $1.82\times$ while running on a 32-node InfiniBand cluster of multicores. Shared-memory parallelization results obtained on a 32-core shared-memory NUMA SMP system show a mean speedup of $2.67\times$ over code that is not data tiled.

The rest of this paper is organized as follows. Section 2 describes our approach to find computation placements. Section 3 describes how data tilings are found. We describe in Section 4 how allocation is performed based on a tiled view of data spaces. Experimental results are presented in Section 5. Related work and conclusions are presented in Section 6 and 7 respectively.

2. DETERMINING COMPUTATION MAPPINGS

In this section, we describe how we find a suitable way of mapping available parallelism to a set of nodes. In particular, the presented strategy subsumes commonly used distributions like block, block-cyclic as well as more complex mapping schemes. The mappings are obtained for all parallel loop nests together. In the rest of this paper, the term *node* is used to refer to a set of processors that have shared memory, typically, an SMP system of general-purpose multicores. Multiple nodes are connected over an network interconnect.

To be self-contained, we briefly describe certain distribution patterns – although some of them are very well-known. A *block* distribution distributes a set of iterations into equal or nearly equal contiguous chunks where the number of chunks is equal to the number of processors. A *cyclic* distribution distributes a set of iterations across processors in a round-robin manner at the granularity of a single iteration. When this granularity is changed to a contiguous chunk of some fixed size, the resulting mapping is a *block-cyclic* mapping. We define another specialized mapping that we call a *sudoku* mapping due to its similarity with the popular number placement puzzle of the same name. A *sudoku* mapping assigns tiles from an n-dimensional view to processors in a way such that all tiles along any of the $(n-1$ dimensional) canonical hyperplanes are mapped on to distinct processors. We will see that such a mapping (Figure 5a) has interesting properties in minimizing communication if two pieces of computation require an array to be distributed in conflicting ways. Multipartitioning [15] implemented in dHPF is one possible perfect sudoku mapping and such a mapping was used in the manually parallelized versions of NAS BT and SP.

The computation mapping of a statement S_i, denoted by π_{S_i}, maps computation tiles to nodes. The chosen computation mappings have a significant impact on the execution time of a program. Two key factors to be considered while deciding computation mappings are communication volume and load balance. We call a computation mapping for a given program optimal if it leads to the lowest communication volume and perfect load balance. Consider the sample tiled ADI code shown in Figure 2. The optimal computation mapping π_{S1} for the forward x sweep loop is the block distribution along ii loop iterations. This distribution leads to no communication and all nodes gets equal number of iterations. Similarly, the optimal mapping π_{S2} for the y sweep is the block distribution of jj. However, these mappings are not optimal for the entire program as they demand a transpose of array X between the nests of $S1$ and $S2$, and thus a large amount of communication. Significantly better mappings exist and in this section, we will describe a technique to find such computation mappings automatically.

```
//forward x sweep
for (i=0; i<N; i++)
 for (j=1; j<N; j++)
  X[i][j] = X[i][j] - X[i][j-1] * A[i][j] / B[i][j-1]; //S1

//upward y sweep
for (j=0; j<N; j++)
 for (i=1; i<N; i++)
  X[i][j] = X[i][j] - X[i-1][j] * A[i][j] / B[i-1][j]; //S2
```

Figure 1: Sample ADI program with only forward x and y sweeps.

We model the problem of finding optimal computation mappings as a graph partitioning problem on the inter-tile communication graph (TCG). Each vertex in the TCG represents a computation tile of the program. An edge e is added between two vertices if and only if there is communication between the corresponding two tiles when they are executed on different nodes. The weight of the edge e_w will be equal to the communication volume between the two tiles. Finding the optimal computation mappings is equivalent to partitioning the TCG into p (number of nodes) equal size partitions with the objective to minimize the sum of those edge weights that straddle partitions. This objective function represents the total communication volume for the entire program execution. The resulting computation mappings will thus have lower communication volume.

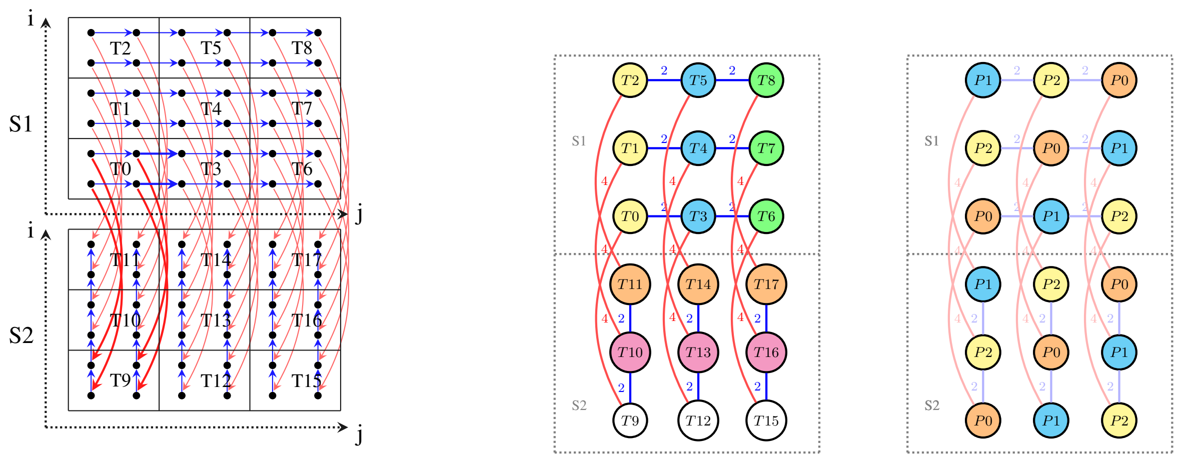
```
//forward x sweep
for (jj=0; jj<floord(N, 128); jj++) //serial loop
 for (ii=0; ii<floord(N, 128); ii++) //parallel loop
  for (i=max(1, ii*128); i<min(ii*128+127, N); i++)
   for (j=max(1,jj*128); j<min(jj*128+127, N); j++)
    X[i][j] = X[i][j] - X[i][j-1] * A[i][j] / B[i][j-1]; //S1

//upward y sweep
for (ii=0; ii<floord(N, 128); ii++) //serial loop
 for (jj=0; jj<floord(N, 128); jj++) //parallel loop
  for (j=max(1, jj*128); j<min(jj*128+127, N); j++)
   for (i=max(1, ii*128); i<min(ii*128+127, N); i++)
    X[i][j] = X[i][j] - X[i-1][j] * A[i][j] / B[i-1][j]; //S2
```

Figure 2: Tiled ADI program, tile size = 128

Figure 3a illustrates the tiled iteration domain along with RAW dependences for the ADI example. A vertex is added to TCG for each of the tiles. Previous work [8] describes techniques (FOIFI) to determine the communication sets and receiving tiles for a given tile. Dependence edges that cross tile boundaries are used to determine the necessary communication sets and receiving tiles. We use these techniques to build the TCG. For a given tile, an edge is added to each of its receiving tiles. The size of the communication set between sender and receiver tiles is set as the weight of edge between them. In the Figure 3b, edges are added from $T0$ to its receivers $T3$ and $T9$, with the edge weights 2 and 4 respectively.

2.1 Load balancing constraints

A parallel phase is a contiguous band of parallel loops/dimensions that have been identified for potential extraction of parallelism. Programs often consist of multiple parallel phases. To achieve good load balance, it is essential that an equal or a nearly



(a) Tiled iteration space with dependences **(b)** Inter-tile communication graph **(c)** Computation distribution

Figure 3: ADI

equal number of tiles are allocated to all nodes in each parallel phase. We identify the tiles that belong to a parallel phase and add constraints that will minimize load imbalance within each parallel phase. Vertex weights are used to distinguish between tiles that belong to different parallel phases. The vertex weight is a vector of size equal to the total number of parallel phases. All tiles which belong to the i^{th} parallel phase will have a vertex weight with the i^{th} component set and rest zero. Let S_i be the sum of the i^{th} vertex weight component of all vertices belonging to a single partition. For each vertex weight component, we add load balancing constraints that minimizes the difference between S_is of any two partitions. These constraints make sure that the resulting compute tile mappings will assign an equal or a nearly equal number of tiles to each node in each parallel phase.

When performing static scheduling, all tiles belonging to a band of tiled parallel loops, for a given value of surrounding sequential loops, are said to belong to a single parallel phase. Figure 2 shows the tiled code for the ADI example. For the first loop, ii is the innermost tiled parallel loop. All the iterations of ii, for a particular value of outer sequential loop jj, belong to the same parallel phase. In case of dynamic scheduling, we do a topological sort of the TCG to identify tiles that belong to the same parallel phase – all tiles at the same level belong to a single phase. For the ADI example in Figure 3b there are six parallel phases. All tiles belonging to the same parallel phase are marked with the same color. Tiles $T0$, $T1$ and $T2$ belong to same parallel phase and each will have vertex weights as $\langle 1,0,0,0,0,0\rangle$. Similarly $T9$, $T12$ and $T15$ belong to same parallel phase and will have vertex weight as $\langle 0,0,0,1,0,0\rangle$. The load balancing constraints ensures that Tiles $T0$, $T1$ and $T2$ are equally divided among all nodes.

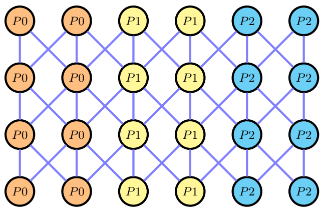
The above formulation finds partitions that have an equal number of tiles in each partition. If number of iterations in the tiles are not equal, assigning equal number of tiles to each partition could lead to load imbalance. To overcome this issue, we set number of iteration in the tile as the vertex weight component. Load balancing constraints on vertex weights ensures that each partition will have an equal number of iterations, in each parallel phase.

Figure 3c shows one of the optimal solutions for graph partitioning of the ADI example. In each parallel phase, equal number of tiles are assigned to all the nodes, which ensures perfect load balance. Computation mapping is identical for both $S1$ and $S2$. Computation tile $\langle ii,jj\rangle$ of $S1$ and $\langle ii,jj\rangle$ of $S2$ are mapped to the same node, thus eliminating communication between $S1$ and $S2$. We call this type of computation mapping as 'sudoku' distribution, since all the node are assigned equal number of tiles in each row and column. Sudoku distribution is very similar to multi-partitioning mappings [15]. Sudoku mappings may not have modulo shift mappings of nodes as defined in multi-partitioning mappings. Figure 4a shows the computation mappings for stencil programs with near-neighbor communication, which is exactly the block distribution. Figure 4b shows the obtained computation mappings for tapered iteration spaces. This mapping is slightly different from block-cyclic mapping. In this mappings, first and last columns are mapped to node $P0$, where as in block-cyclic mappings first and fourth columns will be mapped to node $P0$. The obtained mappings has slightly better load balance than block-cyclic distribution, as equal number of tiles are allocated to all the nodes.

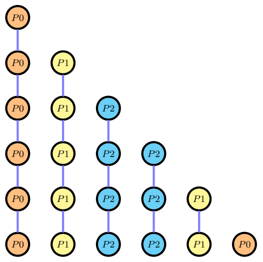
The graph partitioning solution of the TCG also determines the optimal dimensionality of the computation mapping. Note that a higher dimensional mapping is not necessarily optimal for an entire sequence of loop nests being optimized. Consider a program with the first loop nest (forward x sweep) of ADI. The TCG of this program contains only the upper half of Figure 3b with just nodes of $S1$. The optimal computation mapping for this graph is a 1-d block distribution of ii loop. Similarly, for the lower half with the second loop nest (upward y sweep), the optimal computation mapping is a 1-d block distribution of the jj loop. Our graph partitioning approach is able to find these solutions.

2.2 Scalability of graph partitioning

Graph partitioning with load balancing constraints is an NP-hard problem. Solutions to these problems are generally derived using heuristics and approximation algorithms. Open-source software packages such as METIS [16] and SCOTCH [19] can be used to solve graph partitioning problems. As the problem size increases, the number of vertices and edges in the graph also increase. The number of load balancing constraints also increases as we add load balancing constraints to each parallel phase, and this depends on the problem size. Even state-of-the-art graph partitioning software such as these do not scale as the problem size increases. For example, to partition a TCG of ADI with 64 vertices into 4 partitions, METIS takes around 240s. If the problem size is increased further, both the time taken and the memory required for partitioning increases drastically. Another major problem is that the quality of

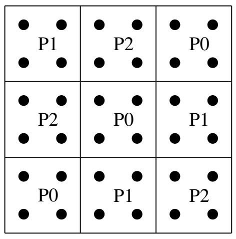


(a) computations with near-neighbor communication

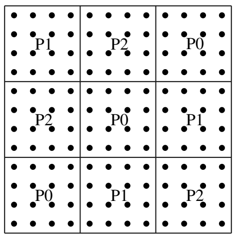


(b) Unbalanced computation

Figure 4: Computation distributions



(a) Representative mapping



(b) Scaled mapping (dot is tile)

Figure 5: Scaling a computation mapping for ADI

the obtained solution degrades as the problem size increases. For the ADI example, we observe that perfect "sudoku" mappings are not obtained for more than 32 vertices.

In order to make our approach scalable for larger problem sizes, we use an approximation to partition the TCG. Note that the dependence patterns typically do not change as the problem size is increased beyond a certain point. Hence, the optimal mappings for a larger problem size can often be obtained by scaling the optimal mappings for a smaller one. The computation mappings for larger problem sizes and the actual number of processors are derived from the computation mappings for a smaller problem size and number of processors. At compile time, we build the TCG for a smaller problem size. Problem sizes are chosen such that the number of vertices in each parallel phase is a particular number that is sufficient to distinguish the nature of the obtained mapping. The number of processors is fixed at four which we found to be sufficient in practice, and the problem size is set so that we have at least two times the number of processor tiles along each parallel dimension. This allows us to distinguish between block and block-cyclic mappings. The edges weights and vertex weights are computed for this smaller graph, and this is partitioned using METIS. This step is very fast owing to a very small graph. At runtime, when the problem size is known, the partitioning solution for the input problem size is obtained from the solution of the smaller representative graph. The mapping obtained is first classified as either being block, block-cyclic, sudoku or an arbitrary one. If a mapping is identified as block, sudoku, or arbitrary, then we perform a "block" scaling of the mapping for the right problem size and the number of processors. This is illustrated in Figure 5 which shows the computation mapping obtained for larger problems sizes for the ADI example. We then generate a function that returns the correct mapping for any given number of nodes.

