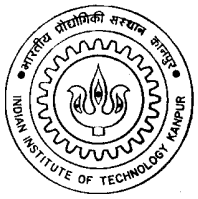
A Framework for Automatic Data Layout for Distributed Memory Machines

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Master of Technology

by

Raghavendra Maloo



to the

Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

May, 2001

Certificate

This is to certify that the work contained in the thesis entitled "*A Framework for Automatic Data Layout for Distributed Memory Machines*", by *Raghavendra Maloo*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

May, 2001

(Dr. Sanjeev Kumar Aggarwal)
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

Acknowledgements

As per the indian tradition the almighty God is always the first to be acknowledged, because without proper intuition, induced by the God, mind is either null or garbage box.

I am grateful to Dr. Sanjeev Kumar Aggrawal, my thesis supervisior, for providing me this opportunity to work with him, and helping me always. It is a great pleasure to have Dr. Aggrawal as adviser. His guidance is crucial for the development of my initial ideas. His time to time reviews of my design infused confidence in me. Documentation of this thesis work would have been a formidable task for me without him.

I would like to thank Dr. Keshav Pingali, