Optimization Techniques for Irregular and Pointer-based Programs[*]

R. Asenjo F. Corbera E. Gutiérrez M.A. Navarro O. Plata E.L. Zapata
Department of Computer Architecture
University of Málaga
P.O. Box 4114, E-29080 Málaga, Spain
{asenjo,corbera,eladio,angeles,oscar,ezapata}@ac.uma.es

Abstract

Current compilers show inefficiencies when optimizing complex applications, both analyzing dependences and exploiting critical performance issues, like data locality and instruction/thread parallelism. Complex applications usually present irregular and/or dynamic (pointer-based) computational/data structures. By irregular we means applications that arrange data as multi-dimensional arrays and issue memory references through array indirections. Pointer-based applications, on the other hand, organize data as pointer-based structures (lists, trees, ...) and issue memory references by means of pointers. This paper discusses optimization/parallelization and program analysis techniques we have developed to instruct a compiler to generate efficient object code from important classes of irregular and pointer-based applications. These techniques are embodied into a methodology that proceeds in three stages: program structure recognition, data analysis and program optimization/parallelization based on code/data transformations.

1. Introduction

Current automatic optimizers and parallelizers obtain very efficient codes from most of the regular applications. Such applications deal with data organized as multi-dimensional arrays where computations are arranged as uniform nested loops. However, the compiler efficiency is generally much lower for other classes of programs, those that include complex computation and/or data structures. In the presence of such program complexities compilers usually run into trouble both analyzing dependences and optimizing performance issues (like data locality, instruction/thread parallelism, an so on).

We may distinguish two important classes of complex applications: irregular and dynamic (pointer-based). In case of irregular applications, it is usual that data is organized as multi-dimensional arrays, as in regular applications, but using array indirections to reference an important set of such data (arrays with subscripted subscripts). We may find many examples of this class of programs in the numerical scientific/engineering field. Pointer-based applications, on the other hand, deal with data organized as complex, pointer-based structures (lists, trees, ...). This data is, therefore, referenced by means of pointers.

```
do i = 1, N
   ...
   compute ξ
   ...
   A(f( i )) = A(f( i )) ⊕ ξ
   ...
enddo
```

(a)

```
while (condition)
{
    ...
    p→data = value;
    p = p→next;
    ...
}
```

(b)

Figure 1. Example of an irregular computation (a) and a dynamic computation (b)

Figure 1 shows example tiny codes for irregular and dynamic computations. The first piece of code represents an irregular histogram reduction, where a reduction array (A) is updated at some points by means of the indirection array (f). A key issue in the optimization/parallelization of this loop includes solving the possible cross-iteration true data dependences due to the indirection array. The second piece of code corresponds to a variable loop where a pointer-based data structure is updated. Similarly to the previous case, this loop may present cross-iteration dependences due to cycles in the pointer-based list.

In this paper we discuss our recent work about developing efficient optimization, parallelization and program analysis techniques for irregular and pointer-based applications. This techniques may be enclosed into an optimization methodology, that can be broken down into three phases: recognition and classification of the complex struc-

[*]This work was supported by Ministry of Education and Culture (CICYT), Spain, through grant TIC2000-1658

ture of the program, data analysis (data structure and memory reference pattern), and selection of an ad-hoc optimization/parallelization technique fulfilling some performance constraints.

We present some of our recent advances in this field, together with a brief survey of solutions appeared in the literarture. Specifically, we designed a framework to parallelize codes with irregular reductions exploiting data locality. This framework also allowed us to classify and compare the various solutions to this problem that other researchers developed. From this methodology we derived a number of efficient locality oriented run-time parallelizing techniques. On the other hand, we developed new shape analysis techniques for pointer-based data structures to enable dependence analysis in dynamic codes. Such techniques may be used to analyze memory references needed to develop efficient optimization and parallelization methods for these codes.

The rest of the paper is organized as follows. section 2 discusses the methodology we use to develop our optimization and parallelization compilation techniques for irregular/dynamic codes. Next, solutions for a widely found irregular computational structure, named irregular reduction, is described. Shape analysis techniques for dynamic data structures are analyzed in section 4. Finally, conclusions are drawn.

2. Methodology for optimizing complex applications

This section describes a methodology for the optimization and/or parallelization of programs with irregular and/or dynamic computation/data structures. We developed techniques to discover certain program (code and data) properties that are essential in the effective optimization, as well as parallelization methods that take advantage of such properties. The optimization methodology proceeds in three stages, as follows:

1. *Recognition of program structure:* Analysis of the computational structure of the program, as well as the data structures used. The aim of this analysis is to discover irregular and/or dynamic features in the code.

2. *Data analysis:* A complete data analysis is needed to enable a broad set of optimizations, as well as discover parallelism. It is also needed to know where and how such parallelization/optimizations can be done. For our target applications this stage becomes very complex. Two important tasks included into this stage are both the analysis of data structures and the analysis of memory references. The first one determines how data is organized and the relationship among different data items. The second analysis discovers how data is referenced and the relationship among these data references.

3. *Program optimization and/or parallelization:* Information resulting from program structure recognition and data analysis allows to decide what specific optimization technique is best suited to be used. We are specially interested in the development of methods that optimize some important program properties, like data locality or parallelism.

```
REAL   A(1:ADim)
INTEGER f$_1$(1:N$_1$, 1:N$_2$,...)
INTEGER f$_2$(1:N$_1$, 1:N$_2$,...)
        ...

do i$_1$ = 1, N$_1$
    do i$_2$ = 1, N$_2$
        ...
            Compute $\xi_1, \xi_2, \ldots$
            A(f$_1$($i_1, i_2, \ldots$)) = A(f$_1$($i_1, i_2, \ldots$)) + $\xi_1$
            A(f$_2$($i_1, i_2, \ldots$)) = A(f$_2$($i_1, i_2, \ldots$)) + $\xi_2$
             ...
        ...
    enddo
 enddo
```

Figure 2. Nested loop with multiple irregular reductions

In the rest of the paper we describe two representative case studies in the context of the described optimization methodology. The first case study, that constitutes an important class of irregular programs, corresponds to codes with irregular reductions. For these codes the three stages in the methodology will be discussed. The second case study will focus on the second stage, data analysis, for general dynamic codes processing pointer-based data structures.

3. Programs with irregular reductions

Many common data organizations used in numerical applications involve irregular memory accesses, in which array elements are referenced by means of indirections. Reduction operations are often found in the context of irregular codes in scientific and numerical applications, representing an important class of irregular problems. Reduction operations are based in commutative and associative operators, like additions, multiplications, and so on.