For block and sudoku mappings, block scaling ensures that the respective property continues to hold. For arbitrary mappings, we find the block scaling to be a reasonable approach though we have not seen cases where we found arbitrary mappings. When the solution obtained through graph partitioning corresponds to a cyclic or a block-cyclic mapping, we perform a cyclic scaling analogous to the block scaling described above. Overall, this approximate approach of using a representative graph and then scaling the solution analytically based on an identified template mapping does not take more than a second on any of the examples considered for evaluation. Actual communication costs finally depend on network topology – finding computation mappings that are optimal for a given network topology is beyond the scope of this work and is left for future.

3. DATA TILING

As introduced earlier, the idea is to tile the data space similar to tiling an iteration space, compute data required by a compute tile at the granularity of data tiles, and allocate only the required data tiles on-demand at a node. A node itself could comprise multiple cores that share memory.

3.1 Finding data tiling hyperplanes

In this section, we describe techniques to find the shape of the computation and data tiles. We determine the shape of computation and data tiles such that the data accessed by a computation tile is packed into as few data tiles as possible.

Let $S_1, S_2, \ldots, S_n$ be the statements of a program. Let D_{S_j} be the domain of S_j. Let E be the set of dependences edges with each edge $e \in E$ characterized by a dependence polyhedron P_e. P_e is a set of linear constraints that relate source iterators and target iterators that are in dependence. A one-dimensional affine transformation for S_j, denoted by ϕ_{S_j}, is defined as

$$\phi_{S_j}(\vec{i}) = \vec{h}^{S_j} \cdot \vec{i} + h_0^{S_j}. \tag{1}$$

ϕ_{S_j} can be viewed as a function mapping iterations of S_j to numbers that represent virtual processor ids. For example, $\phi_{S_j}(\vec{i}) = (1,0)^T.\vec{i} + 0$ maps all iterations (i,j) to virtual processor i. ϕ_{S_j} partitions the iteration space of S_j, and $\vec{h} = (1,0)$ represents the orientation of the hyperplane that partitions it. When ϕ_{S_j} satisfies certain properties, we call it a tiling hyperplane.

Similarly, for arrays we define an array mapping function ψ_{A_k} that maps array elements to virtual processors represented by

$$\psi_{A_k}(\vec{a}) = \vec{d}^{A_k} \cdot \vec{a} + d_0^{A_k}, \tag{2}$$

where $\vec{d}$ represents the orientation of the hyperplane that partitions the array space, d_0 is the constant offset, and $\vec{a}$ is a data element in A_k. We call these mappings data tiling hyperplanes if they are found to satisfy certain properties that we will describe later in this section.

Consider the first loop nest of the ADI example shown in Figure 1. Assume that the computation mapping for $S1$ is $\phi_{S_1}(\vec{i}) = (1,0)^T \cdot \vec{i}$. Different iterations of the i loop will be mapped to different virtual processors. For the array access $X[i][j]$, different iterations of i loop access different rows of array X. Hence, the first dimension of X has to be partitioned. This corresponds to the array mapping $\psi_X(\vec{a}) = (1,0)^T \cdot \vec{a}$, which in turn corresponds to a row distribution of X. If the array access had been $X[j][i]$, then for $\phi_{S_1}(\vec{i}) = (1,0)^T \cdot \vec{i}$, the corresponding data mapping would be $\psi_X(\vec{a}) = (0,1)^T \cdot \vec{a}$, i.e., a column-wise distribution of X. Hence, the choice of data partitioning hyperplanes, for a given computation mapping, is driven by the array accesses.

Let $F^i_{s_j,A_k}$ be the i^{th} access function of array A_k in statement S_j. In our model, F is an affine function of loop iterators and program parameters. $\phi_{s_j}(\vec{i})$ is the virtual processor to which iteration $\vec{i}$ will be mapped. $F^i_{s_j,A_k}(\vec{i})$ represents data accessed by $\vec{i}$. $\psi_{A_k}(F^i_{s_j,A_k}(\vec{i}))$ is the virtual processor to which the data accessed by $\vec{i}$ will be mapped. Now, we require that the data accessed by $\vec{i}$ be mapped to the same virtual processor as the one $\vec{i}$ is mapped to, i.e.,

$$\phi_{s_j}(\vec{i}) = \psi_{A_k}(F^i_{s_j,A_k}(\vec{i})). \quad (3)$$

The above condition is conceptually the same as that used by Anderson and Lam [2], but it was in a form that worked only for perfect loop nests and uniform dependences, and hence the subsequent approach relying on it was also different. It is not obviously always possible to ensure condition (3) since multiple iterations can access the same data. Hence, what we try to capture is the difference between the LHS and RHS of (3) as follows and try to minimize it:

$$\gamma^i_{s_j,A_k}(\vec{i}) = |\phi_{s_j}(\vec{i}) - \psi_{A_k}(F^i_{s_j,A_k}(\vec{i}))| \;\; \vec{i} \in D_{s_j}. \quad (4)$$

The above definition is thus used to connect compute and data tiling hyperplanes. We now first describe the existing technique to characterize and choose from valid compute tiling hyperplanes, and then show how data tiling constraints and objectives are integrated into it to minimize γs. Previous work [5] provided an automatic approach to find compute tiling hyperplanes that exposed maximal course-grained parallelism and locality based on an Integer Linear Programming formulation. To enforce validity for tiling for an edge e with dependence polyhedron P_e, the following constraint ensures non-negative dependence components that is sufficient for tiling:

$$\phi_{s_j}(\vec{t}) - \phi_{s_i}(\vec{s}) \geq 0, \;\; \langle \vec{s}, \vec{t} \rangle \in P_e. \quad (5)$$

Then, the following cost function has been used to select the best hyperplane among the set of valid tiling hyperplanes.

$$\delta_e(\vec{t}, \vec{s}) = \phi_{s_j}(\vec{t}) - \phi_{s_i}(\vec{s}), \;\; \langle \vec{s}, \vec{t} \rangle \in P_e \quad (6)$$

This cost function is a measure of communication volume or reuse distance – and the ILP is solved to minimize it.

We now add data tiling hyperplane coefficients to the ILP described above and use (4) to relate compute and data tiling hyperplane coefficients. (4) cannot be directly added as it leads to non-linear constraints between hyperplane coefficients and loop iterators. The bounding function technique [10] can again be used here obtain constraints in a linear form. When the iteration spaces are bounded, one can obtain an upper bound on $\gamma^i_{s_j,A_k}(\vec{i})$. The maximum mismatch quantified by γ occurs when the iterations and the entire data accessed by them are mapped onto different virtual processors. This mismatch can be bounded by an affine function of program parameters $\vec{p}$, i.e., there exists $v_{A_k}(\vec{p}) = u_{A_k}.\vec{p} + w$ such that

$$v_{A_k}(\vec{p}) - \gamma^i_{s_j,A_k}(\vec{i}) \geq 0, \;\; \vec{i} \in D_{s_j} \quad (7)$$

By minimizing the bounding function coefficients $\vec{u}_{A_k}$, we indirectly minimize (4). Now the affine form of the Farkas lemma can be applied to (7).

$$v_{A_k}(\vec{p}) - \gamma^i_{s_j,A_k}(\vec{i}) \equiv \lambda_{s_j,A^i_{k0}} + \Sigma_t \lambda_{s_j,A^i_{kt}}(a_t\vec{i} + b_t), \quad (8)$$

where $\lambda_{S_j,A^i_{kt}} \geq 0$ are the Farkas multipliers and $a_t\vec{i} + b_t \geq 0$ are the faces of D_{s_j}. The coefficients of $\vec{i}$ and $\vec{p}$ in the resulting equations are eliminated using Fourier-Motzkin elimination to obtain constraints in a linear form.

