Automatic parallelization of vector parallel codes for preconditioned iterative solvers

Oleg Batrashev
Institute of Computer Science
University of Tartu
Tartu, Estonia
olegus@ut.ee

Eero Vainikko
Institute of Computer Science
University of Tartu
Tartu, Estonia
eero@ut.ee

ABSTRACT

There is a lot of effort to make programming for HPC more productive and we are to make our contribution. After gaining some experience in programming preconditioned iterative solvers in Fortran and MPI we propose new approach, that is based on the mixed ideas from vector parallel languages and parallelizing compilers like HPF. We follow two rules, first try to vectorize our code as much as possible and then apply static analysis techniques to the new representation to get parallelized code automatically. This paper describes our motivation and walks through major steps of our approach without looking into much details. Our wish is to get feedback from the community in early stage of the research.

Categories and Subject Descriptors

E.1 [**DATA STRUCTURES**]: Distributed data structures; F.3.2 [**LOGICS AND MEANINGS OF PROGRAMS**]: Semantics of Programming Languages—*Program analysis*

General Terms

Languages, Experimentation

Keywords

iterative solver, irregular problem, parallelizing compiler

1. INTRODUCTION

Iterative solvers of linear equations are always portrayed as easy to parallelize when compared to direct solvers. These algorithms exist for several decades and gained a lot of popularity, but their development is still done using explicit communication commands in low-level languages. Probably, the reason is that this domain is too narrow to develop specific language for it alone, while the patterns of parallel algorithms from other domains are too complicated.

In the last years there has been a lot of interest towards languages for HPC: Fortress, X10, and Chapel are trying to provide HPC specific abstractions. Apart from that, languages and libraries for vector parallel programming using Nested Data Parallelism (NDP) are gaining attention, particularly Intel Array Building Blocks (ArBB) and Data-Parallel Haskell. However, the latter are supposed for shared memory architectures and this is not enough for iterative solvers, which are mainly used on massively parallel computers. A decade ago there was a lot of interest in data-parallel languages and parallelizing compilers, all revolving around High Performance Fortran (HPF). All these HPF related approaches have eventually disappeared, because of some technical and sociological mistakes as explained in [7].

This is our position paper about upcoming research that describes the approach different from the mentioned solutions. Having some experience in programming parallel iterative solvers [3, 9], we decided to develop a solution for iterative solvers alone. It seems possible to express certain iterative solvers in a vector parallel language, which expresses the code using abstract high-level operations on arrays (higher than in SIMD instruction-sets like SSE). The ideas similar to those in HPF extensions can be used to automatically parallelize such code. Automatic distribution seems complex and often domain specific – we assume that some initial distribution is given by user.

The paper first presents the domain of preconditioned iterative solvers in Section 2, the related work is shortly described in Section 3, then the new approach is described. Section 4 presents vectorized representation of some code from our domain and Section 5 walks through some static analysis techniques from our solution. The last section is about the results and future work.

2. DOMAIN DESCRIPTION

While there are different iterative solvers and preconditioners, we have experience with two-level Schwarz preconditioners that combine Additive Schwarz preconditioner on subdomains and a coarse grid preconditioner: $M^{-1} = M_{AS}^{-1} + M_C^{-1}$. This experience comes from the collaboration between University of Bath and University of Tartu and the result is the DOUG package [1].

2.1 Iterative solvers

Iterative solvers most often use some type of preconditioned Krylov subspace method, e.g Conjugate Gradient (CG). They use two main types of operations in its core iteration: vector dot product and sparse matrix vector multiplication.

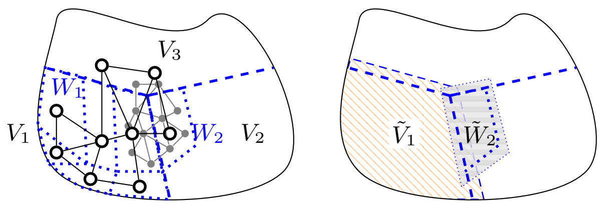

Figure 1: Two-level grid and overlaps

Vector dot product is very expensive for distributed memory systems, because it requires synchronization. Sparse matrix-vector multiplication includes irregular access patterns that requires communication with neighbor processes. The communication-computation ratio largely improves when preconditioner is added to the solver. Moreover, two-level preconditioner allows for substantial overlap of computation and communication.