An example piece of code carrying out multiple irregular reductions inside a nested loop is shown in Figure 2 (it is also known as histogram reduction). A() represents the reduction array (that could be multidimensional), which is updated (the reduction operation is an addition in this example) by means of the subscript arrays $f_1()$, $f_2()$, ... Terms ξ_1, ξ_2, ... represent effective computation.

Considering the methodology described in the previous section, the first stage corresponds to the recognition of the irregular reduction and further what arrays work as reduction array(s) and which ones as subscript arrays. This stage may be accomplished in a compiler through the use of pattern-matching or idiom recognition techniques [18, 2].

Once irregular reductions have been recognized, a data analysis of the code is accomplished. As shown in Figure 2, all relevant data (from the viewpoint of this stage) is organized as arrays, so no further data structure analysis is needed. We next proceed to analyze memory references. Due to the subscripted subscripts, loop–carried data dependences may be present, and they cannot be detected at compile time (due to the subscript arrays). Techniques have been developed to detect this kind of data dependences at run-time [20].

However, because of the associative and commutative properties satisfied by the reduction operator, the possible data dependences due to the array reductions may be overcome by code/data transformations. Such transformations corresponds to the third stage in methodology. In the last few years various code/data transformations that parallelize irregular reduction loops appeared in the literature. Next sections are devoted to discuss, classify and compare the techniquesi more representative, commonly used to automatically parallelize irregular reductions in the context of share memory multiprocessors.

3.1. Methods for Reduction Parallelization

Different specific solutions to parallelize irregular reductions on shared-memory multiprocessors have been proposed in the literature. We may classify them into two broad categories: *loop partitioning oriented* techniques (LPO) and *data partitioning oriented* techniques (DPO). The LPO class includes those methods based on the partitioning of the reduction loop and further execution of the resulting iteration blocks on different parallel threads. A DPO technique, on the other hand, is based on the (usually block) partitioning of the reduction array, assigning to each parallel thread preferably those loop iterations that issue write operations on a particular data block (then it is say that the thread owns that block).

To facilitate the analysis of the above categories, we consider in the rest of the paper the general case shown in Fig 2 but with only one nesting level (the case of multiply nested loops is not relevant for our discussion). Taking into account this example irregular reduction loop, Fig. 3 shows a graphical representation of generic techniques in the described two classes, LPO and DPO.

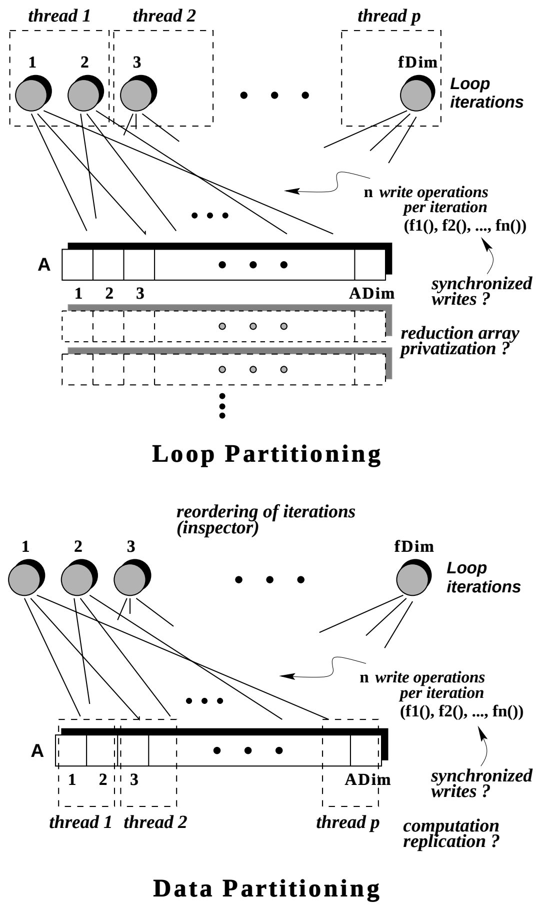
thread 1
thread 2
thread p
1
2
3
fDim
Loop iterations
n write operations per iteration (f1(), f2(), ..., fn())
A
1 2 3
ADim
synchronized writes ?
reduction array privatization ?
Loop Partitioning
reordering of iterations (inspector)
1
2
3
fDim
Loop iterations
n write operations per iteration (f1(), f2(), ..., fn())
A
1 2 3
ADim
synchronized writes ?
thread 1 thread 2
thread p
computation replication ?
Data Partitioning

Figure 3. General schematic representation of the LPO and DPO classes of reduction parallelization techniques

3.1.1 Loop partitioning oriented techniques

The simplest solution in the LPO class is based on critical sections, where the reduction loop is executed fully parallel by just enclosing the accesses to the reduction array in a critical section. This method exhibits a very high synchronization overhead and, consequently, a very low efficiency.

The synchronization pressure can be reduced (or even eliminated) by privatizing the reduction array [16, 23], as it is done by the *replicated buffer* and *array expansion* techniques. The *replicated buffer* method replicates private copies of the full reduction array on all threads. Each thread accumulates partial results on its private copy, and finally the global result is obtained by accumulating the partial results across threads on the global reduction array (this last step needs synchronization to ensure mutual exclusion). The other method, *array expansion*, expands the reduction array by the number of parallel threads. Now each thread accumulates partial results on its own section of the expanded array. This approach allows to obtain the final result in a similar way than the first method, but with no need of the final synchronization.

Note that these two methods transform the reduction loop into a fully parallel one, as the possible loop-carried dependencies disappear as a result of the privatization of the reduction array. However, they have scalability problems for large data sets, as the privatization affect the whole reduction array and on all threads (the memory overhead increases in proportion to the number of parallel threads).

LPO methods basically exploit maximum parallelism in a very balanced way although they are very eager for memory. For this reason, several solutions has been proposed to reduce this high memory overhead, based on the array expansion and replicated buffer basic idea. The *reduction table* method [16] assigns a private buffer to each thread of a fixed size (lower than the size of the reduction array). Then, each thread works on its private buffer indexed by using a fast hash formula. When the hash table is full, any new operation will work directly on the global reduction array within a critical section. Other method is *selective privatization* [23], where the replication include only those elements referenced by various threads. It first determine (inspector phase) which are those elements and then allocate for them private storage space. Each thread, then, works on its private buffer when updating conflicting elements, while it works on the global reduction array otherwise. This execution behavior implies a replication of each subscript array in order to store the new indexing scheme. Also it has been proposed in the literature some adaptive schemes that include an inspector to select the most suitable technique for a given memory pattern [23].