The resulting ILP system with tile validity conditions (5), cost function constraints (6) and data hyperplane constraints (7) is solved using PIP [9] to get both compute and data tiling hyperplanes. PIP computes the lexicographical minimal solution for the ILP. Hence, the order of variables is important. We add separate bounding function coefficients for each of the arrays to ensure that non-zero bounding coefficients of one array do not affect the choice of data hyperplanes for the other arrays. If there is a mismatch between computation and data tiling hyperplanes, then the bounding function coefficients of (7) will be non-zero. If same bounding function coefficients are used for all arrays, then the mismatch between computation and data hyperplanes for a single array may lead to sub-optimal data tiling hyperplane for other arrays. Let u_s, w_s be the bounding function coefficients from the (6), u_{A_k}, w_{A_k} be the bounding function coefficients for array A_j resulting from (7), and h_{s_j} be the vector of compute hyperplane coefficients, and d_{A_k} that of the data tiling hyperplane coefficients. (9) shows the order of variables used for the lexicographic minimal solution:

$$min_{\prec}(u_s, u_{A_1}, \ldots, w_s, w_{A_1}, \ldots, h_{s_1}, \ldots, d_{A_1}, \ldots) \quad (9)$$

Consider the example shown in Figure 6. For compute tiling hyperplane (1,0), there is no need to tile array $v1$ and $v2$, as all iterations of i loop access entire $v1$ and $v2$ arrays. Once we solve the ILP, we obtain a compute tiling hyperplane for each of the statements, ϕ_{S_j}, and a data tiling hyperplane for each of the arrays, ψ_{A_k}. A data tiling hyperplane ψ_{A_k} is considered to be invalid for all $F^i_{S_j,A_k}$ if the ϕ_{S_j} lies in the union of the null spaces of all array access functions, $F^i_{S_j,A_k}$. If this condition is satisfied, then all iterations of S_j access the entire array A_k. Hence, it is not necessary to tile the array space even if the iteration space of S_j is tiled with ϕ_{S_j}. It is necessary to tile the data space only when different partitions of the iteration space, due to ϕ_{S_j}, access different parts of array A_k.

```
for (i=0; i<N; i++)
  for (j=1; j<N; j++)
    B[i][j] = A[i][j] + u1[i]*v1[j] + u2[i]*v2[j];
```

Figure 6: First loop nest of gemver

3.2 Iteratively finding all hyperplanes

A statement can have as many compute tiling hyperplanes as the dimensionality of its iteration space. Similarly, an array can have as many data tiling hyperplanes as its dimensionality. Solving the ILP formulation described in the previous section gives us a single compute tiling hyperplane for all statements and a single data tiling hyperplanes for all arrays. We add new constraints to the ILP to ensure that subsequent compute tiling hyperplanes are linearly independent of ones already found. However, for data tiling hyperplanes, linear independence constraints are only added with respect those that were found to be valid. Algorithm 1 describes the complete procedure to find all compute and data tiling hyperplanes. For the ADI example 1, the first compute hyperplane for $S1$ and $S2$ is (1,0), corresponding data tiling hyperplane for arrays X, A and B is (1,0). The second compute hyperplane for $S1$ and $S2$ is (0,1), corresponding data tiling hyperplane for arrays X, A and B is (0,1). Since, each of the data tiling hyperplanes are linearly independent of each other, they form a full rank matrix.

4. DATA ALLOCATION

In this section, we describe how data is indexed and managed once data tiling hyperplanes for each array have been determined. The data accessed for array A_k through access function F in S_j can be computed by taking the image of the D_{S_j} under F. However, we are interested in computing data accessed by a particular

Algorithm 1: Finding compute and data hyperplanes

```
   Input: Data dependences ($E$), array access functions ($\mathbf{F}$)
   Output: $\phi_{S_j}\ \forall S_j$, $\psi_{A_k}\ \forall A_k$
1  $C \leftarrow \emptyset$;
   valid_hyperplanes_$S_j \leftarrow \emptyset\ \ \forall S_j$;
   valid_hyperplanes_$A_k \leftarrow \emptyset\ \ \forall A_k$;
   num_valid_hyperplanes_$S_j \leftarrow 0\ \ \forall S_j$;
   num_valid_hyperplanes_$A_k \leftarrow 0\ \ \forall A_k$;
   max_num_hyperplanes $\leftarrow \max(dim(S_j))\ \forall S_j$;
   for each $e \in E$ within fused loops do
2      Add validity constraints resulting from $\phi_{s_j}(\vec{t}) - \phi_{s_i}(\vec{s}) \geq 0$ to $C$
3  for each $e \in E$ do
4      Add the bounding function constraints resulting from
       $|v(\vec{p}) - \delta_e(\vec{t}, \vec{s})| \geq 0$ to $C$;
5  for each $F^i_{s_j,A_k} \in \mathbf{F}$ do
6      Obtain constraints resulting from $\phi_{s_j}(\vec{i}) - \psi_{A_k}(F^i_{s_j,A_k}(\vec{i})) \geq 0$
       and $\psi_{A_k}(F^i_{s_j,A_k}(\vec{i})) - \phi_{s_j}(\vec{i}) \geq 0$ to $C$;
7  while max_num_hyperplanes $\geq 0$ do
8      Solve the ILP with constraints in $C$;
       max_num_hyperplanes $\leftarrow$ max_num_hyperplanes - 1;
       for each $\phi_{S_j}$ found do
9          if num_valid_hyperplanes_$S_j <$ dim($S_j$) then
10             Add $\phi_{S_j}$ to valid_hyperplanes_$S_j$;
               num_valid_hyperplanes_$S_j$++;
               Add constraints to exclude hyperplanes linearly
               dependent on $\phi_{S_j}$;
11     for each $\psi_{A_k}$ found do
12         if $\psi_{A_k}$ is a valid data tiling hyperplane then
13             if num_valid_hyperplanes_$A_k$ <dim($A_k$) then
14                 Add $\psi_{A_k}$ to valid_hyperplanes_$A_k$;
                   num_valid_hyperplanes_$A_k$ ++;
                   Add constraints to exclude hyperplanes linearly
                   dependent on $\psi_{A_k}$;
```

compute tile. This enables us to allocate only the data required for the tile on the node it executes on. To accomplish this, the image of the access function is computed while treating dimensions outer to the tile, that we call inter-tile iterators, as parameters. The resulting image will be a set, parametric in the inter tile iterators. By plugging in a particular value for these inter tile iterators, precise data accessed by that particular compute tile can be obtained. The shaded rectangle in Figure 7 shows the parametric data region of compute tile (1,0) due to the array access $A[i][j-1]$.