2.2 Two-level grid

Iterative solvers usually solve problems that arise from the discretizations of partial differential equations on 2D or 3D domains. The final grid may be constructed top-down starting from coarse mesh and refining it where necessary.

DOUG takes the opposite approach because it reads a system matrix without any knowledge of the initial geometry. It combines neighboring nodes using the matrix, which gives a partitioning of the fine grid W_i, $i = 1 \dots n_a$ (see partitions separated with dotted lines on Figure 1). The coarse grid is built as the dual graph of this partitioning where each partition W_i corresponds to a single coarse grid node. Next, the partitioning of the coarse grid V_i, $i = 1 \dots n_s$ is performed using the same technique, combining neighboring coarse nodes.

The two-step partitioning is done locally on each process after the initial grid and the system matrix A are distributed. The partitions of both levels are used in preconditioners: fine grid partitions as *supports* in coarse grid preconditioner and coarse grid partitions as *subdomains* in Schwarz preconditioner. Partitions are usually extended by a few layers which creates *overlaps* (see Figure 1). This creates additional work in managing communication and synchronization of the values before and/or after preconditioning.

2.3 Schwarz preconditioner

In Schwarz preconditioner the original domain is divided into (non-)overlapping subdomains, usually large enough to fit into one computer memory, although several subdomains per process may in some cases give better results. On each subdomain a process solves the local problem, limited to the subdomain freedoms, and then injects it back to the whole domain. The values on the overlap are just added together in case of Additive Schwarz method

$$M_{AS}^{-1} = \sum_{i=1}^{n_s} R_i^T A_i^{-1} R_i.$$

The subdomain injection matrices R_i are also used in the construction of local matrices $A_i = R_i A R_i^T$. These are just minors of the system matrix A with the rows and columns corresponding to the subdomain freedoms. Communication is required on the overlap of the subdomains if they belong to different processes.

2.4 Coarse grid preconditioner

Second level of the two-level preconditioner is based on the coarse grid, which is constructed on top of the fine grid using so called coarse space basis function supports. These supports is just another partitioning of the fine grid into (non)overlapping partitions, though they are smaller than subdomains. In DOUG the partitioning is done using smoothed aggregation algorithm [11]. The coarse grid preconditioner is then expressed as

$$M_C^{-1} = R_C^T A_C^{-1} R_C,$$

where R_C is the coarse space restriction matrix and the matrix $A_C = R_C A R_C^T$ defines the problem for the coarse space. After the problem is solved the solution is interpolated back to the fine grid using interpolation matrix R_C^T. The conceptual difference from the Schwarz preconditioner is that only a single problem needs to be solved but it uses the values from the whole grid and then interpolates them back – the problem is not local.

There are quite sophisticated methods to construct coarse grid preconditioners[13], but the implementation of the corresponding algorithm is often based on the overlaps of the partitions of both levels.

3. RELATED WORK

Before describing the solution we'll give an overview of the existing solutions. We are not aware of any analogous attempts, the closest is probably HPF with its extensions.

3.1 DUNE

One possible approach to simplify the programming of preconditioners is to use some mesh library like DUNE [4] – the Distributed and Unified Numerics Environment. DUNE distinguishes overlapped regions and allows to exchange the values associated with the regions. This is good approach when overlaps are simple, it is not so in complex preconditioners. Automatic calculation of array sections that need to be exchanged is missing from library-based solutions, because a library does not have access to language constructs.

3.2 HPF and its extensions

HPF is the most famous data-parallel language with distributions given by a programmer and automatic communication generation. While HPF-1 standard and its implementations did not include support for irregular array distributions, HPF-2 did include required directives, but the momentum was lost.

