Unified framework for sparse *and* dense SPMD code generation (preliminary report)

Vladimir Kotlyar Keshav Pingali Paul Stodghill

March 10, 1997

Abstract

We describe a novel approach to sparse *and* dense SPMD code generation: we view arrays (sparse and dense) as distributed relations and parallel loop execution as distributed relational query evaluation. This approach provides for a uniform treatment of arbitrary sparse matrix formats and partitioning information formats. The relational algebra view of computation and communication sets provides new opportunities for the optimization of node program performance and the reduction of communucation set generation and index translation overhead.

1 Summary of the paper

1.1 Problem statement

Sparse matrix computations are at the core of many computational science algorithms. A typical application can often be separated into the *discretization* module, which translates a continuous problem (such as a system of differential equations) into a sequence of sparse matrix problems, and into the *solver* module which solves the matrix problems.

Typically the solver is the most time and space-intensive part of an application and, quite naturally, much effort both in numerical analysis and in compilers community has been devoted to producing efficient parallel and sequential code for sparse matrix solvers. There are two challenges in generating solver code that has to be interfaced with discretization systems:

- Different discretization systems produce the sparse matrices in many different formats. Therefore compiler should be able to generate solver code for different storage formats. We have addressed this problem in [14].
- Some discretization systems partition the problem for parallel solution, and use various methods for specifying the partitioning (distribution). Therefore a compiler should be able to produce parallel code for arbitrary distribution formats.

Our problem is as follows: *given a dense program and descriptions of sparse matrix formats and data and computation distribution formats – generate parallel sparse SPMD code.* In this paper, we consider DOALL loops and loops with reductions. This allows us to address the needs of many important applications.

1.2 Previous work

Libraries A number of sparse matrix libraries have been developed over the years: SPARSPAK [12], PETSc [5], BlockSolve [15] – just to name a few. Their disadvantage is that they use special storage formats and partitioning strategies. Therefore they impose the overheads of data conversions both in terms of user effort and run-time overhead.

Compilers: regular computations Most of the research in parallelizing compiler community has focused on dense matrix (regular) computations. This is because the flow of data in these computations can be analyzed at compile-time.

Compilers: irregular computations The closest alternative to our work is a combination of Bik's sparse compiler [8] and the work on specifying and compiling irregular codes in HPF Fortran ([20], [19], [17]). One could use the sparse compiler to translate dense sequential loops into sparse loops. Then, the Fortran D or Vienna Fortran compiler can be used to compile these sparse loops. First of all, this scheme would restrict a user to a fixed set of sparse matrix formats, as well as to a particular way of specifying distributions. Second, this approach prevents the compiler from recognizing and exploiting "hybrid" dense-sparse sets as described in Section 4. One reason is that in this scheme the compiler has to deal with indirect array subscripts and other complications, which obscure the actual structure in the code. Another reason is that this scheme relies on the use of run-time library (Chaos or PARTI) to perform index translation and communication schedule generation. This library, even with extensions for multi-block problems [1], supports either fine grain irregular communication (an integer, or a real number is a unit of index translation and communication), or regular communication.

Interface problem The Meta-Chaos library [11] provides means of *transferring* data between different applications. We address the problem of *using* different data formats *directly*. This is advantageous because the different matrix formats usually reflect the needs of an application. For example, the BlockSolve format is designed for matrices that have groups of rows with identical non-zero structure; similarly, the jagged diagonal format is used for obtaining good performance on vector machines [6].

1.3 Our approach

In [14], we have solved the problem of dealing with various sparse data structures by viewing sparse matrices as relations (as in relational databases), and reducing the problem of generating sparse code to a relational query optimization problem. In this paper, we apply this relational abstraction to sparse SPMD code generation. The advantages of this approach are the following.

- We hide the details of arbitrary sparse matrix formats by representing sparse matrices as relations.
- We hide the details of various distribution and index translation functions by representing them as relations, as well. Thus, we solve the "interface problem".
- We are able to recognize when various sets are either completely regular (i.e. can be represented in closed form) or are certain combinations (such as a cross product) of dense and sparse sets and thus can optimize their storage and communication costs between processors.

Our code generator produces code that is a series of set (relational) assignments and communication statements, which can be optimized by variations of standard code optimization techniques.

- Since we can handle dense code generation *and* arbitrary storage formats, our techniques might be of value in the implementation of HPF Fortran-like languages: an HPF front-end can produce storage declarations, which our code generator can use to generate parallel code. In this way, the treatment of dense and sparse codes can be unified.

Here is the outline of the rest of the abstract. In Section 2 we describe our code generation algorithm. In Section 3 we show some experimental results. In Section 4 we present optimizations on the generated code, which are possible within our framework. In Section 5 we discuss the use of our compiler within an implementation of an HPF-like compiler. And in Section 6 we present some conclusions, outstanding issues and on-going work.

2 SPMD code generation as distributed query evaluation

In this section we use an example to describe our code generation strategy. First, we describe sparse matrix-vector multiplication and outline how this application is "hand-parallelized" in practice. Then we briefly mention the connection between compilation of sparse matrix codes and relational query optimization and describe how the distributed query is formulated. In particular, we describe an extension to HPF alignment/distribution specification that allows us to use user-specified distributions and data structures for local storage. Then we outline our code generation algorithm.