3.1.2 Data partitioning oriented techniques

Methods in the DPO class avoid the privatization of the reduction array, as it is partitioned and assigned to the parallel threads. In order to determine which loop iterations each thread should execute (mostly those that write in its assigned block), an inspector is introduced at runtime whose net effect is the reordering of the reduction loop iterations (through the reordering of the subscript arrays). The selected reordering tries to minimize write conflicts, and, in addition, to exploit data (reference) locality.

Two methods has been proposed in the literature in the DPO class. One method was termed LOCALWRITE [13, 14], and is based on the *owner–computes rule*. Each thread owns a portion of the reduction array (block partitioning). The inspector has reordered the subscript arrays in such a way that, in the execution phase, the set of iterations assigned to that thread only updates array elements of the owned block. Note, however, that, in order to fulfill the computes rule, those iterations that updates more than one block of the reduction array must be replicated across the owner threads. This computation replication introduces a performance penalty (parallelism loss).

```
integer f1(fDim), f2(fDim ), ..., fn(fDim)
real    A(ADim)
integer init (nThreads,0:nThreads−1)
integer count(nThreads,0:nThreads−1)
integer next(fDim)

do ΔB = 0, nThreads−1
   do s = 1, ΔB+1
c$omp parallel do
      do B$_{min}$ = s, nThreads−ΔB, ΔB+1
         i = init (B$_{min}$,ΔB)
         cnt = count(B$_{min}$,ΔB)
         do k = 1,cnt
            Calculate $\xi_1, \xi_2, ..., \xi_n$
            A(f1(i))=A(f1(i)) $\oplus$ $\xi_1$
            A(f2(i))=A(f2(i)) $\oplus$ $\xi_2$
                            ...
            A(fn(i))=A(fn(i)) $\oplus$ $\xi_n$
            i = next(i)
         enddo
      enddo
c$omp end parallel
   enddo
enddo
```

Figure 4. The execution phase of DWA–LIP **in OpenMP (**nThreads **is the number of threads cooperating in the computation)**

An alternative method that avoids computation replication is DWA–LIP [11]. Consider that the blocks of the reduction array are indexed by the natural numbers. The inspector (named *loop-index prefetching* phase, or LIP) now sorts all the iterations of the reduction loop into sets characterized by the pair $(B_{min}, \Delta B)$, where B_{min} (B_{max}) is the minimum (maximum) index of all blocks touched by the iterations in that set, and ΔB is the difference $B_{max} - B_{min}$. The execution phase (or computation phase) of the method is organized as a synchronized sequence of non-conflicting (parallel) stages. In the first stage, all sets of iterations of the form $(B_{min}, 0)$ are executed in parallel because they are all data flow independent (optimal utilization of the threads). The second stage is split into two sub-stages. In the first one, all sets $(B_{min}, 1)$ with an odd value of B_{min} are executed fully parallel, followed by the second sub-stage where the rest of sets are executed in parallel. A similar scheme is followed in the subsequent stages, until all iterations are exhausted. Fig. 4 explains an OpenMP implementation of this method, while Fig. 5 shows the data structures for an example code with two subscript arrays.

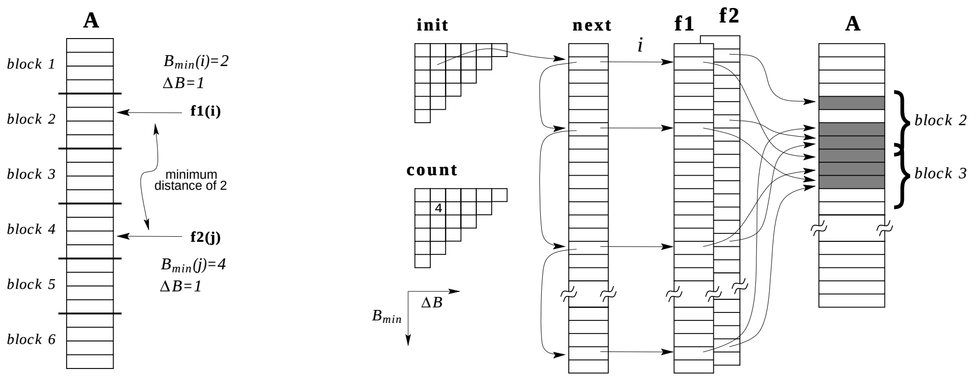

Figure 5. Data structures needed in the execution phase of DWA–LIP (for two subscript arrays)

3.2. Performance Properties of Reduction Methods

Methods in the LPO and DPO classes have, in some sense, complementary performance characteristics. Methods in the first class exhibit optimal parallelism exploitation (the reduction loop is fully parallel), but no data locality is taken into account and lack memory scalability. In addition, as the reduction loop is uniformly partitioned, these methods usually exhibit balanced workload.

Methods in the second class, however, exploit data locality and exhibit usually much lower memory overhead, and it is not dependent on the number of threads (the inspector may need some extra buffering to store subscript reorderings, independently on the number of threads). However, either the method introduces some computation replication or is organized in a number of synchronized phases. In any case, this fact represents loss of parallelism. In addition, there is the risk that the number of the loop iterations that write some specific block is very different from the same in another block (workload unbalance).

Table 1 shows typical characteristics of methods in LPO and DPO classes considering four relevant performance aspects: data locality, memory overhead, parallelism and workload balance. Data locality is in turn split into inter-loop and intra-loop localities. Inter-loop locality refers to the data locality among different reduction loop iterations. Intra-loop locality, on the other hand, corresponds to data locality inside one reduction loop iteration.

Data locality is not exploited by a LPO method. This situation could be relieved by adding an external preprocessing stage before executing the irregular code [13]. Usually these techniques have a high algorithmic complexity.

DPO methods, on the other hand, are designed to exploit, at runtime, data locality, specially inter-loop locality, at the cost of reducing a fraction of parallelism (including computation replication). Intra-loop locality could be, additionally, exploited externally by means of a preprocessing reordering algorithm. Other interesting characteristic is that usually memory overhead is much lower than in basic LPO methods, improving significantly the scalability properties.

In some cases DPO methods may perform under optimal, either due to loss of parallelism (a lot of conflicting interblock writes) or to workload unbalance. Specific solutions for these special patterns have been proposed [12] as a trade-off between memory overhead and degree of parallelism.

In cases where parallelism loss or computation replication is significant it is possible to remove write conflicts by replicating the reduction arrays partially (a fixed number of times less than the number of threads). This optimization is called *partial array expansion* and may improve the behavior of DPO methods for those memory patterns exhibiting a low inter–iteration locality [12].

As DPO methods are based on an uniform block partitioning of the reduction arrays (to exploit inter–iteration locality) certain patterns may introduce workload unbalance. A generic approach to balance the computational load is partitioning the reduction array into small subblocks in a number multiple of the number of threads. This way, blocks of different sizes my be built by grouping, in a suitable way, several contiguous subblocks.

A special case of workload unbalance arises when many loop iterations write on specific and small regions of the reduction array (called high contention regions). A way of relieving this problem consists of the privatization (replication) of the blocks containing the contention region (*local expansion*). This way, write conflicts on that region disappear with memory requirements much lower than corresponding to the full privatization of reduction array.

3.3. Experimental Evaluation

We have experimentally evaluated the proposed solutions to improve the performance of DPO methods (specifically, DWA–LIP) and compared with other parallel irregular reduction methods on a SGI Origin2000 multiprocessor,

	Critical sections	Array Expansion / Replicated Buffers	Selective Privatization / Reduction Hash Tables	LOCAL–WRITE	DWA–LIP
Inspector Overhead	0	0	medium	low	low
Memory Requirements	0	high	medium (external)	low	low
Synchronization	high	0	0	low	low
Computation Replication	0	0	0	external	0
Parallelism loss	0	0	0	0	external
Inter–iteration Locality exploitation	no	no	no	yes	yes

Table 1. Typical performance properties for the LPO and DPO classes of parallel irregular reduction methods. The term `external` means that the property is not intrinsically exploited by the method, but it depends on input data

with 250-MHz R10000 processors (4 MB L2 cache) and 12 GB main memory, using IRIX 6.5. All parallel codes were implemented in Fortran 77 with OpenMP directives, and compiled using the SGI MIPSpro Fortran 77 compiler.

Different methods have been experimentally tested on the EULER code, from the motivating application suite of HPF-2 [8]. This code solves the differential Euler equation on an irregular grid, computing some physical magnitudes (such as velocities or forces) on the nodes described by a mesh. The code includes a single loop with two subscripted reductions on one array with three dimensions, which is placed inside an outer time-step loop. As an static problem, the inspector phase needs to be computed only once.

The parallel EULER kernel has been tested using an 800K nodes mesh with connectivities 8 and 18 (ratio between edges and nodes). Two versions of each mesh has been generated. One of them is obtained after applying a coloring algorithm to the edges, and placing edges of the same color consecutively in the indirection array. For this version we expect a low inter-loop locality in accessing the reduction array between different iterations. In the other version the list of edges has been sorted, and the inter–iteration locality is expected to be higher. This fact becomes obvious in serial executions, where the colored version is about 3 times slower than for the sorted version.

In Fig. 6 parallel reduction loop speed-up is shown for the tested techniques and data sets. We observe that in general DPO methods performs better than LPO. This behaviour is more significant for low inter–loop locality patterns (colored meshes). In addition, not only DPO methods is able to exploit data locality but also their extra memory requirements are lower (excluding the critical section technique, `C$OMP atomic`, that does not replicate arrays at all but its efficiency is very low).

Likewise, in previous works [12] we have tested optimizations of DPO methods with input data sets that exhibits low intra–loop locality, load unbalance or high contention regions. Although these are not common cases in irregular scientific codes, we have shown that our approaches (partial expansion, workload balance support and local expansion, respectively) allow to obtain a good trade off between memory overhead and degree of parallelism.

4. Pointer-based Programs

4.1. Motivation and related works

Programming languages such as C, C++, Fortran90, or Java are widely used for non-numerical (symbolic) and numerical applications. All these languages allow the use of complex data structures usually based on pointers and dynamic memory allocation. The use of complex data structures is very helpful in order to speedup code development and, besides this, it also may lead to reducing the program execution time. However, compilers are not able to successfully optimize codes based on these complex data structures for current computers or multicomputers. This is due to current compilers are not able to capture, from the code text, the necessary information to exploit locality, automatically parallelize the code, or carry out other important optimizations in pointer-based codes.

With this motivation, the goal of our research line is to propose and implement new techniques that can be included in compilers to allow for the automatic optimization of real codes based on dynamic data structures. As a first step, we have selected the shape analysis subproblem, which aims at estimating at compile time the shape the data will take at run time. Given this information, subsequent analysis (not implemented yet) would focus on particular optimizations, for example, to exploit the memory hierarchy or to detect whether or not certain sections of the code can be parallelized because they access independent data regions. Therefore, this work is part of the first step (program structure analysis) of our parallelization methodology.

There are other open research lines dealing with the analysis of codes in the presence of pointers, such as alias analysis or points-to analysis. Basically, these analysis are designed to determine the superset of locations to which a pointer *must* or *may* point (points-to sets) [7]. These kinds

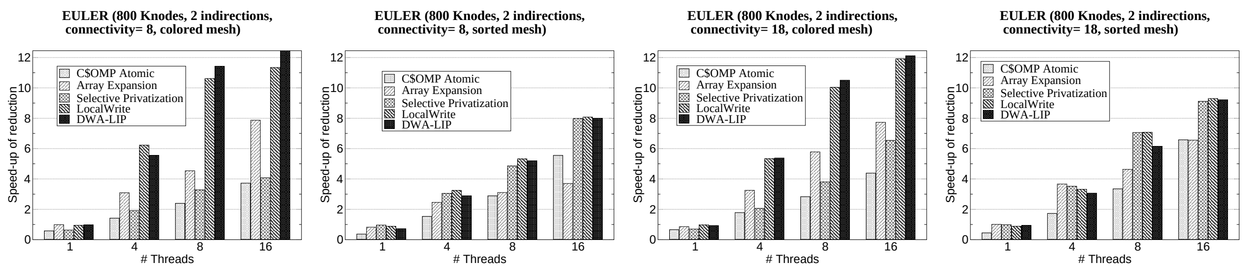

Figure 6. Speedup for the parallel reduction loop using LPO and DPO techniques

of pointer analysis provide enough information to allow for some scalar optimizations, such as Common Subexpression Elimination, Loop Invariant Removal, or Location Invariant Removal [9]. However, the information provided by the points-to sets is not accurate enough to enable more ambitious optimizations such as loop-level automatic parallelization, automatic data distribution, and locality exploiting. Currently, the majority of research groups rely on manual annotations when dealing with such complex code optimizations in the presence of pointers, due to points-to analysis is not sufficient. For instance, Chilimbi et al. ask the programmer to annotate the code to exploit cache locality [4] or a previous execution profile is needed in order to exploit cache prefetching [3]. In the area of distributed memory locality exploiting and communication optimization, Zhu and Hendren [24] also rely on code annotations with special compiler directives. Similarly, Rogers et al. [19] propose a thread-level parallelism in codes annotated with directives such as *futurecall* and *touch*.