Much work has been put into data-parallel languages and parallelizing compilers: Vienna-Fortran [12], Fortran D [10], POLARIS [6], PARADIGM [8] to name some. It seems that the approach is not easy but worthy if can be successfully implemented. Automatic calculation of required communication is present in these solutions, however all of them allow arbitrary code with access to single array elements and all

Algorithm 1 Sparse mv multiplication in ArBB

```
void Ax(const Matrix &A,
      const dense<f64> &x,
      dense<f64> &y)
{
  dense<f64> colvals = gather(x, A.cols);
  dense<f64> mvals = colvals * A.vals;
  nested<f64> nmvals =
     reshape_nested_offsets(mvals, A.nrows);
  y = add_reduce(nmvals);
}
```

array creation	`y=zeros(N), y=copy_like(x)`
array copy	`y=copy(x)`
element-wise	`y=sqrt(x)`
binary el-wise	`z=x+y, z=x*y, z=x==y`
reduction	`r=reduce(x, op)`
gather	`z=x[y]`
scatter	`z[y]=x`
combining scatter	`z=hreduce(y, x, op='+')`, i.e. `z[y]+=x`
gather indices	`z=index(x)`
set ∈	`z=set_in(x,y)`
set ∪	`z=set_union(x,y)`
inverse array	`z=inverse(x)`

Table 1: array operations: x, y, and z are arrays

kind of loops. This complicates compile-time analysis, indeed most of the papers on HPF extensions are devoted to some form of *loop analysis*.

3.3 Nested Data Parallelism

Recently there has been increased interest in Nested Data Parallelism (NDP), namely Data-Parallel Haskell [5] and Intel Array Building Blocks [2] (ArBB). NDP represents irregular structures as nested arrays and allows a limited set of irregular operations on the arrays. Algorithm 1 shows the implementation of sparse matrix-vector multiplication in Intel ArBB.

The main observation here – though the arrays are read-only and the semantics of each array operation is to create new array, the final code fuses the operations together. Intermediate arrays that are supposed by language semantics may never be created, so the performance does not degrade. Moreover, it may make easier code optimizations for the target platform because all operations are high-level. Unpreconditioned iterative solvers may already be expressed by the current NDP implementations, however for preconditioners more sophisticated irregular operations are needed.

4. VECTORIZED REPRESENTATION

One of the goals of our research is to express DOUG code as much as possible using vectorized code, leaving parts with unvectorized code for sequential execution. This requires more abstractions than are currently present in the NDP solutions. First, lets define the set of high-level vector operations for the new language (see Table 1).

First operations are intuitive, e.g. binary element-wise compare returns boolean array z. The *gather* operation `z=x[y]` picks elements of x using index array y, the *scatter* operation stores array x values at the locations specified by the index array y. If two values collapse, *combining scatter* should be used to specify reduction operator[1], otherwise the result is unspecified.

Algorithm 2 Dot product and sparse mv multiplication

```
def dot(x,y):
   z = x*y
   return ops.reduce(z,'+')

def Ax(A, x):
   tmp = x[A.icols]*A.vals
   y = ops.hreduce(A.irows, tmp, like=x)
   return y
```

Last four operations are the most complex: `z=index(x)` lists indices of *true* elements of boolean array x. The two other operations treat arrays as sets: `z=set_in(x,y)` checks for each x element whether its value is present in the array y, `z=set_union(x,y)` combines two arrays as sets, i.e. duplicates are eliminated. The last operation `z=inverse(x)` treats array as a function with integer co-domain and creates the array that reflects the inverse function. It is used for mapping global to local subdomain indices in Schwarz preconditioner.

Python was chosen as a prototype language, because of handy NumPy library that works with numeric arrays and Python *ast* package providing compiler front-end. Only limited set of Python constructs and NumPy functions may be used. All of the basis array operations are defined in Python by the *ops* package.