2.1 An example

Consider the loop nest shown in Figure 1, that computes a matrix-vector product $\mathbf{y} = \mathbf{A} \cdot \mathbf{x}$ [1] . The

```
DO $i = 1, n$
    DO $j = 1, n$
        IF $\langle i, j \rangle \in A$ THEN
            $Y(i) = Y(i) + A(i, j) * X(j)$
```

Figure 1: Sparse matrix-vector product

matrix A is sparse (we do not care about the storage format at this point) and the vectors X and Y are dense. Also A has a non-zero in every row – we say that it is dense in the row dimension. Only those iterations $\langle i, j \rangle$ for which $A(i, j) \neq 0$ have to be executed. Since only non-zeros of A are stored, this condition translates into the condition in the loop. This sparse matrix-vector multiplication kernel (SMVM) is an important part of many matrix solvers and, despite its simplicity, illustrates most of the ideas of our code generation approach.

We consider the case when the matrix is partitioned *irregularly* by rows, which means that the processor assignments $p = \mu(i)$ for the rows i have been computed at run-time and can not be represented in closed-form. Usually, μ is a result of a graph partitioning or domain partitioning procedure [13]. We call μ a *partition function.* On processor p the matrix A is stored as a matrix

[1] We use bold letters, as in $\mathbf{x}$, to denote matrices, vectors and tuples of attributes

$A^{(p)}$ in new coordinates $\langle i', j\rangle$: i' is the *local index* of the rows of A and j is the local column index, which is the same as the global column index. i' runs between 1 and n', which is the number of rows assigned to p. Since A is not replicated, there is a 1-1 mapping $\delta(i, p, i')$ between the global row indices i and the pairs $\langle p, i'\rangle$. We call δ a *placement function.* There is an obvious relationship between μ and δ:

$$(p = \mu(i)) \Leftrightarrow (\exists i' : \delta(i, p, i')) \tag{1}$$

The elements of X and Y are placed identically to rows of A. That is, on processor p we store the vectors $X^{(p)}$ and $Y^{(p)}$ with the local index i' related to the global index through δ.

Just as A is a relation that is partitioned across the processors, so is δ. Typical examples are:

- Each processor keeps an array of i's indexed by i''s. This is how, for example, Parallel ELLPACK ([10]) feeds problems to the matrix solver.
- The $\langle i, p, i'\rangle$ tuples are placed on processor $q = i/B$ for some block-size B. This is the case with the paged translation tables in the Chaos library.
- The simplest case: δ is replicated. This is actually the case in the Petsc library [5]. The rows are assigned to processors in contiguous blocks and δ can be represented compactly by an array of block sizes that is replicated across the processors.

It is not hard to see that if each processor p executes the iterations i such that $p = \mu(i)$, then no communication is necessary for A and Y. The set of elements of X to be communicated, or the *view set* of X, is:

$$\{j \mid \exists i : (1 \le i, j \le N) \wedge (\langle i, j\rangle \in A)\} \tag{2}$$

The local and non-local elements of X are collected into an array $\bar{X}$. An index translation function $\bar{j} = f(j)$ has to be computed to represent the relationship between the local index $\bar{j}$ in $\bar{X}$ and the global index j in X. One common optimization is to renumber the j indices stored in A into $f(j)$ (if possible). Given all of the above a typical parallel implementation is outlined in Figure 2. The

```
Collect the elements of the view set of $X$
Exchange requests for non-local data
Allocate storage for $\bar{X}$
Renumber $A^{(p)}$
Exchange the values for $X$
DO $i' = 1, n'$
    DO $\bar{j} = 1, \bar{n}$
        IF $\langle i', \bar{j}\rangle \in A^{(p)}$ THEN
            $Y^{(p)}(i') = Y^{(p)}(i') + A^{(p)}(i', \bar{j}) * \bar{X}(\bar{j})$
```

Figure 2: Outline of a parallel implementation of SMVM

highlights of this example are:

- We had to recognize that A, Y and the iterations of the loop are *globally collocated.* This point is discussed in more detail in Section 2.5.

- The local computation is just a matrix-vector product on the local data. In general, this means that the relationship between the global indices i of A and Y has been translated into the relationship between the local indices i' of the local storage. We call this effect *local collocation*. It is discussed in Section 2.6.

2.2 Relational query optimization approach

In [14] we show how the problem of generating efficient sequential code for a loop nest like the one in Figure 1 can be reduced to a relational query optimization problem. If we view arrays and loop iteration space as relations, then the final loop nest enumerates tuples satisfying the following query:

$$Q = \sigma_{\mathrm{bnd}}\sigma_{\mathrm{acc}}\Big(I(i,j) \times A(i_a, j_a, v_a) \times X(j_x, v_x) \times Y(i_y, v_y)\Big) \tag{3}$$

where I is the relation that represents the bounds of the original loop. Relations A, X and Y correspond to the arrays in the program. The notation "$A(i_a, j_a, v_a)$" names the fields (attributes) of the relation A. In our case i_a is the row index, j_a is the column index and v_a is the value of the matrix element. To simplify the presentation we ignore the value fields.

σ_{bnd} is the selection operator that "filters" the tuples satisfying the loop bounds. And σ_{acc} relates the dimensions of the arrays to loop indices according to array access functions in the original loop nest:

$$\begin{aligned} \text{bnd} &\equiv 1 \le i,j \le n & (4)\\ \text{accesses} &\equiv (i = i_a = i_y) \wedge (j = j_a = j_x) & (5) \end{aligned}$$

The key to efficient evaluation of this query lies in "pushing" the σ_{acc} selection through the cross products to obtain natural joins:

$$Q = \sigma_{bnd}(I(i,j) \bowtie A(i,j) \bowtie X(j) \bowtie Y(i)) \tag{6}$$

In general, when compiling dense loops with sparsity predicates we get relational queries of the form:

$$\sigma_{\mathrm{bnd}}\sigma_{\mathrm{acc}}(I \times \Phi(A_1, A_2, \ldots, A_n)) \tag{7}$$

where I is the relation representing the iteration set of the original loop and A_k's represent arrays in the loop. The selection σ_{bnd} represents loop bounds. σ_{acc} enforces equalities between the attributes of the relations induced by array access functions in the loop nest. Both selections are naturally limited to affine equalities and inequalities. Φ is a combination of set union and cartesian product operators. In the SMVM example it is just a cross product. In this abstract we limit our discussion to conjunctive queries: Φ is a cross product of its inputs. We will discuss the general case in the full paper.

2.3 Distributed query formulation

The question now is: how do we specify the distributed query that corresponds to parallel execution of the loop? First of all, we need to specify how the relations are partitioned across the processors. Ullman [18] describes a *horizontal* scheme for partitioning a relation. Each *global relation* is represented as a union (not necessarily disjoint) of processor contributions, or *fragments*:

$$A = \bigcup_p A^{(p)} \tag{8}$$

Associated with a fragment $A^{(p)}$ we define a *guard condition* g_p such that:

$$\sigma_{g_p} A = A^{(p)} \tag{9}$$

This means that if $g_p(\mathbf{x}) = true$ for some tuple $\mathbf{x}$, then $\mathbf{x} \in A$ implies $\mathbf{x} \in A^{(p)}$. Notice that this definition does not imply that $\sigma_{g_p} A^{(q)} = \emptyset$ for $p \neq q$. In other words, this scheme allows replication. In our SMVM example: $g_p(i,j) \equiv (p = \mu(i))$.

Unfortunately, this simple scheme is insufficient. Each fragment in (8) holds the original, global, attributes, but we would like to be explicit about local attributes (indices) and the placement function. Notice that in the SMVM example, the local matrices $A^{(p)}$ would have been sparse both in rows and in columns if they stored the global row index i, because each processor only stores a subset of the rows. By using a local row index i' we made the local matrices dense in the row index, just like the global matrix A. In general, storing local relations in attributes other then global allows us to translate structure present in the global space into the local space. This will become more evident in Section 2.6.

If we represent a placement function $\delta(\mathbf{x}, p, \mathbf{x}')$, that maps global attributes $\mathbf{x}$ to local attributes and processor numbers, as a relation with $\langle \mathbf{x}, p, \mathbf{x}' \rangle$ tuples, then we can rewrite (8) as: [2]

$$A = \bigcup_p \pi_{\mathbf{x}} \Big(\delta(\mathbf{x}, p, \mathbf{x}') \bowtie A^{(p)}(\mathbf{x}') \Big) \tag{10}$$

This query says that the global relation A is the union of local fragments $A^{(p)}$ translated into global attributes. Notice that δ can be a single (distributed) relation or a relational algebra expression. In the following discussion we use the notation:

$$\phi_p(A) = \pi_{\mathbf{x}} \Big(\delta(\mathbf{x}, p, \mathbf{x}') \bowtie A^{(p)}(\mathbf{x}') \Big) \tag{11}$$

That is $\phi_p(A)$ is the contribution to the global relation from processor p.

Now to complete the specification we need to give more structure to δ, so that the compiler is able to recognize collocation in global and local spaces.

2.4 Placement function

Languages like HPF Fortran recognize collocation in the global space by dividing the partitioning function μ into two steps: alignment and distribution. Alignment is an affine map to an abstract set of virtual processors, a.k.a. template. Distribution then partitions the template between the physical processors. The essential property of this scheme is that if two data elements are mapped to the same template element, they would also end up on the same processor: collocation in the virtual processor space is sufficient for collocation in the physical processor space. For a detailed discussion see [4], [7], [9].

We can not use this scheme because it only specifies a part of the placement function: the assignment of global indices to processors. HPF compilers derive the rest based on the parameters of the partition (see [3], for example). The disadvantage of this scheme in our context is that compiler has a set of distribution formats "hard-wired".