When the data tiling hyperplanes are used, the parametric data regions obtained above end up getting tiled as well in the same way iteration spaces are tiled. Same tile sizes are used when tiling the iteration space using compute tiling hyperplanes and the parametric data region using the corresponding data tiling hyperplane. After tiling, the dimensions of the parametric data region include the newly added inter data tile iterators, intra data tile iterators and the inter compute tile iterators. For a particular value of compute tile iterators, inter data tile iterators enumerate all data tiles accessed, and intra data tile iterators scan data points inside a data tile.

On-Demand data tile memory allocation: Projecting out the inter data tile iterators, we get parametric polyhedron that can be used to enumerate all the data tiles that a compute tile accesses. We use this polyhedron to generate a function that will allocate memory for the data tiles required by a given compute tile. This function will be called just before the execution of a compute tile. This will ensure that data required by a compute tile is allocated only on that node which will execute the compute tile. For the ADI example generated function returns data tiles $D0$ and $D1$ for the compute tile (0,1), and only $D0$ for the compute tile (0,0).

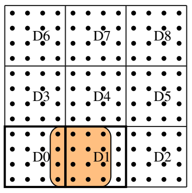


Figure 7: Accessed data for compute tile (0,1) for $A[i][j-1]$ in ADI

Algorithm 2: Determine accessed data tiles for array A_k

```
   Input: Parametric data region $D_{A_k}$ of array $A_k$, data tiling
          hyperplanes $\psi_{A_k}$, tile sizes $\tau_k$
   Output: Parametric data tiles accessed
1  for each data tiling hyperplane $\psi_{A_k}(\vec{a}) = \vec{d_k} \cdot \vec{a} + d_{k0}$, tile size $\tau_k$
   do
2      add a inter data tile dimension $\vec{a_T}$, corresponding to $\vec{a}$
       add the following two constraints to $D_{A_k}$
       $\tau_k * (\vec{d_k} \cdot \vec{a_T}) \leq \vec{d_k} \cdot \vec{a} + d_{k0} \leq \tau_k * (\vec{d_k} \cdot \vec{a_T}) + \tau_k - 1$
3  Project out the intra-tile data dimensions $\vec{a}$ in $D_{A_k}$
   return $D_{A_k}$
```

Allocation of first-read data: Input data of the program being compiled has to be initialized and distributed before start of program execution. We thus also need to allocate that part of the input data that is “live in” to the compute tiles to be executed on that node. We call this the first-read data. First-read data of an array A is set of all array elements whose values are first read by a compute tile before a write is performed if at all. This set is identified by computing the data accessed all read accesses, and then subtracting out data accessed by target iterations of RAR and RAW dependences entering the tile.

Data tile buffer reuse: Programs access different parts of the array during the entire execution of the program. Often, we can reuse the data tile buffers, rather than allocating new buffers for every new data tile. A data tile buffer can be safely reused when all the compute tiles that require this data tile have finished their execution. We can precisely count the number of compute tiles that require a given data tile. Per data tile ref-count is used to capture number of compute tiles that need this data tile. We generate a function that will enumerate all the compute tiles executed by the given processor and use Algorithm 2 to get all the data tiles accessed by tile, and increment their ref-count. This function is invoked at the start of the program. Once the compute tile has finished its execution and data required by other tiles is packed, we decrement the ref-count of all the data tiles accessed by this compute tile. If the ref-count becomes zero we add the data tile buffer pointer to free-buffers queue. When we want to allocate a new buffer for data tile, the free-buffer queue is checked fist, if it is non empty, one of its buffers is returned. A new allocation is done only when free-buffer queue is empty. Per data tile ref-count is used to track the liveness information of data tiles. Since, we use dynamic scheduling, actual schedule will be decided at runtime. Above techniques provide an efficient, dynamic and schedule independent mechanism for data tile buffer reuse.

Data tiling with dynamic scheduling: We have integrated data tiling with a dynamic scheduling framework which schedules at the granularity of compute tiles. First, we determine the computation mappings π_S for a given problem size and the number of nodes. The dynamic scheduling runtime distributes compute tiles according to π_S. The read-in data is allocated and data tile reference counts are initialized as per π_S mappings. All compute tiles that are mapped to a single node are dynamically scheduled. Be-

Algorithm 3: initialize_ ref_counts ()

```
  Input: Set of all compute tiles
  Output: data tile ref-counts
1 for each data tile $\vec{d}$ do
2     ref_count_ $\vec{d} \leftarrow 0$;
3 for each compute tile $\vec{t}$ do
4     if $\pi(\vec{t}) = node\_id$ then
5         required_data_tiles ← determine required data tiles ($\vec{t}$);
          for each data tile $\vec{d} \in required\_data\_tiles$ do
6             ref_count_$\vec{d}$++;
```

fore start of tile execution, we call the on-demand allocate function (Algorithm 2), which will allocate all required data tiles if they had not been already allocated. After the tile has finished execution, we call the pack function which packs data required by other nodes from the current node. Packed data is sent to its receive nodes using asynchronous MPI primitives. Once the pack function is finished, we decrement the ref-counts of data tiles used. Besides being called at schedule time, the on-demand allocate function is also called before data received from other nodes is unpacked. A concurrent queue is used to maintain a list of free data tile buffers. Atomic increment and decrement are used to modify data tile ref-counts, and a compare-and-swap to update data tile buffer pointers. Thus, the whole implementation is thread-safe and lock-free.

4.1 Re-indexing data spaces

After we perform data tiling transformation, the memory layout of the arrays is changed. Array elements within the data tile are now packed in contiguous memory locations. So we need to modify the original array access functions such that they access correct elements in the new memory layout. The dimensionality of array A is double because of the new tile dimensions that are added. There should be an one-to-one mapping between original array dimensions and new tiled dimensions to ensure correctness. We use following equation to obtain new array accesses from original array accesses.

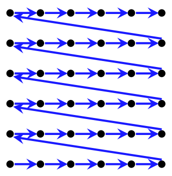


Figure 8: Original memory layout

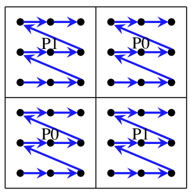


Figure 9: Tiled memory layout

$$\begin{aligned}\vec{T} &= (\psi_{A_k} \cdot \vec{a})/\tau_k \\ \vec{t} &= (\psi_{A_k} \cdot \vec{a})\%\tau_k\end{aligned} \tag{10}$$

where $\vec{T}$ is the access vector corresponding to inter-tile dimensions, $\vec{t}$ to intra-tile dimensions, ψ_A is the data tiling hyperplane, and τ is the tile size. Since the array tiling hyperplanes form a full-ranked matrix, a one-to-one mapping exists between original and new array accesses. $\vec{T}$ represents inter data tile dimensions which enumerates data tiles and $\vec{t}$ scans points inside a data tile. Array access $X[i][j]$ will be transformed into $X[ii_t][jj_t][i_t][j_t]$. If the data tiling hyperplanes used are (1,0) and (0,1) and tile sizes τ_1, τ_2, the new array access will be
$A[i/\tau_1][j/\tau_2][i\%\tau_1][j\%\tau_2]$.
The size of the data tile that is to be allocated is $\tau_1 * \tau_2$. This mapping is exact for data hyperplanes which have only one non-zero component, i.e., all points in the new array layout will have corresponding points in the original layout. If the data tiling hyperplanes used are (1,1) and (0,1) and the tile size is τ, the new array access as per (10) is $A[(i+j)/\tau_1][j/\tau_2][(i+j)\%\tau_1][j\%\tau_2]$. The size of the data tile that needs to be allocated is $(\tau_1 + \tau_2) * \tau_2$. Hence, if the data tiling hyperplane has more than one non-zero component, for example (1,1), we still obtain a correct one-to-one mapping. We would have however allocated more than the required amount of memory. We choose this mapping in spite of it not being exact due to the simplicity of resulting new access expressions. With this mapping of (10), we always have either a mod or a divide of an entire expression – this enables us to simplify access expressions.