4.1 Data structures

Simple data structures, without inheritance, are allowed in the language. Matrix is represented in triple format with three arrays: `irows`, `icols` and `vals`, that are stored to a `SparseMatrix` object.

```
class SparseMatrix:
   def __init__(self,m,n,nnz):
      self.m = m
      self.n = n
      self.nnz = nnz
      self.irows = ops.zeros(nnz, int)
      self.icols = ops.zeros(nnz, int)
      self.vals = ops.zeros(nnz, float)
```

4.2 Vectorized code examples

Together with element-wise vector operations, solver uses dot product and sparse matrix-vector multiplication. Algorithm 2 shows the implementation, which is almost the same as for Intel ArBB.

4.2.1 Schwarz preconditioner

Schwarz preconditioner requires operations that are not present in the NDP libraries. Construction of the local matrix for

[1]combining scatter is sometimes referred to as *histogram reduction*, this is why its name is `hreduce`

Algorithm 3 Local matrix A_i for a single subdomain

```
def __init__(self, A, d):
   r = ops.set_in(A.irows, d)
   c = ops.set_in(A.icols, d)
   tb = r * c
   t = ops.index(tb)
   irows = A.irows[t]
   icols = A.icols[t]
   d_inv = ops.inverse(d)
   Al = sparse.SparseMatrix(d.size, d.size,
                            irows.size)
   Al.irows = d_inv[irows]
   Al.icols = d_inv[icols]
   Al.vals = A.vals[t]
   self.d = d
   self.Al = Al
```

Algorithm 4 Application of Schwarz preconditioner for a single subdomain

```
def apply_prec(self, r):
   N_ITER=8
   x = ops.zeros_like(r)
   rl = r[self.d]
   xl = stat.sym_gauss_seidel(self.Al, rl,
                              N_ITER)
   x[self.d] = xl
   return x
```

one subdomain and application of the preconditioner on this subdomain are shown by Algorithm 3 and Algorithm 4. They use 3 out of 4 complex vector operations, and the last – `set_union` operation – is needed to extend a domain and create overlaps (see Algorithm 5).

Although, this is large part of Schwarz preconditioner, some high level data abstraction, like nested array, is needed to hold all subdomains and special loop operation that applies the same function to each subdomain. This research still needs to be done.

4.2.2 *Coarse grid preconditioner*

There is currently no attempt to implement coarse grid preconditioner with vectorized code. Most of it can be vectorized, only sparse matrix-matrix multiplication seems hard to abstract and parallelize.

4.3 Motivation for code analysis

We believe several code optimizations can be done using the representation given above. Moreover, array distribution and distributed array synchronization points can often be derived from the code.

Algorithm 5 Add one layer to a domain

```
def add_layer(domain, A):
   r = ops.set_in(A.irows, domain)
   t = ops.index(r)
   v = A.icols[t]
   newDomain = ops.set_union(domain, v)
   return newDomain
```

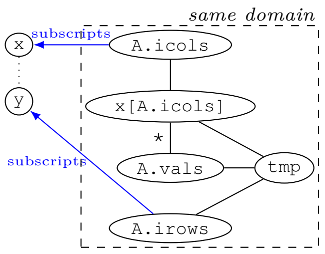

Figure 2: Array relations in mv multiplication

4.3.1 *Array relations*

Matrix vector multiplication in Algorithm 2 induces certain relations between the arrays (see Figure 2). First type of relation, depicted with solid line, tells that the arrays are *aligned,* i.e. equal length (same domain) and their corresponding elements are expected to be present on the same process for some operation, e.g. element-wise multiplication. The same relation appears in the gather operation, where the indexing array `A.icols` is aligned with the resulting array `x[A.icols]`, and in the combining scatter operation, where the indexing array `A.irows` is aligned with the source array `tmp`. The alignment of `x` and `y` is derived from the caller code.

Another type of relation is array subscription which appears in gather and scatter operations. This is shown with arrowed lines on Figure 2. In DOUG the distribution of matrix values is computed from the distribution of vector `y` and the values of `A.irows`, so that the combining scatter is a local operation. The communication is needed in the gather operation, where `A.icols` subscripts `x`. Here the values of `x`, that are not stored locally – the *ghost values,* need to be updated before the gather operation can take place.