We augment the HPF partitioning scheme to arrive at a full placement function in the following way (see Figure 3):

[2] π is the projection operator

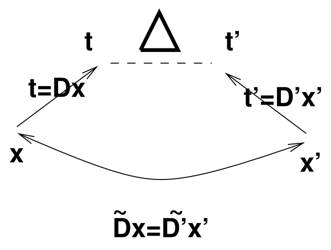

Figure 3: Relationship between global and local attributes and template spaces

- A global attribute $\mathbf{x}$ is mapped to the *global template space* $\mathbf{t}$ via an affine map $\mathbf{D}$: $\mathbf{t} = \mathbf{D}\mathbf{x}'$ [3] .The map $\mathbf{D}$ is called *global alignment*.
- Global template space is related to *local template space* $\mathbf{t}'$ and processor numbers via a 1-1 *distribution* map $\Delta(\mathbf{t}, p, \mathbf{t}')$. The flexibility in the specification of different partitions lie in the flexibility of having an arbitrary relation for Δ.
- Local template space is related to local attributes $\mathbf{x}'$ via $\mathbf{t}' = \mathbf{D}'\mathbf{x}'$. The affine map $\mathbf{D}'$ is called *local alignment*.
- The picture is not complete without specifying how the dimensions of A that are not partitioned are mapped to the dimensions of the fragments $A^{(p)}$. This can be achieved by using an affine relation: $\tilde{\mathbf{D}}\mathbf{x} = \tilde{\mathbf{D}}'\mathbf{x}'$.

The tuple $\langle \mathbf{D}, \mathbf{D}', \tilde{\mathbf{D}}, \tilde{\mathbf{D}}', \Delta \rangle$ is called the *placement* of relation A. The placement function δ can be written as:

$$\delta(\mathbf{x}, p, \mathbf{x}') \equiv \Big(\exists \mathbf{t}, \mathbf{t}' : \Big(\mathbf{t} = \mathbf{D}\mathbf{x}' \wedge \Delta(\mathbf{t}, p, \mathbf{t}') \wedge \mathbf{t}' = \mathbf{D}'\mathbf{x}' \wedge \tilde{\mathbf{D}}\mathbf{x} = \mathbf{D}'\mathbf{x}'\Big)\Big) \tag{12}$$

Or using relational algebra:

$$\delta = \pi_{\mathbf{x},p,\mathbf{x}'} \sigma_{\mathbf{t}=\mathbf{D}\mathbf{x}' \wedge \ldots} \Big(\mathrm{Domain}(\mathbf{x}) \times \mathrm{Domain}(p, \mathbf{x}') \times \Delta(\mathbf{t}, p, \mathbf{t}')\Big) \tag{13}$$

The values for $\mathbf{D}$, $\mathbf{D}'$, $\tilde{\mathbf{D}}$ and $\tilde{\mathbf{D}}'$ for the matrix A in our SMVM example are shown in Table 1.

	Matrix value	Comment
global alignment	$\mathbf{D} = \begin{pmatrix}1 & 0\end{pmatrix}$	$t = i$
local alignment	$\mathbf{D}' = \begin{pmatrix}1 & 0\end{pmatrix}$	$t' = i'$
collapsed dimensions	$\tilde{\mathbf{D}} = \begin{pmatrix}0 & 1\end{pmatrix}$ $\tilde{\mathbf{D}}' = \begin{pmatrix}0 & 1\end{pmatrix}$	$j' = j$

Table 1: Fragmentation in SMVM example. $A(i, j)$ is divided into $A^{(p)}(i', j')$

By mistake, the user may specify inconsistent placement functions δ when using the above scheme. These inconsistencies, in general, can only be detected at runtime. For example, it can

[3]We actually allow replication by using an affine relation $\mathbf{R}\mathbf{t} = \mathbf{D}\mathbf{x}$, but in this abstract we limit our discussion to non-replicated alignments

only be verified at run-time if a user specified distribution relation Δ in fact provides a 1-1 and onto map. This problem is not unique to our framework – HPF with value-based distributions [19] has a similar problem. Basically, if a function is specified by its values at run-time, its properties can only be checked at run-time. In the full version of the paper, we will describe how to generate a "debugging" version of SPMD code that checks for consistency in the placement function. Moreover we will discuss the conditions on the affine maps that can be checked at compile-time.

2.5 Global collocation

Let us consider the following query, which involves the relations R and S with attributes $\mathbf{x}$ and $\mathbf{y}$, respectively:

$$Q = \sigma_{f(\mathbf{x},\mathbf{y})}(R \times S) = \sigma_{f(\mathbf{x},\mathbf{y})} \bigcup_p \bigcup_q (\phi_p(R) \times \phi_q(S)) \tag{14}$$

where $f(\mathbf{x},\mathbf{y})$ is a selection predicate, which consists of linear equalities and inequalities. We will say that R and S are *globally collocated relative to the query* Q, if for all "interesting" x and y (that satisfy f), we only need to test membership in the fragments of R and S on the same processor. In this case we can rewrite the query in (14) as:

$$Q = \sigma_{f(\mathbf{x},\mathbf{y})} \bigcup_p (\phi_p(R) \times \phi_p(S)) \tag{15}$$

Let $\langle \mathbf{D}_R, \mathbf{D'}_R, \tilde{\mathbf{D}}_R, \tilde{\mathbf{D}}'_R, \Delta_R \rangle$ be the placement of R. Let $\langle \mathbf{D}_S, \mathbf{D'}_S, \tilde{\mathbf{D}}_S, \tilde{\mathbf{D}}'_S, \Delta_S \rangle$ be the placement of S. Furthermore, suppose we can solve $f(\mathbf{x},\mathbf{y})$ to represent $\mathbf{x}$ and $\mathbf{y}$ through a parameter $\mathbf{u}$:

$$\begin{aligned} \mathbf{x} &= \mathbf{x}_0 + \mathbf{F}\mathbf{u} \\ \mathbf{y} &= \mathbf{y}_0 + \mathbf{H}\mathbf{u} \end{aligned}$$