However, some groups are trying to automatically extract more information from the code text to optimize codes based on pointers. For example, Ghiya [9] have implemented the McCAT compiler to put pointer analysis to work. Basically, this compiler uses points-to analysis to deal with stack-directed pointers and connection analysis and shape analysis to deal with heap-directed pointers. This analysis is used for exploiting two parallelism levels in codes based on recursive data structures which do not change their shape while they are traversed: at the function level when routines traverse disjoint sub-tree structures; and at the loop level in two cases; single liked list traversing and array of pointers to disjoint structures traversing. However, their shape analysis is too simple and conservative leading to a serious lack of parallelism exploitation. This is mainly due to it does not keep information about the topological structure of the links between heap locations.

Thus, we have to emphasize that our final goal is to allow for the automatic optimization of codes based on recursive data structures, but it is clear that, first of all, better shape analysis techniques have to be proposed. That is, new approaches to automatically capture the essential characteristics and properties of heap-allocated data structures are essential. With this in mind, our proposal is based on approximating all the possible memory configurations that can arise after the execution of a statement by a set of graphs: the *Reduced Set of Reference Shape Graphs* (RSRSG). With our framework we can achieve accurate results in a reasonable analysis time and expending a reasonable ammount of memory. Besides this, we cover situations that were previously unsolved, such as detection of complex structures (arrays of pointers, lists of trees, lists of lists, etc.) and structure permutation, as we will see in the next sections.

There are several ways the shape analysis problem can be approached. We have focus in the graph-based methods in which the "storage chunks" are represented by nodes, and edges are used to represent references between them. For example, Plevyak et al. [17] have proposed the the "Abstract Storage Graph" (ASG), while Sagiv et al. [21] improved the ASG method with what they call "Static Shape Graphs" (SSG). In a previous work [5] we saw that ASG or SSG were not sufficient to deal with the complex data structures we presented in the previous section. Basically, ASG and SSG approaches were too imprecise and too conservative in many simple cases, due to they associate just one graph with each statement in the code. Besides, too much information is fused in a single node and then it is impossible to capture the real properties of the data structures represented by the graphs. We have overcome this drawback by considering several graphs per statement, while fulfilling some rules to avoid an explosion in the number of graphs and nodes in each graph.

A more recent work that also allows several graphs per statement is the one presented by Sagiv et al. [22]. They propose a parametric framework based on a 3-valued logic. To describe the memory configuration they use 3-valued structures defined by several predicates. However, as far as we know the currently proposed predicates do not suffice to deal with the complex data structures that we handle in this paper. There are several differences between our shape analysis method and that of Sagiv et al. [22]. The main one is that we join similar graphs (RSGs) to build a reduced set of RSGs (RSRSG) for each program point, while in [22] they keep all the graphs (multiple structure approach) or just one (single structure approach). We think that this may explain why their Three-Valued-Logic Analyzer (TVLA) runs out of memory for simple codes such as the singly linked list bubble sort using the multiple structure approach [15]. Besides, they recognize that their TVLA engine is only useful to analyze small programs and report experimental results for small, singly linked list operations (insert, reverse, sort, etc.). Actually, they have not published experimental results successfully dealing with real codes based on the combination of complex data structures such as doubly linked lists pointing to trees or to other lists, etc. The next section briefly describes the shape analysis techniques we have developed in order to successfully capture such complex dynamic data structures.

4.2. Shape analysis

Basically, our method is based on approximating by graphs all possible memory configurations that can appear after the execution of a statement in the code. We call a collection of dynamic structures a *memory configuration*. These structures comprise several memory chunks, that we call *memory locations*, which are linked by references. Inside these memory locations there is room for data and for pointers to other memory locations. These pointers are called *selectors*.

Note that due to the control flow of the program, a statement could be reached by following several paths in the control flow. Each "control path" has an associated memory configuration which is modified by each statement in the path. Therefore, a single statement in the code modifies all the memory configurations associated with all the control paths reaching this statement. Each memory configuration is approximated by a graph we call *Reference Shape Graph* (RSG). So, taking all this into account, we conclude that each statement in the code will have a set of RSGs associated with it.

4.2.1 Reference Shape Graphs

The RSGs are graphs in which nodes represent memory locations which have similar reference patterns. To determine whether or not two memory locations should be represented by a single node, each one is annotated with a set of properties. Now, if several memory locations share the same properties, then all of them will be represented by the same node. This way, a possibly unlimited memory configuration can be represented by a limited size RSG, because the number of different nodes is limited by the number of properties of each node. These properties are related to the "reference pattern" used to access the memory locations represented by the node. Hence the name *Reference Shape Graph*.

As we have said, all possible memory configurations which may arise after the execution of a statement are approximated by a set of RSGs. We call this set *Reduced Set of Reference Shape Graphs* (RSRSG), since not all the different RSGs arising in each statement will be kept. On the contrary, several RSGs related to different memory configurations will be fused when they represent memory locations with similar reference patterns. There are also several properties related to the RSGs, and two RSGs should share these properties to be joined. Therefore, besides the number of nodes in an RSG, the number of different RSGs associated with a statement are limited too. This union of RSGs greatly reduces the number of RSGs and leads to a practicable analysis.

4.2.2 Generating the RSRSGs: the symbolic execution

To move from the "memory domain" to the "graph domain", the calculation of the RSRSGs associated with a statement is carried out by the **symbolic execution** of the program over the graphs. In this way, each program statement transforms the graphs to reflect the changes in memory configurations derived from statement execution. The **abstract semantic** of each statement states how the analysis of this statement must transform the graphs.

Let us illustrate all this with an example. In Figure 7 we can see a simple code with seven pointer statements. Our analyzer symbolically executes each statement to build the RSRSG associated with them. Actually, after the execution of the third statement we obtain an RSRSG with a single RSG which represents three different memory locations by three nodes; all of them of the same type, with the same *nxt* selector, but pointed to by different pointer variables (*pvars)*. Now, this RSRSG is modified in three different ways because there are three different paths in the control flow graph, each one with a different pointer statement. All these paths join in statement 7, and after the execution of this statement we obtain an RSRSG with two RSGs. This is because the RSGs coming from statements 5 and 6 are compatible and can be summarized into a single one.