Note, that the roles of these two array subscription operations can be exchanged, so that the relation from the gather operation is used for the computation of distribution and the relation from the combining scatter operation for the derivation of ghost values. In this case the communication needs to be done as the final phase of the combining scatter operation.

Algorithm 4 contains the function call (`sym_gauss_seidel`) that cannot be parallelized. All the arrays that are passed to or returned from it are completely local, so are `rl` and `xl`. The array `self.d` is aligned with them through subscription operations and hence should also be stored to a single process. This array in another function in Algorithm 3 is used to construct several other arrays using complex operations `set_in`, `index` and `inverse`. The exact distribution and communication behavior is not yet defined for these operations, although we assume the propagation of such information using static code analysis may be helpful.

In general, there are usually more complex situations when several subscription operations define the distribution and the ghost values. In such cases overlaps, when elements are replicated over several processes, are typical. Duplicate exclusion has to be performed for some operations, and we feel that this analysis is better addressed by the compiler

Algorithm 6 IR of the Ax() function

```
0   = A.icols       : A(INT)
1 := x[0]           : A(FLOAT)
2   = A.vals        : A(FLOAT)
3 := 1 * 2          : A(FLOAT)
tmp = 3             : A(FLOAT)
5   = A.irows       : A(INT)
6   = ops.hreduce   : oF(hreduce)
7   = 6(5,tmp,x)    : A(FLOAT)
y   = 7             : A(FLOAT)
return = y          : A(FLOAT)
```

and its supporting library.

4.3.2 *Synchronization points*

Currently, in our prototype the synchronization of distributed arrays is done at the time of operation. By using code analysis this point may be shifted in the code even to another subroutine, enabling more communication-computation overlap. In DOUG this kind of optimizations for the Schwarz and coarse grid preconditioners are done explicitly, which largely tangles the code.

The synchronization of an array may take different forms. For example, DUNE [4] defines *additive representation* for array x as $x = \sum_{i=0}^{P-1} R_i^T x_i$, where x_i are local parts of the array. Recall from Section 2 that the preconditioner is defined as the sum of the preconditioners of both levels

$$z = M^{-1}r = M_{AS}^{-1}r + M_C^{-1}r.$$

The sum of local arrays with additive representations gives the array with additive representation, so the synchronization may be shifted to the last minute. In DOUG this kind of optimization is also done by the programmer.

5. COMPILE-TIME ANALYSIS

This section describes the foundations of compile-time analysis with some data-flow analysis schemes, which are used in the solution. It only sketches the approach without going into details.

5.1 Intermediate Representation

In order to analyze the code it must be represented in convenient format, e.g. *three address code.* Expressions in the code are split into sequences of IR (Intermediate Representation) commands, which only accept variable names or *block variables* as arguments. Algorithm 6 shows the IR code for the `Ax()` function from Algorithm 2.

Block variables are temporary variables in block that appear after splitting expressions into IR commands. These variables are simply named by the IR command index in the block and written as numbers in the example.

The set of IR commands partly mirrors the array operations from the previous section, some important IR commands are listed in Table 2. Complex array operations do not have corresponding IR commands, they are just function calls and are recognized by the *ops* module and their name. Binary command may be applied to scalar or array types, field read

Alias	$x = y$
Const	$x := \text{c}$
Binary	$x := y \otimes z$
Call	$x = y(z_1, \dots, z_n)$
AttrUse	$x = y.z$
AttrDef	$x.y = z$
SubscriptUse	$x := y[z]$
SubscriptDef	$x[y] = z$