To selectively allocate only the required data tiles, we split the transformed array access $A_t[ii_t][jj_t][i_t][j_t]$ into two parts, (i) $ptr = A_t[ii_t][jj_t]$ which returns the pointer to data tile (ii_t, jj_t), and (ii) $ptr[i_t][j_t]$ which indexes the array element within a data tile. A_t is used as an array of pointers to data tiles. Only when a particular data tile (ii_t, jj_t) is required by a node, a new data tile buffer is allocated and stored in $A_t[ii_i][jj_t]$.

Simplification of access expressions: Modified access functions obtained after transformation have the additional cost of a divide, a mod, and an additional memory access (to obtain the data tile pointer) for each array access. This could lead to significant overhead and may prohibit other optimizations such as vectorization. We hoist the divide, mod and array dereference operations out of the innermost loop by splitting it. Consider the data tiled code shown in Figure 10. The transformed code is shown in Figure 11.

```
//forward x sweep
  for (jj=0; jj<floord(N, 128); jj++)
   for (ii=0; ii<floord(N, 128); ii++)
    for (i=max(1,ii*128); i<min(ii*128+127, N); i++)
     for (j=max(1,jj*128); j<min(jj*128+127, N); j++)
      X[i/128][j/128][i%128][j%128] =
         X[i/128][j/128][i%128][j%128] -
         X[i/128][(j-1)/128][i%128][(j-1)%128] *
         A[i/128][j/128][i%128][j%128] /
         B[i/128][(j-1)/128][i%128][(j-1)%128]; //S1
```

Figure 10: Data tiled ADI example

```
//forward x sweep
  for (jj=0; jj<floord(N, 128); jj++)
   for (ii=0; ii<floord(N, 128); ii++)
    for (i=max(1, ii*128); i<min(ii*128+127, N); i++){
      j = max(1, jj*128);
      //peeled iteration
      X[i/128][j/128][i%128][j%128] =
              X[i/128][j/128][i%128][j%128] -
               X[i/128][(j-1)/128][i%128][(j-1)%128] *
               A[i/128][j/128][i%128][j%128] /
               B[i/128][(j-1)/128][i%128][(j-1)%128]; //S1
     j++;
     X_ptr = X[i/128][j/128];
     A_ptr = A[i/128][j/128];
     B_ptr = B[i/128][j/128];
     i_mod = i%128;
     lb = max(1, jj*128+1);
     for (j=max(1,jj*128+1); j<min(jj*128+127, N); j++)
      X_ptr[i_mod][j-lb] =  X_ptr[i_mod][j-lb] -
               X_ptr[i_mod][j-lb-1] * A_ptr[i_mod][j-lb] /
               X_ptr[i_mod][j-lb-1]; //S1
}
```

Figure 11: Optimized data tiled ADI example

5. EXPERIMENTAL EVALUATION

This section presents experiments demonstrating improvement over existing techniques. Our framework is implemented as a part of a publicly available source to source polyhedral tool chain. The input for our framework is sequential C code which can be arbitrarily nested affine loop nests. Compilable code to find computation placements and to distribute data is automatically generated. Cloog-isl [3] is used to generate code from the polyhedral representation. METIS [16] is used to partition the initial graph (Section 2), and to determine computation distributions.

We first determine the compute and data tiling hyperplanes using techniques described in Section 2. Computation hyperplanes are used to transform and tile the sequential code [17]. Techniques described in [4, 8] are used to construct communication sets and generate MPI code for distributed memory. The communication and distributed-memory code works for any arbitrary computation placement. Data spaces are tiled and array access expressions are modified using the data tiling hyperplanes. We generate functions to perform on-demand allocation and buffer management as explained in Section 4. These functions are called at runtime for buffer allocation and management. All of these steps work in an end-to-end automatic manner taking unmodified sequential affine loop nests in C to parallelized code.

Benchmarks: We present results for Floyd-Warshall (`floyd`), LU Decomposition (`lu`), Cholesky Factorization (`cholesky`), Alternating Direction Implicit solver (`adi`), 2mm (`2mm`), and 3mm (`3mm`) benchmarks. All these benchmarks are chosen from the publicly available Polybench/C 3.2 suite [18]. For comparing against ScaLAPACK programs, we use `atax`, BiCG Sub Kernel (`bicg`), `gemver`, `gesummv`, and matrix vector product and transpose (`mvt`) benchmarks, also from the Polybench/C 3.2 suite [18]. All benchmarks use double-precision floating-point operations. The compiler used for all experiments is ICC 13.0.1 with options -O3 -ansi-alias -fp-model precise. *pluto-data-tile-gp* refers to our code. Where applicable, we compare or comment on solutions that would have been found by previous approaches [13, 15, 4], and we also mention the specific mapping found by the graph partitioning approach. Problem sizes used are listed in Table 1 and 2.

Benchmark	Problem size
`floyd`	4096 x 4096
`cholesky`	4096 x 4096
`lu`	8192 x 8192
`2mm`	2048 x 2048
`3mm`	2048 x 2048

Table 1: Problem sizes for shared memory evaluation

Benchmark	Problem size per processor
`gemver`	20000 x 20000
`bicg`	40000 x 40000
`gesummv`	30000 x 30000
`mvt`	30000 x 30000
`atax`	30000 x 30000
`floyd`	2048 x 2048
`lu`	4096 x 4096
`adi`	128 x 4096 x 4096

Table 2: Problem size (per proc) for distributed-memory evaluation

5.1 Distributed memory

The experiments were run on a 32-node InfiniBand cluster of dual-SMP Xeon servers. Each node on the cluster consists of two quad-core Intel Xeon E5430 2.66 GHz processors with 12 MB L2 cache and 16 GB RAM. The InfiniBand host adapter is a Mellanox MT25204 (InfiniHost III Lx HCA). All nodes run 64-bit Linux kernel version 2.6.18. The cluster uses MVAPICH2-1.8.1 as the MPI implementation. We measured a point-to-point latency of 3.36 μs, unidirectional and bidirectional bandwidths of 1.5 GB/s and 2.56 GB/s respectively. We developed ScaLAPACK versions of the benchmarks using multi-thread ScaLAPACK routines of Intel MKL 11.0.1 library. All experiments are run with 8 threads per node.