The global template indices can be expressed as:

$$\begin{aligned} \mathbf{t}_R &= \mathbf{D}_R\mathbf{x} &= \mathbf{D}_R\mathbf{x}_0 + \mathbf{D}_R\mathbf{F}\mathbf{u} \\ \mathbf{t}_S &= \mathbf{D}_S\mathbf{y} &= \mathbf{D}_s\mathbf{y}_0 + \mathbf{D}_S\mathbf{H}\mathbf{u} \end{aligned} \tag{16}$$

This gives a sufficient condition for global collocation that can be checked at compile-time:

$$(\Delta_R = \Delta_S) \wedge \left(\forall \mathbf{u} : \mathbf{D}_R\mathbf{x}_0 + \mathbf{D}_R\mathbf{F}\mathbf{u} = \mathbf{D}_s\mathbf{y}_0 + \mathbf{D}_S\mathbf{H}\mathbf{u} \right) \tag{17}$$

This is exactly the alignment condition used in HPF (see [4] and [7] for more details).

In our SMVM example the equalities (coming from array access functions and global alignments) are:

$$\left. \begin{array}{ll} i_A = i & i_Y = i \\ t_A = i_A & t_Y = i_Y \end{array} \right\} \Rightarrow t_A = t_Y$$

which proves that A and Y are globally collocated.

2.6 Local collocation

So far we have shown how our fragmentation scheme allows us to recognize global collocation: we do not have to communicate R and S to compute Q. We now turn to the issue of *local collocation*, which is absent from HPF terminology and has not been addressed in the literature on parallelization of sparse codes. In [14] the main step in generating efficient sparse code is recognition of equalities between the attributes. The equalities allows us to convert cross products into natural joins. By using affine maps for local alignment our placement scheme enables translation of equalities in global attributes into equalities in local attributes.

If the equalities in (16) and (17) are satisfied, we can relate the local attributes of the relations, as well. Notice that our placement scheme gives us the equalities:

$$\mathbf{t'}_R = \mathbf{D'}_R\mathbf{x'} \tag{18}$$

$$\mathbf{t'}_S = \mathbf{D'}_S\mathbf{y'} \tag{19}$$

If R and S are globally collocated, then $\mathbf{t}_R = \mathbf{t}_S$, and we conclude that $\mathbf{t'}_R = \mathbf{t'}_S$ (because Δ is 1-1 and onto) and:

$$\mathbf{D'}_R\mathbf{x'} = \mathbf{D'}_S\mathbf{y'} \tag{20}$$

In the SMVM example we have:

$$(t_A = t_Y) \Rightarrow (t'_A = t'_Y) \Rightarrow (i'_A = i'_Y)$$

which caused an equi-join $A^{(p)}(i', j) \bowtie Y^{(p)}(i')$ to be performed locally.

To summarize the discussion so far:

- We have extended the HPF alignment/distribution scheme to allow arbitrary sparse matrix storage formats.
- Our scheme also allows arbitrary formats for specifying distributions.
- Placement functions are structured in a way that permits translation of equalities in global attributes into equalities in local attributes, thus facilitating efficient local code generation.

2.7 Outline of the query evaluation strategy

In general, we are dealing with distributed queries of the form [4] :

$$Q = \sigma_{bnd}\sigma_{acc}\left(\left(\bigcup_{p_0}\phi_{p_0}(I)\right) \times \left(\bigcup_{p_1}\phi_{p_1}(A_1)\right) \times \left(\bigcup_{p_2}\phi_{p_2}(A_2)\right) \times \ldots\right) \tag{21}$$

Now we have to describe how such queries are evaluated.

In the distributed query literature the optimization problem is: *find the sites that will evaluate parts of the query (21)*. In the context of, say, a banking database spread across branches of the bank, the partitioning of the relations is fixed, and may not be optimal for each query submitted to the system. This is why the choice of sites might be non-trivial in such applications. See [18] for a detailed discussion of the general distributed query optimization problem.

[4]We restrict the discussion in this abstract to queries with cross products

In our case, we expect that the placement of the relations is correlated with the query itself and is given to us by the user. In particular, the placement of the iteration space relation I tells us where the query should be processed. That is the query to be evaluated on each processor is:

$$Q^{(p)} = \sigma_{bnd}\sigma_{acc}\left(\phi_p(I) \times \left(\bigcup_{p_1} \phi_{p_1}(A_1)\right) \times \left(\bigcup_{p_2} \phi_{p_2}(A_2)\right) \times \ldots\right) \tag{22}$$

An obvious optimization is to group all collocated relations together. This reduces the work of figuring out communication sets for each processor and of global-to-local attribute translation. Without loss of generality, assume that I denotes the group collocated with the iteration space (we call it the *local group*) and A_k denotes a group of collocated relations.