Table 2: IR commands

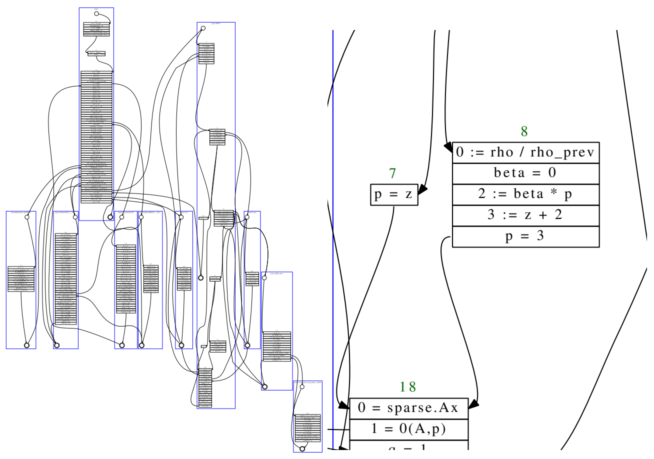

Figure 3: Control-flow graph from basic blocks

and write operations (AttrUse, AttrDef) are applied to objects.

5.2 Control-flow analysis

The classical approach in static analysis is to split analyzed program/function into *basic blocks,* which contain only sequentially executed IR commands and do not have branching. All the branching is introduced in the control-flow graph (CFG) that connects basic blocks based on the flow of control (see Figure 3).

The control-flow graph is the main structure on which different types of data-flow analysis work. The idea is to walk the graph and find constraints that hold on basic block entry/exit and inside between IR commands.

5.3 Data-flow analysis

In this section we give very brief introduction to data-flow analysis and the schemes that are currently used by the solution. Data-flow analysis computes data-flow values before (IN[s]) and after (OUT[s]) each statement s. The value may be pretty much anything of interest, e.g. the knowledge that a variable can contain only one certain value irrespectively of execution paths that lead to the statement (Constant Propagation analysis) or the knowledge that a variable may still be used after the statement at some following execution path (Live Variable analysis).

Generally, in forward data-flow analysis:

- *transfer function* f_s computes the changes to the data-flow value when the statement s is executed $\mathrm{OUT}[s] = f_s(\mathrm{IN}[s])$,

- the IN value is the same as the OUT value of the previous statement in the basic block $\mathrm{IN}[s_i] = \mathrm{OUT}[s_{i-1}]$,
- the transfer function of a block B is the composition of statements transfer functions $f_B = f_{s_n} \circ f_{s_{n-1}} \circ \cdots \circ f_{s_1}$,
- the input value $\mathrm{IN}[B]$ of a basic block B is computed by a *join operator* combining output values of its predecessor blocks in the control-flow graph

$$\mathrm{IN}[B] = \bigwedge_{B' \text{ is a pred. of } B} \mathrm{OUT}[B'].$$

5.3.1 *Points-to analysis*

From the section about our motivation for code analysis it may be clear that we need to track arrays in the code and investigate in which operations they meet. The classical way is to use Reaching Definitions, that tracks which *definitions* of a variable may reach what code locations. The more elaborate is the Pointer analysis that also tracks aliases and object field references.

Pointer analysis finds which memory locations a pointer may refer to. For languages like C it is extremely difficult, because C allows pointer arithmetics and pointer may refer to anything. In our case we limit the model:

- a program contains variables and heap objects,
- a variable may point to a heap object,
- a heap object has fields that may point to other heap objects but not variables.

Consider the following example with 2 definitions (d_3 does not act as definition):

d_1:	`arr=ops.zeros(n)`
d_2:	`A=SparseMatrix()`
d_3:	`A.irows=arr`

Here the first statement creates array heap object and `arr` variable points to it, the second statement creates heap object that is `SparseMatrix` instance, and the third statement makes the `irows` field of the `SparseMatrix` instance refer to the first array.