The whole symbolic execution process can be seen by looking at Fig. 8. For each statement in the code we have an input $RSRSG_i$ and the corresponding output $RSRSG_o$ representing the memory configurations after statement execution. During the symbolic execution of the statement all the rsg_{ij} belonging to $RSRSG_i$ are going to be updated. The first step comprises graph division to better focus on the several memory configurations represented by the RSG. Pruning removes redundant or nonexistent nodes or links that may appear after the division operation. Then the abstract interpretation of the statement takes place and usually the complexity of the RSGs grows. In order to counter this effect, the analysis carries out a compression operation. In this phase each RSG is simplified by the summarization of compatible nodes, to obtain the rsg^*_{ijk} graphs. Furthermore, some of the rsg^*_{ijk} can be fused into a single rsg_{ok}

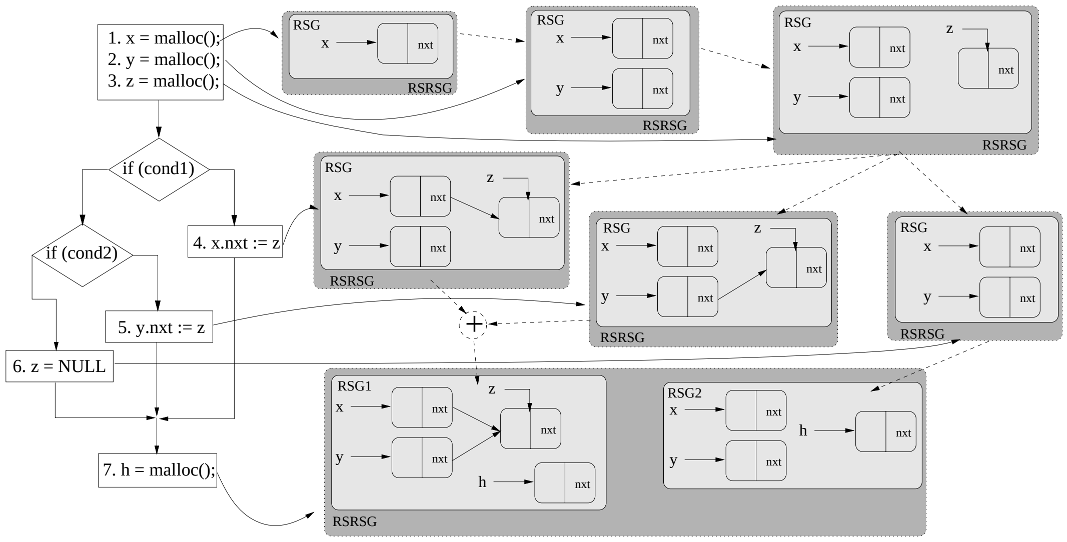

Figure 7. Building an RSRSG for each statement of an example code

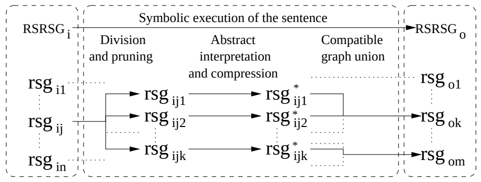

Figure 8. Schematic description of the symbolic execution of a statement.

```
typedef struct str {
    ...
    struct str1 *sel;
    struct str2 *sel1[256];
    struct str **sel2; }
ss=sizeof(struct str);
sp=sizeof(struct str *);
x=(str *)malloc(ss);
x->sel2=(str **)malloc(n*sp);
```

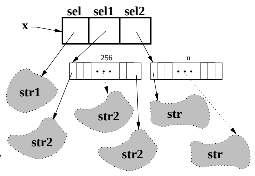

Figure 9. Example of data structure containing arrays of pointers

if they represent similar memory configurations. This operation greatly reduces the number of RSGs in the resulting RSRSG.

The abstract interpretation is carried out iteratively for each statement until we reach a fixed point in which the resulting set of rsg_{oj} associated with the statement does not change any more. This way, for each statement that modifies dynamic structures, we have to define the abstract semantics which describe how these statements modify the rsg_{ij}. We consider six simple instructions that deal with pointers: $x = NULL$, $x = malloc$, $x = y$, $x \rightarrow sel = NULL$, $x \rightarrow sel = y$, and $x = y \rightarrow sel$. More complex pointer instructions can be built upon these simple ones and temporal variables. Due to space constraints we cannot formally describe the abstract semantics of each one of these statements.

4.2.3 Dealing with arrays of pointers: multiselectors

We can view an array of pointers as a set of n selectors (links), all with the same name. Our original method, briefly described before, only deals with single selectors (which represent single links). Thus, the problem arising with the arrays of pointers is that a single selector name represents several links, and all of them belong to the same memory location (due to having been allocated by the same *malloc* instruction).

We illustrate all this with the following example. Figure 9 shows an example of a complex data structure definition comprising two arrays of pointers, and it also illustrates the corresponding memory configuration after the execution of the last "malloc()" statement. As we note, sel is a single selector which can point to a single memory location and which can be modified by statements like "x→sel=...". These kinds of selectors can be managed by our previous analyzer. However, $sel1$ and $sel2$ represent arrays of selectors. The difference between $sel1$ and $sel2$ is that we know the size of the $sel1$ array at compile time, but the size of $sel2$ is defined at run time. In any case, we now want to deal with both types of arrays of selectors, which now have to be modified by statements like "x→sel1[i]=..." or "x→sel2[i]=...".

Since $sel1$ and $sel2$ are not single selectors, we have called them *multiselectors*. In order to take into account multiselectors in our method we have introduced in our analyzer the following procedure: since our method is already able to deal with single selectors our goal is now to include a previous step in the symbolic execution process to focus on one of the selectors included in a particular multiselector. In other words, a statement like "x→sel1[i]=..." is going to update a single selector (a particular selector included in the multiselector $sel1$), but before applying the symbolic execution, our analizer start by identifying the particular sel1[i] which is going to be updated, to subsequently proceed with the abstract interpretation.

4.3. Experimental results

Our RSRSG analyzer has been written in C and can be fed with an input code to generate the RSRSG associated with each statement of the code. The codes have to be preprocessed in a first step to just keep the statements dealing with pointers. We have implemented the analyzer to carry out a progressive analysis which starts with fewer constraints to summarize nodes, but, when necessary, these constraints are increased to reach a better approximation of the data structure used in the code. More precisely, the analysis comprises three levels: L_1, L_2, and L_3, from less to more complexity as we explain in [6].

With this tool we have analyzed several codes: an artificial code ("working example"), the sparse Matrix by vector multiplication, the sparse Matrix by Matrix multiplication, the Sparse LU factorization, and the Barnes-Hut code. These five codes have two implementations, one in which arrays of pointers are implemented by doubly linked lists and the other in which arrays of pointers are keep.