Figure 12 shows the weak scaling performance for both ScaLAPACK code and our framework. ScaLAPACK internally uses 2-d block cyclic distributions for all routines. Our framework computes the optimal computation placements for each of the benchmarks. For `gemver`, our framework finds the *sudoku* distribution that significantly outperforms 2-d block cyclic distribution. As we are able to fuse the first two loop nests in `gemver` and perform data tiling, our single thread performance is improved by about 3x. For `mvt`, `bicg` and `gesummv` benchmarks, transformations applied result in an outer parallel loop. The output of our framework is a 1-d block distribution with no communication, and this results in near ideal scaling. Figure 13c shows the weak scaling performance for `adi`. Previous schemes [4, 13] would have chosen 1-d block distribution for `adi`, that leads to $O(n^2)$ communication (nxn being the data size), and does not scale. On the other hand, our framework finds the sudoku-like placement that has $O(n)$ communication only. Figure 13a shows the weak scaling performance for `floyd`. The performance of *pluto-data-tile-gp* is very close to manually written 2-d blocked floyd. 2-d block distribution performs better than a 1-d block one due to a higher ratio of computation to communication – in this case, it leads to a 3$\times$ reduction in communication volume. Our framework also implicitly finds the optimal dimensionality of the distribution leading to the minimum communication volume. Note that a higher dimensional mapping may not be necessarily optimal for an entire sequence of loop nests being optimized.

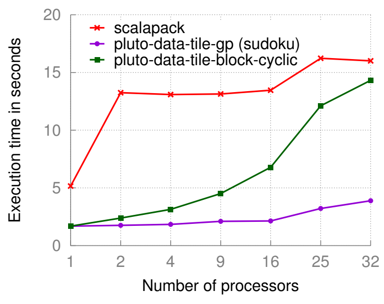
5.2 Shared memory

The experiments were run on a four socket machine with AMD Opteron 6136 CPUs (2.4 GHz, 128 KB L1, 512 KB L2 and 6 MB L3 cache). The shared memory has a NUMA architecture and numactl to bind threads and pages appropriately for all our experiments. When not performing data tiling (for comparison), we did a simple interleaving of pages across all NUMA nodes.

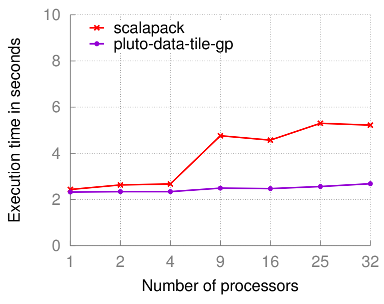
Figure 14 shows that data tiling leads to a significant improvement in single thread performance, and hence benefits shared-memory parallelization as well. Data tiling enhances the spatial locality of space tiled loops. After data tiling, data accessed by a compute tile is contiguous in memory. There will only be cold caches for all accesses to a data tile, i.e., conflict misses are eliminated. It also reduces TLB misses and false sharing. Due to simplification of the modified access functions, we completely eliminated associated overhead from the innermost loop. This results in a geometric mean speedup of 2.67$\times$ over code with no data tiling. For `cholesky` we see a very high speedup of 5.42x. This is also due to data tiling enabling vectorization. `cholesky` kernel had spatially conflicting accesses in a single statement. This kernel is not readily vectorizable by icc as the memory accesses are not contiguous. If j is the innermost loop, then consecutive access of $A[j][i]$ are array size apart. However, after data tiling, accesses due to $A[j][i]$ are tile size apart, and icc can vectorize the code. So, in addition to enhancing locality, data tiling also enables vectorization.

6. RELATED WORK

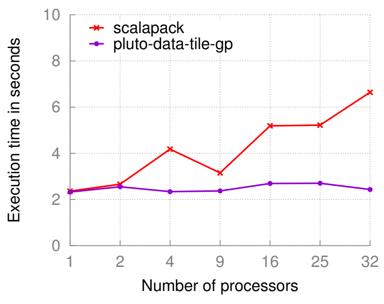
Our approach is built on top of and coupled with communication set generation and distributed-memory code generation works of Bondhugula [4] and Dathathri et al [8]. The communication set construction scheme of [8] is used and the volume of communication for a given transformation does not increase due to data tiling.



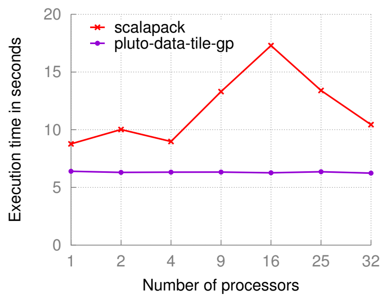
(a) gemver



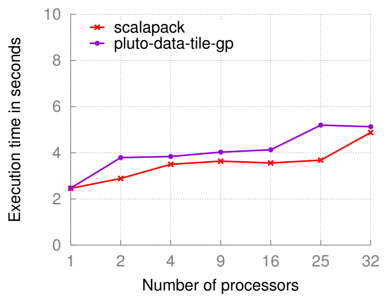
(b) mvt



(c) bicg

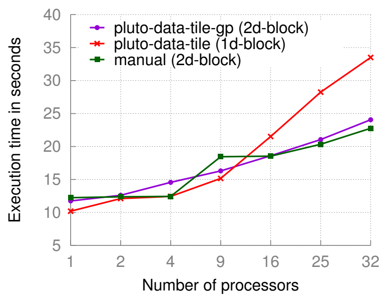


(d) gesummv

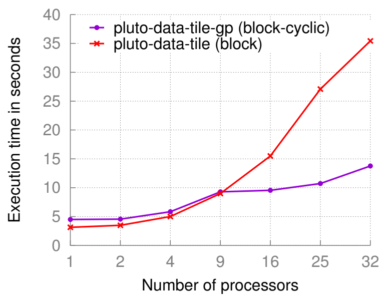


(e) atax

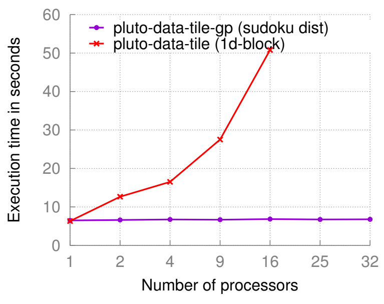
Figure 12: Weak scaling performance of scalapack and pluto-data-tile



(a) floyd



(b) lu



(c) adi

Figure 13: Weak scaling performance of pluto-data-tile-gp on 32 processors (256 cores)

The works of Kennedy and Kremer [13], Chapman et al [6], Garcia et al [11], Gupta and Banerjee [12] have addressed the problem of finding automatic data distributions for distributed memory architectures in the context of regular programs. These approaches first decompose the input program into regions (also called phases) at the granularity of loop nests. An array has a single distribution throughout a phase and data is remapped between phases (dynamic distributions). Within each phase, data distributions and array alignments are found that lead to the least communication volume. The solution space of these works only includes data distributions typically supported by HPF (High Performance Fortran), i.e., block or block-cyclic. On the other hand, our technique first determines computation placements and the data distributions are then derived from it. It thus automatically captures array alignments, static and dynamic distributions, and array replications modeled in the previous approaches. To provide this flexibility, our approach included an elaborate data allocation scheme. To summarize, our approach has the following advantages over all previous works: (i) our solution space includes arbitrary mappings including multipartitioning-style, not just block and block cyclic, (ii) it has the flexibility to apply locality-enhancing transformations such as time tiling since we do not adhere to the "owner computes" rule, and (iii) it minimizes both communication volume and load imbalance.

The work of Anderson and Lam [2] combined with distributed-memory code generation [1] deals with finding computation and data distributions in a unified manner. It only deals with sequences of perfectly nested loops, and finds affine computation and data mappings to virtual processors. Heuristics are then used to map virtual processors to actual physical processors, which again only support block or block-cyclic mappings.