In our analysis we need somehow to represent heap objects, and the usual way to do it by the statement where the object is created. The variable `arr` then points to the heap object d_1, the variable `A` points to d_2, and the field `irows` of the heap object d_2 points to d_1. The program may execute the same code several times and create multiple objects from the same statement, however we do not distinguish them. Our data-flow value is a map m, so that

$$\begin{aligned} m[\mathrm{x}] &= \{d | \text{variable x points to } d\} \\ m[(d', f)] &= \left\{d | \text{field } f \text{ of the object } d' \text{ points to } d\right\}. \end{aligned}$$

The transfer function for a statement s usually changes the map $\mathrm{OUT}[s] = m' = f_s(m) = f_s(\mathrm{IN}[s])$. If we ignore the arguments that map to an empty set and represent the fact $m[a] = Y$, $Y \neq \emptyset$ as the set element $a \to Y$, then our example has the following data-flow values:

$$\begin{aligned} \mathrm{IN}[d_1] &= \emptyset \\ \mathrm{OUT}[d_1] &= \{\mathrm{arr} \to \{d_1\}\} \\ \mathrm{OUT}[d_2] &= \{\mathrm{arr} \to \{d_1\}, \mathrm{A} \to \{d_2\}\} \\ \mathrm{OUT}[d_3] &= \{\mathrm{arr} \to \{d_1\}, \mathrm{A} \to \{d_2\}, (d_2, \mathrm{irows}) \to \{d_1\}\}. \end{aligned}$$

The transfer function for the *alias* command $d : x = y$ is

$$\mathrm{OUT}[d][x] = \mathrm{IN}[d][y],$$

the set with definitions representing heap objects, that the variable may point to, just copied to the target variable x. The transfer function for most IR commands $d : x := c$ creates new heap object and kills all other definitions with the target variable x :

$$\mathrm{OUT}[d][t] = \begin{cases} \{d\} & \text{if } t = x \\ \mathrm{IN}[d][t] & \text{otherwise} \end{cases}.$$

For the field read command $d : x = y.z$ we have to get all heap objects that y may refer to and combine all definitions their z field may refer to:

$$OUT[d][t] = \begin{cases} \cup_{d' \in IN[d][y]} IN[(d', z)] & \text{if } t = x \\ IN[d][t] & \text{otherwise} \end{cases}$$

The join operator is the union for all map arguments

$$\mathrm{IN}[B][t] = \bigcup_{B' \text{ is a pred. of } B} \mathrm{OUT}[B'][t].$$

5.3.2 *Interprocedural analysis*

Points-to analysis requires tracking of references to heap objects in called procedure, otherwise any field of any object passed to or indirectly accessible in the procedure may change. This will make points-to analysis very imprecise, so *interprocedural analysis* is required.

When a procedure is called, its parameter variables get definitions from the actual arguments and all field definitions are copied to the start block of the function. On exit the definitions of return value are copied to the target variable of the function call command and all new field definitions are reassigned.

If a procedure is called from several locations in the code, the IN map of its start block and OUT map of its end block get polluted by the definitions from all these locations. The interprocedural analysis has to be context-sensitive and the simplest way is to clone procedure for each call location. *Cloning-based context-sensitive* interprocedural analysis distinguishes data-flow values for a procedure called from different locations. The implementation of this analysis is not yet included in the solution but we are working on it.

5.4 Distribution propagation

After we know which array definitions may reach what code locations, we may start to analyze how the arrays should be distributed depending on the commands they are accessed in. If we know the distribution of one array we may compute

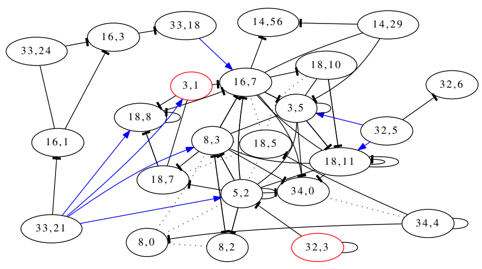

Figure 4: Graph of definition relations in CG

desirable distributions of other arrays – *distribution propagation.*

For example, in command $d: x = y + z$ all three arrays that reach this command in the variables x, y, and z should be distributed in the same way between processes. This is like *align* directive in HPF languages, only implicit in the code and not explicitly defined. The definitions in y and z are symmetric and are related with Eq relation

$$\text{Eq}(d_1, d_2)\, \forall d_1 \in \text{IN}[d][y],\, \forall d_2 \in \text{IN}[d][z].$$

The definitions in x are related to others with asymmetric AEq relation

$$\text{AEq}(d_1, d_2)\, \forall d_1 \in \text{IN}[d][x],\, \forall d_2 \in \text{IN}[d][y] \cup \text{IN}[d][z].$$

The corresponding relations depicted as a solid line and a line with brick at one end on Figure 4. The value inside each definition are block number and IR command number inside the block, which is the way definitions d_i are currently stored. In case of context-sensitive interprocedural analysis calling context must also be added.