The first four codes were successfully analyzed in the first level of the analyzer, L_1. However, for the Barnes-Hut program the highest accuracy of the RSRSGs was obtained in the last level, L_3. All these codes where processed by our analyzer in a Pentium 4 1.6 GHz with 128 MB main memory. The time and memory required by the analyzer are summarized in Table 2. In this table we also show the number of code lines after the preprocessing of the original C codes. The particular aspects of these codes are described next.

1. **Working example's RSRSG.** This code generates, traverses, and modifies the data structure presented in Figure 10 (a). A compact representation of the resulting RSRSG for the last statement of the code can be seen in Figure 10 (b). The data structure is a doubly linked list of pointers to trees (header list). Besides this, the leaves of the trees have pointers to doubly linked lists. All the trees pointed to by the header list are independent and do not share any element. In the same way, the lists pointed to by the leaves of the same tree or different trees are also independent.

   This data structure is built by a C code that also traverses the elements of the header list with two pointers and eventually can permute two trees. From the properties associated with the nodes in the RSRSG represented in Figure 10 (b) we can infer the actual properties of the real data structure: the trees and lists do not share elements and therefore they can be traversed in parallel. More precisely: (i) The analyzer successfully detects the doubly linked list which is acyclic by selectors nxt or prv and whose elements point to binary trees; (ii) Two different items of the header list cannot point to the same tree; (iii) Different trees do not share items; (iv) The same happens for the doubly linked list pointed to by the tree leaves: all the lists are independent, there are no two leaves pointing to the same list, and these lists are acyclic by selectors nxt or prv.

   The other implementation of this code is based on an array of pointers to the trees instead of the header list. Again, the analyzer can extract the same conclusions commented in the previous paragraph.

2. **Sparse matrix codes.** Here we deal with some irregular codes which implements sparse matrix operations: the sparse matrix by vector multiplication, $r = M \times v$; the sparse matrix-matrix multiplication, $A = B \times C$; and the sparse LU factorization, $A = LU$.

   The sparse matrices are stored in memory as a header doubly linked list (or an array of pointers) with pointers to other doubly linked lists which store the matrix rows (if the matrix is row-wise) or columns (for column-wise matrices). In figure 11 (a) we show the sparse matrix data structure for a row-wise matrix where the matrix header is implemented by an array of pointers. The sparse vectors, v and r are doubly linked lists. After the analysis process, carried out by our analyzer, the resulting RSRSG accurately represents the data structures. In the resulting RSRSG for the last statement of these codes we can identify the main properties of the data structures: (i) The rows of the matrix are pointed to from different elements of the header list/array; (ii) The doubly linked lists which store the rows of the matrices and the vectors are acyclic by selectors nxt and prv. A subsequent analysis of the code and the RSRSG associated with each statement would be able to state that several sparse matrix row can be traversed and updated in parallel and, in addition, it is also possible to update each row in parallel.

3. **Barnes-Hut N-body simulation.** The structure used in this code is basically an octree where each leaf

	Working Ex.	S.Mat-Vec	S.Mat-Mat	S.LU	Barnes-Hut
Level	L_1 / L_2 / L_3	L_1 / L_2 / L_3	L_1 / L_2 / L_3	L_1 / L_2 / L_3	L_1 / L_2 / L_3
	Codes without arrays of pointers				
Time	0’03”/0’05”/0’06”	0’01”/0’02”/0’03”	0’20”/0’38”/1’00”	7’50”/-/-	5’56”/0’34”/2’06”
MBytes	2.11/2.78/3.02	1.37/1.85/2.17	8.13/11.45/12.68	99.46/-/-	37.82/8.82/8.94
Lines	213	104	156	164	216
	Codes including arrays of pointers				
Time	0’05”/0’07”/0’08”	0’01”/0’01”/0’01”	0’04”/0’06”/0’06”	1’08”/1’12”/-	23’08/25’27”/0’21”
MBytes	1.77/2.29/2.50	0.92/1.03/1.2	1.19/1.31/1.49	3.96/4.18/-	40.14/42.86/3.06
Lines	144	87	103	143	177

Table 2. Time and space required to process several codes with different number of code lines

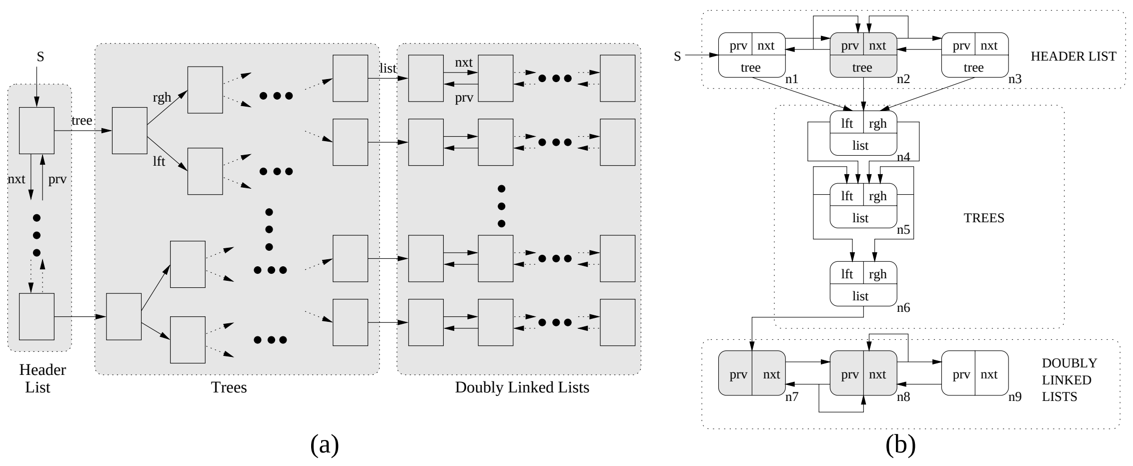

Figure 10. A complex data structure (a), and compact representation of the resulting RSRSG (b)

points to an element of a single linked list. In the implementation which avoids pointer arrays, each octree node which is not a leaf has a pointer $child$ pointing to the first of its eight children which are linked by selector $next$. If pointer arrays are allowed, the pointers to the eight children are stored in an array of pointers, as we can see in figure 11 (b). The analysis of this code enable the parallel traversal of the octree which is precisely captured by the obtained RSRSG’s.

5. Conclusions

This paper addresses the problem of compiler optimization and/or parallelization of irregular and pointer-based (dynamic) applications. From our work on this problem we may derive two main conclusions. First, a complete and powerful data analysis is fundamental. This analysis must include, at least, two important tasks: Analysis of the data organization, and analysis of the memory references. In irregular codes, data organization analysis is not difficult as typically data is arranged as arrays. However, memory references are dynamic and data dependant. In dynamic codes, however, both analysis are very complex. In this line, we have developed shape analysis techniques to capture properties for complex pointer-based data structures.