In our SMVM example I, A and Y form the local group. X is the only relation to be communicated (in a group by itself):

$$Q^{(p)} =$$

$$\left(\left(\pi_{i,j}(\delta(i,p,i') \bowtie I^{(p)}(i,j) \bowtie A^{(p)}(i,j) \bowtie Y^{(p)})\right) \bowtie \left(\bigcup_q \pi_j(\delta(j,p,j') \bowtie X^{(p)}(j))\right)\right) \tag{23}$$

Now our code generation algorithm is as follows:

1. Order the non-local groups heuristically.

2. Translate the sub-expression for the local group so that it can be evaluated locally. Notice that the sub-expression

   $$\phi_p(I) = \pi_{\mathbf{i}}(\delta_0(\mathbf{i},p,\mathbf{i}') \bowtie I^{(p)}(\mathbf{i}')) \tag{24}$$

   can not be evaluated locally because δ_0 might be a distributed relation (or distributed query expression). This *localization* procedure is outlined in Section 2.8. Let Λ denote the localized expression.

3. Refine the localized expression Λ by bringing the necessary data for each non-local group to the processing cite. This is done by projecting $\sigma_{bnd}\sigma_{acc}(\Lambda \times \text{Domain}(A_k))$ on the attributes of A_k. Now if we let A_k^{loc} be the expression for the localized data of the group A_k, then we update the expression Λ to

   $$\Lambda_{next} = \sigma_{bnd}\sigma_{acc}(\Lambda \times A_k^{\text{loc}})$$

   and continue with the next group. Section 2.9 goes into more detail.

   It should be noted, that the above procedure only applies to conjunctive queries. For queries involving set unions, the procedure for updating Λ is a bit more involved. We leave the details for the full paper.

2.8 Localization of localization of ...

In (23) the sub-expression for the local group is not entirely local to processor p: δ is a distributed relation. We need to collect on the processor p the tuples $\langle i,p,i'\rangle \in \delta$. This turns out to be just another SPMD code generation problem! We get the desired result when we assign δ to another relation $\gamma(i,p,i')$:

```
DO $i = 1, n$
    DO $p = 1, \mathrm{nproc}$
        DO $i' = 1, n$
            IF $\langle i, p, i' \rangle \in \delta$ THEN
                insert $\langle i, p, i' \rangle$ into $\gamma$
```

We partition the iteration space relation $J = J(i, p, i')$ of this loop in the same way as δ is partitioned. We partition $\gamma(i, p, i')$ by the p attribute. The communication required to scatter the values into the non-collocated relation γ would do the trick. Of course, when we generate the code for the local group in the new query, we will have to localize the placement function δ' for δ. It is not hard to see that this recursion terminates: one of the placement functions has to be in closed form – otherwise we would have an infinite chain of placement definitions.

2.9 Communication of communication of ...

After we fully localize the local group we get the query:

$$Q^{(p)} = \pi_{i,j}\Big(\Lambda(i, i', j) \bowtie \Big(\bigcup_q \delta(j, q, j') \bowtie X^{(q)}(j')\Big)\Big) \tag{25}$$

How do we ship X around? By projecting Λ onto j we get the *view set* for X – the set of global indices of X used locally:

$$\mathcal{V}_X^{(p)}(j) = \pi_j(\Lambda(i, i', j) \bowtie \{1 \leq j \leq n\}) \tag{26}$$

To generate communication we need to translate global j into local processor numbers q and local attributes j'. We call the resulting relation a *communication set* of X:

$$\mathcal{C}_X^{(p)}(j, q, j') = \delta(j, q, j') \bowtie \mathcal{V}_X^{(p)}(j) \tag{27}$$

Here again, we are dealing with another distributed query: δ is distributed. The problem now is that for each j that we want to translate we might not know which processor hold the translation tuple $\langle j, q, j' \rangle$. In the Chaos library and in Fortran D, this problem is solved by creating a *paged translation table*: the tuple $\langle j, q, j' \rangle$ resides on the processor $r = j/B$ for some block size B. In our framework, this means that we need to assign δ to another relation γ which is distributed block-wise based on attribute j. The new query is the same as in the previous section, except that the γ is partitioned differently. Once the paged table is computed, we can formulate a distributed query for (27):

$$\mathcal{C}_X(p, j, q, j') = \gamma(j, q, j') \bowtie \mathcal{V}_X(p, j) \tag{28}$$

where $\mathcal{C}_X$ is the global relation built out of $\mathcal{C}_X^{(p)}$ fragments. $\mathcal{V}_X$ is defined analogously. In this query the non-collocated relation is γ. Again, the argument that this recursion terminates is the same as in the previous Section. In the full paper we will describe in more detail how the actual communication is represented in our framework. It should also be noted that our compiler performs invariant code motion of the computation of the communication and storage sets. This is analogous to what is done in the inspector/executor approach [20]).

3 Experimental results

At this point we have implemented a prototype to test out some of our ideas. Table 2 shows the performance of the sparse conjugate gradient (CG) code generated by the prototype, compared with the code found in the PETSc version 2.0 beta 2. CG is an iterative linear system solver ([6]). Each iteration executes a matrix-vector multiply and some vector operations (inner products and saxpys). Times include 16 iterations of the solver and do not include preprocessing costs. The codes were run on 1 through 64 nodes of the Cornell Theory Center SP-2. Both codes used the vendor supplied MPI implementation as the communication layer. The matrix A was determined from a 64,000 node 3D mesh. In other words, A had 64,000 rows and columns, and an average of 27 non-zeros per row. A 3D mesh was used so that the problem could be optimally partitioned, thus eliminating the effects of heuristic partitioning techniques. In the full paper we will report the results showing how effective the hybrid set optimizations is. We will also show performance comparisons between the code generated by our compiler and the code in BlockSolve library.