Multipartitioning [15] and generalized multipartitioning [7] were specialized computation mapping schemes implemented in dHPF that provided excellent scaling for SP and BT from NAS parallel benchmarks. They are also suitable for the smaller *gemver* and *adi* codes we used for evaluation. However, a general mapping strategy that automatically deduced multipartitioning as a suitable mapping while also incorporating block, block-cyclic, and other arbitrary mappings for affine loop nests did not exist prior to this work.

Many recent works [14] and [22] addressed the problem of opti-

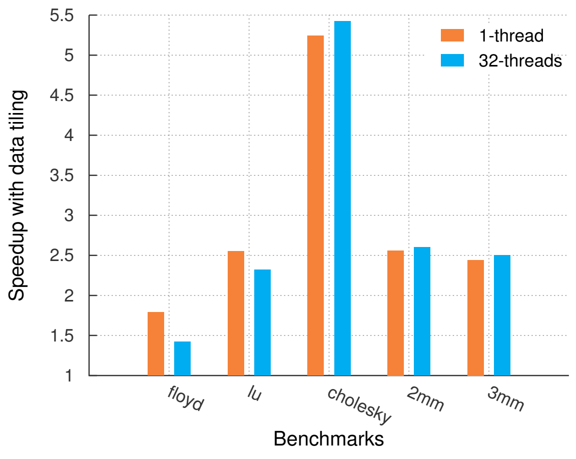


Figure 14: Speedup with data tiling over no data tiling on shared memory multicore. Sequential execution times for floyd, lu, cholesky, 2mm, 3mm are 225s, 494s, 297s, 93s and 109s respectively.

mizing data layouts for shared-memory architectures. Lu et al [14] proposed a data layout framework to enhance locality on NUCA-based chip multiprocessors. They find a single "localizable" data and computation partitioning, and the data is tiled along only one dimension. On the other hand, we find a full-ranked computation and data mapping in a unified manner, and the data is thus tiled along multiple dimensions. This approach is required due to our problem being very different from that of [14]. Zhang et al [22] proposed techniques to determine data tiling hyperplanes and computation-to-core mappings. They too formulate a graph partitioning problem to find the computation-to-core-mapping to minimize communication volume. However, they do not consider load balance and use simple heuristics to partition the graph. Both the approaches do not address on-demand allocation and buffer reuse and their techniques do not provide the flexibility to support any arbitrary computation mapping. Yuki et al [21] describe a memory allocation for distributed-memory. Though it enables weak scaling, it only works for uniform dependences and solves the problem in a very different way to avoid the use of modulos with respect to runtime parameters – a problem that we never face.

Jeremy et al [20] propose techniques to optimize sequences of BLAS kernels calls for shared-memory architectures. They develop a domain-specific language to express linear algebra operations and their BTO (Build to Order) compiler performs optimizations such as loop fusion, tiling etc. across sequences of BLAS calls. Our approach fits well in such domain-specific compilers as well to enable targeting distributed memory.

7. CONCLUSIONS

We proposed techniques for data allocation and computation mapping when compiling affine loop nest sequences for distributed-memory clusters. These techniques allowed us to complete missing steps in allowing effective end-to-end distributed-memory parallelization of affine loop nests. Our approach for data allocation relies on a tiled view of data spaces. The scheme allocates and deallocates tiles on-demand and exploits their reuse. We showed how our approach for computation mapping is able to come up with more effective mappings than those that can be used with vendor-supplied BLAS libraries. These mappings that were automatically determined also subsume mappings with similar properties that were implemented and used manually in previous works. Experimental results on some sequences of BLAS calls demonstrated a mean speedup of 1.82× over versions written with ScaLAPACK and a maximum speedup on 4× while running on a 32-node cluster. Besides enabling weak scaling for distributed memory, data tiling also improves locality for shared-memory parallelization. Experimental results on a 32-core shared-memory NUMA SMP system show a mean speedup of 2.67× over code that is not data tiled.

8. REFERENCES

[1] S. P. Amarasinghe and M. S. Lam. Communication optimization and code generation for distributed memory machines. In *PLDI*, pages 126–138, 1993.

[2] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *ACM SIGPLAN PLDI*, pages 112–125, 1993.

[3] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT*, pages 7–16, 2004.

[4] U. Bondhugula. Compiling affine loop nests for distributed-memory parallel architectures. In *Supercomputing (SC)*, page 33. ACM, 2013.

[5] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *ETAPS CC*, 2008.

[6] B. M. Chapman, T. Fahringer, and H. P. Zima. Automatic support for data distribution on distributed memory multiprocessor systems. In *LCPC*, pages 184–199, 1993.

[7] A. Darte, J. Mellor-Crummey, R. Fowler, and D. Chavarría-Miranda. Generalized multipartitioning of multi-dimensional arrays for parallelizing line-sweep computations. *JPDC*, 63:887–911, Sep 2003.

[8] R. Dathathri, C. Reddy, T. Ramashekar, and U. Bondhugula. Generating efficient data movement code for heterogeneous architectures with distributed-memory. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[9] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[10] P. Feautrier. Some efficient solutions to the affine scheduling problem: Part I, one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, 1992.

[11] J. Garcia, E. Ayguade, and J. Labarta. A novel approach towards automatic data distribution. In *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference*, pages 78–78. IEEE, 1995.

[12] M. Gupta and P. Banerjee. Paradigm: A compiler for automatic data distribution on multicomputers. In *Proceedings of the International Conference on Supercomputing*, ICS '93, pages 87–96. ACM, 1993.

[13] K. Kennedy and U. Kremer. Automatic data layout for distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 20(4):869–916, 1998.

[14] Q. Lu, C. Alias, U. Bondhugula, et al. Data layout transformation for enhancing data locality on nuca chip multiprocessors. In *PACT'09*, pages 348–357, 2009.

[15] J. Mellor-Crummey, V. Adve, B. Broom, D. Chavarria-Miranda, R. Fowler, G. Jin, K. Kennedy, and Q. Yi. Advanced optimization strategies in the rice dHPF compiler. *Concurrency: Practice and Experience*, pages 741–767, 2002.

[16] METIS -Family of Graph and Hypergraph Partitioning

Softwares.
http://glaros.dtc.umn.edu/gkhome/views/metis.

[17] PLUTO: A polyhedral automatic parallelizer and locality optimizer for multicores.
http://pluto-compiler.sourceforge.net.

[18] Polybench. http://polybench.sourceforge.net.

[19] Scotch - Sequential and parallel graph partitioning software.
http://www.labri.fr/perso/pelegrin/scotch.

[20] J. G. Siek, I. Karlin, and E. R. Jessup. Build to order linear algebra kernels. In *In International Symposium on Parallel and Distributed Processing 2008 (IPDPS 2008*, pages 1–8, 2008.

[21] T. Yuki and S. Rajopadhye. Canonic multi-projection: Memory allocation for distributed memory parallelization. Technical report, Technical Report CS-11-106, Colorado State University, 2011.

[22] Y. Zhang, W. Ding, M. Kandemir, J. Liu, and O. Jang. A data layout optimization framework for nuca-based multicores. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 489–500. ACM, 2011.