The second conclusion is that we consider a promising way to obtain an effective parallelization to design ad-hoc techniques for specific complex computational structures. For instance, we discussed efficient solutions for irregular reductions, that optimize some specific performance issues. In a similar way, once the data organization of a pointer-based code has been identified, it is possible to develop efficient automatic techniques to traverse and update these data structures (trees, linked-lists, ...).

References

[1] W. Blume, R. Doallo, R. Eigenmann, *et al.* Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, 1996.

[2] W. Blume and R. Eigenmann. The range test: A dependence test for symbolic, non-linear expressions. *ACM Int’l Conf. on Supercomputing (ICS’94)*, pp. 528–537, 1994.

[3] T.M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. *ACM SIGPLAN Conf. on Programming Languages Design and Implementation (PLDI’01)*, pp. 191–202, 2001.

[4] T.M. Chilimbi, M.D. Hill and J.R. Larus. Cache-conscious structure layout. *ACM SIGPLAN Conf. on Programming*

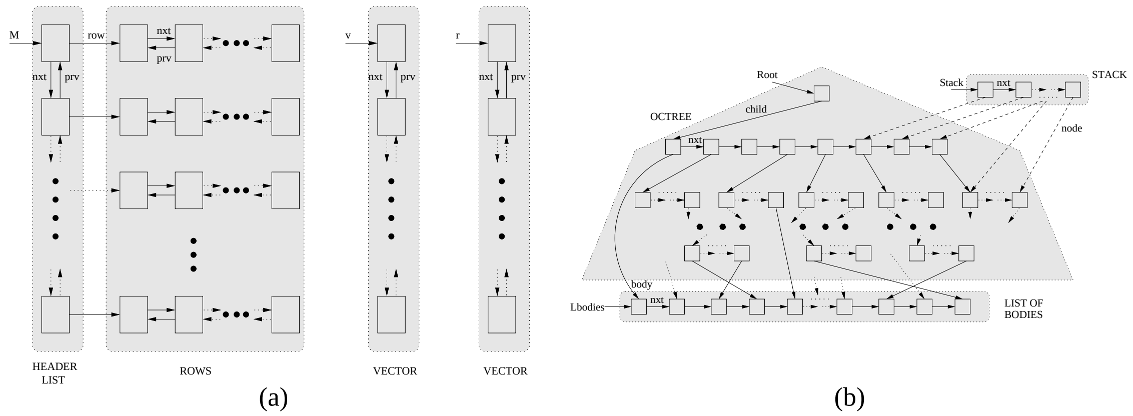

Figure 11. Data structure for sparse matrices and vectors (a); Barnes Hut main data structure (b)

Languages Design and Implementation (PLDI'99), pp. 1–12, 1999.

[5] F. Corbera, R. Asenjo, and E.L. Zapata. New shape analysis for automatic parallelization of C codes. *ACM Int'l Conf. on Supercomputing (ICS'99)*, pp. 220–227, 1999.

[6] F. Corbera, R. Asenjo, and E.L. Zapata. Accurate shape analysis for recursive data structures. *Int'l. Workshop on Languages and Compilers for Parallel Computing (LCPC'2000)*, pp. 1–15, 2000.

[7] M. Das. Unification-based pointer analysis with directional assignments. *ACM SIGPLAN Notices*, 35(5):35–46, 2000.

[8] I. Foster, R. Schreiber and P. Havlak, *HPF-2, Scope of Activities and Motivating Applications*, *Technical Report CRPC-TR94492*, Rice University, November 1994.

[9] R. Ghiya. Putting Pointer Analysis to Work. *PhD thesis*, School of Comp. Sci., McGill Univ., Montreal, 1998.

[10] A. Gibbons Algorithmic Graph Theory. Cambridge University Press, 1999.

[11] E. Gutiérrez, O. Plata and E.L. Zapata. A compiler method for the parallel execution of irregular reductions in scalable shared memory multiprocessors. *ACM Int'l. Conf. on Supercomputing (ICS'2000)*, pp. 78–87, 2000.

[12] E. Gutiérrez, O. Plata and E.L. Zapata. Optimization techniques for parallel irregular reductions. *Journal of Systems Architecture*, 49(3):63–67, 2003.

[13] H. Han and C-W. Tseng. Efficient compiler and run-time support for parallel irregular reductions. *Parallel Computing*, 2000. 26(13–14):1861–1887.

[14] H. Han and C.-W. Tseng, *A Comparison of Parallelization Techniques for Irregular Reductions, 15th IEEE Int'l. Parallel and Distributed Processing Symp. (IPDPS'2001)*, San Francisco, CA, April 2001.

[15] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. *Static Analysis Symp.*, pp. 280–301, 2000.

[16] Y. Lin and D. Padua. On the automatic parallelization of sparse and irregular fortran programs. *4th Workshop on Languages, Compilers and Runtime Systems for Scalable Computers (LCR'98)*, 1998.

[17] J. Plevyak, A. Chien and V. Karamcheti. Analysis of dynamic structures for efficient parallel execution. *Int'l. Workshop on Languages and Compilers for Parallel Computing (LCPC'93)*, pp. 37–57, 1993.

[18] W.M. Pottenger and R. Eigenmann. Idiom recognition in the Polaris parallelizing compiler. *ACM Int'l Conf. on Supercomputing (ICS'95)*, pp. 444–448, 1995.

[19] A. Rogers, M.C. Carlisle, J.H. Reppy and L.J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 17(2):233–263, 1995.

[20] L. Rauchwerger and D. Padua. The privatizing DOALL test: A run-time technique for DOALL loop identification and array provatization. *ACM Int'l Conf. on Supercomputing (ICS'94)*, pp. 33–43, 1994.

[21] M. Sagiv, T. Reps and R. Wilhelm. Solving shape-analysis problems in laguages with destructive updating. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 20(1):1–50, 1998.

[22] M. Sagiv, T. Reps and R. Wilhelm. Parametric shape analysis via 3-valued logic. *Symp. on Principles of Programming Languages*, pp. 105–118, 1999.

[23] H. Yu and L. Rauchwerger. Adaptive Reduction Parallelization Techniques. In *Proceedings of the 14th ACM International Conference on Supercomputing (ICS'2000)*, pages 66–77, Santa Fe, NM, May 2000.

[24] Y. Zhu and L. Hendren. Locality analysis for parallel C programs. *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, 10(2):99–114, 1999.