Procs.	Compiler	PETSc
2	9.95	7.48
4	4.86	4.16
8	2.40	2.20
16	1.26	1.42
32	0.70	1.01
64	0.44	0.47

Table 2: Wall clock times for CG, in secs.

4 Optimizations

Let us modify our example somewhat: make X into a "skinny" dense matrix. So that now we compute a product of a sparse matrix A and a dense matrix X. The code is shown in Figure 4. This loop nest (call it SMMM for "skinny matrix-matrix multiplication") is an important kernel in

```
DO $i = 1, n$
    DO $j = 1, n$
        DO $k = 1, m$
            IF $\langle i, j \rangle \in A$ THEN
                $Y(i, k) = Y(i, k) + A(i, j) * X(j, k)$
            ENDIF
        ENDDO
    ENDDO
```

Figure 4: Skinny matrix-matrix product ($m \ll n$)

many linear system solvers: X becomes a matrix, when we solve a system $\mathbf{A}\mathbf{x}_k = \mathbf{b}_k$ for multiple right-hand sides $\mathbf{b}_k$. One can think of X not as a matrix but as a collection of vectors.

For parallel execution we partition A, Y and X identically by rows. Just as in the SMVM example we do not need to communicate A and Y, but need to communicate X. It is left as an exercise for the reader to show that the view set for X is:

$$\begin{aligned}
\mathcal{V}_X^{(p)} = \pi_(j,k)&\Bigg((\delta(i,p,i') \bowtie A^{(p)}(i',j) \bowtie Y^{(p)}(i',k)) \times \{1 \le k \le m\} \Bigg) \\
&= \{1 \le k \le m\} \times \pi_j\Big(\delta(i,p,i') \bowtie A^{(p)}(i',j) \bowtie Y^{(p)}(i',k)\Big) \\
&= \{1 \le k \le m\} \times \pi_j\Big((\delta(i,p,i') \bowtie A^{(p)}(i',j)\Big) \qquad (29)
\end{aligned}$$

(We have removed the $Y^{(p)}(i',k)$ term, because Y is dense and does not restrict the set in any interesting way.)

This set is a product of two sets: one is in closed form and the other has to be computed at run-time. So we only need to compute, store and communicate the set

$$\pi_j\Big((\delta(i,p,i') \bowtie A^{(p)}(i',j)\Big) \qquad (30)$$

This gives us an m-fold savings both in work and storage.

In general, suppose we have a query:

$$Q = \sigma_{f(\mathbf{x},\mathbf{y})}(R(\mathbf{x}) \times S(\mathbf{y})) \qquad (31)$$

where the relation R can be represented in closed form (say, as a collection of linear equalities and inequalities) and S can not be represented in closed form. Let $g(\mathbf{y})$ be the projection of the constraints in f onto $\mathbf{y}$ ([16]). Then we can rewrite this query as:

$$Q = \sigma_{f(\mathbf{x},\mathbf{y})}(R(\mathbf{x}) \times \sigma_{g(\mathbf{y})}S(\mathbf{y})) \qquad (32)$$

Say, we want to send the tuples in Q from one processor to another. Then it is enough to send $Q' = \sigma_{g(\mathbf{y})}S(\mathbf{y})$ – the whole query Q can be reconstructed from Q'. Basically, we only need to send what the other processor does not already know. It might actually be the case that Q' has more elements than Q, so the decision to break the query up is heuristic. One case when it is definitely profitable to do is if we can separate f into the constraints in $\mathbf{x}$ and $\mathbf{y}$. Then the query Q is in fact a cross product of Q' and a closed-form set. This is precisely the case in the SMMM example.

A degenerate case is when the whole query Q can be represented in closed-form. This is what happens when regular codes are compiled: the communication sets are just combinations of linear constraints (see [2] or [3]). In this case we can eliminate the actual computations, storage and communication of the sets (26) and (27) by a variation on constant propagation: the lattice of values is

$$\begin{tikzcd}
\top = \text{unknown set, computed at run-time} \arrow[d, no head] \\
\text{a hybrid set} \arrow[d, no head] \\
\text{a closed-form set} \arrow[d, no head] \\
\bot
\end{tikzcd}$$

Now compilation of regular (dense) codes is just a special case of compilation of sparse codes!

5 Two uses of the compiler

We began this paper by suggesting that it be used to generate SPMD code for programs interfaced with a discretization system. In such a scenario, the user writes a dense program for the solver, whereas the discretization system supplies the information about storage formats and placement (see the diagram in Figure 5).

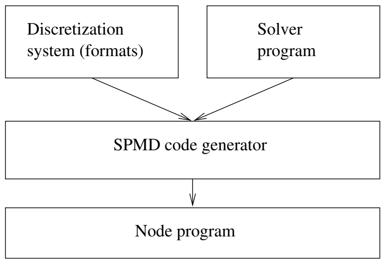

Figure 5: Using our compiler with a discretization system

But since we can also generate SPMD code for dense programs with regular distributions, our compiler can be used as a code generation engine within a regular HPF compiler: the HPF part analyzes alignment/distribution directives and computes local storage formats as described, for example, in [3]. This translates HPF partitioning directives into full placement directives. Our compiler can then be used to generate the node programs (see the diagram in Figure 6).