One relation of special interest is array subscription by another array $d: x = y[z]$. Knowing the distribution of y we may calculate the distribution of z by the inspection of z values. This way all the values needed for the operation will be locally available, because the distribution of x is likely to be the same as of z. The arrowed connections on Figure 4 represent indexing, one definition subscripts the definition the arrow is pointing to.

6. RESULTS AND FURTHER WORK

As a result we have proof of concept application – CG algorithm – which is analyzed by our framework and parallelized by inserting required communication commands. The efficiency and generality of the implementation is not under consideration at the moment, we will turn to them after the main concepts are in place.

Our next steps will be: finish context-sensitive interprocedural analysis, precisely describe all parts of the algorithm and its limitations, and measure the efficiency of the approach. After that the coarse grid preconditioner is to be vectorized and our framework should probably be expanded by new types of analyses. Extending our framework to other types of preconditioners, e.g. Algebraic Multigrid (AMG), and algorithms, i.e. outside of iterative solvers, is possible in the far future.

We will constantly inspect the state of recent HPC languages and data-parallel frameworks and compare our approach with other solutions. The feasibility and efficiency of our solution is not yet clear.

7. REFERENCES

[1] DOUG development pages. http://dougdevel.org/.

[2] Intel® array building blocks (ArBB). http://software.intel.com/en-us/articles/intel-array-building-blocks/.

[3] Oleg Batrashev, Satish N. Srirama, and Eero Vainikko. Benchmarking DOUG on the Cloud. HPCS 2011. (Accepted for Publication).

[4] Markus Blatt and Peter Bastian. On the generic parallelisation of iterative solvers for the finite element method. *International Journal of Computational Science and Engineering*, 4(1):56–69, 2008.

[5] Manuel M. T Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, DAMP '07, page 10–18, New York, NY, USA, 2007. ACM.

[6] E. Gutiérrez, R. Asenjo, O. Plata, and E. L. Zapata. Automatic parallelization of irregular applications. *Parallel Computing*, 26(13-14):1709–1738, December 2000.

[7] Ken Kennedy, Charles Koelbel, and Hans Zima. The rise and fall of high performance fortran: an historical object lesson. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 7–1–7–22, New York, NY, USA, 2007. ACM.

[8] Antonio Lain. *Compiler and Runtime Support for Irregular Computations.* PhD thesis, University of Illinois, 1996.

[9] Christian Pocher, Oleg Batrasev, Ulrich Norbisrath, and Eero Vainikko. Dougflow – offering scientific applications via web services. *Internet and Web Applications and Services, International Conference on*, 0:487–492, 2009.

[10] Ravi Ponnusamy, Yuan-Shin Hwang, Raja Das, Joel H. Saltz, Alok Choudhary, and Geoffrey Fox. Supporting irregular distributions using Data-Parallel languages. *IEEE Parallel Distrib. Technol.*, 3(1):12–24, 1995.

[11] R. Scheichl and E. Vainikko. Additive schwarz and Aggregation-Based coarsening for elliptic problems with highly variable coefficients. Bath Institute For Complex Systems Preprint 9/06, 2006.

[12] Manuel Ujaldon, Emilio L Zapata, Barbara M Chapman, and Hans P Zima. Vienna-Fortran/HPF extensions for sparse and irregular problems and their compilation. *IEEE Trans. Parallel Distrib. Syst.*, 8:1068–1083, October 1997.

[13] Jan Van lent, Robert Scheichl, and Ivan G. Graham. Energy-minimizing coarse spaces for two-level schwarz methods for multiscale pdes. *Numerical Linear Algebra with Applications*, 16(10):775–799, 2009.