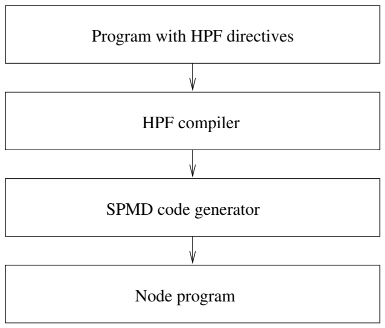

Figure 6: Using our compiler as code generation engine

6 Conclusions

In this abstract, we have presented a relational framework for generating SPMD code for sparse matrix computations, which can be represented as dense computations annotated with sparsity formats. The highlight of this approach are:

- we are able to use arbitrary sparse matrix storage formats and distribution formats
- dense (regular) SPMD code generation is just a special case of sparse SPMD code generation in our framework

- we can recognize hybrid dense-sparse sets and optimize their storage, computation or communication costs
- our compiler can be used as a stand alone module or as a SPMD code generation engine for an HPF Fortran-like compiler. In the latter case this would enable an HPF compiler to unify treatment of regular and irregular codes.

In the full paper we will provide more details in the code generation algorithm, as well as more experimental data, especially to demonstrate the importance of hybrid set optimization.

We are working on extending this work to loops with dependencies.

References

[1] Gagan Agarwal, Alan Sussman, and Joel Saltz. An integrated runtime and compile-time approach for parallelizing structured and block structured applications. *IEEE Transactions on Parallel and Distributed Systems*, July 1995. Also available from http://www.cs.umd.edu.

[2] Saman P. Amarasinghe and Monica Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the PLDI '93*, June 1993.

[3] Corinne Ancourt, Fabien Coelho, Franois Irigoin, and Ronan Keryell. A linear algebra framework for static hpf code distribution. In *CPC'93*, November 1993. Also available at http://cri.ensmp.fr/doc/A-250.ps.Z.

[4] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of PLDI'93*, June 1993. available at http://suif.stanford.edu/papers/-anderson93/paper.html.

[5] Argonne National Laboratory. *PETSc, the Portable, Extensible Toolkit for Scientific Computation.* available at http://www.mcs.anl.gov/petsc/petsc.html.

[6] Richard Barrett, Michael Berry, Tony Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, 1994. Also available from ftp://ftp.netlib.org/-templates/templates.ps.

[7] David Bau, Induprakas Kodukula, Vladimir Kotlyar, Keshav Pingali, and Paul Stodghil. Solving alignment using elementary linear algebra. In *Proceedings of the 7th LCPC Workshop*, August 1994. Also available as Cornell Computer Science Dept. tech report TR95-1478.

[8] A. J. C. Bik and H. A. G. Wijshoff. Compilation techniques for sparse matrix computations. Technical Report 92-13, Leiden University, Department of Mathematics and Computer Science, 1992. available at file://ftp.wi.LeidenUniv.nl/pub/CS/TechnicalReports/1992/tr92-13.ps.gz.

[9] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S.-H. Teng. Optimal evaluation of array expressions on massively parallel machines. *ACM Transactions on Programming Languages and Systems*, 1995.

[10] Department of Computer Sciences, Purdue University. *Parallel ELLPACK PDE Solving System.* available at http://www.cs.purdue.edu/research/cse/pellpack/pellpack.html.

[11] Guy Edjlali, Alan Sussman, and Joel Saltz. Interoperability of data parallel runtime libraries with meta-chaos. Technical Report CS-TR-3633, University of Maryland, May 1996. Also available at ftp://hpsl.cs.umd.edu/pub/papers/Meta-Chaos-tr96.ps.Z.

[12] Alan George and Joseph W-H Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice Hall, Inc., 1981.

[13] Bruce Hendrickson and Rober Leland. A multilevel algorithm for partitioning graphs. Technical Report SAND93-1301, Sandia National Laboratories, 1993.

[14] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. A relational approach to sparse matrix compilation. Submitted to *EuroPar '97 conference*, 1997.

[15] Paul Plassman and M.T. Jones. Blocksolve95 users manual. Technical Report ANL-95/48, Argonne National Laboratory, 1995.

[16] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, August 1992.

[17] Manuel Ujaldon, Emilio Zapata, Barbara M. Chapman, and Hans P. Zima. New data-parallel language features for sparse matrix computations. Technical report, Institute for Software Technology and Parallel Systems, University of Vienna, 1995. Available at http://www.vcpc.univie.ac.at/activities/language.

[18] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, v. I and II.* Computer Science Press, 1988.

[19] Rinhard v. Hanxleden, Ken Kennedy, and Joel Saltz. Value-based distributions and alignments in Fortran D. Technical Report CRPC-TR93365-S, Center for Research on Parallel Computation, Rice University, December 1993.

[20] Janet Wu, Raja Das, Joel Saltz, Harry Berryman, and Seema Hiranandani. Distributed memory compiler design for sparse problems. *IEEE Transactions on Computers*, 44(6), 1995. Also available from ftp://hyena.cs.umd.edu/pub/papers/ieee_toc.ps.Z.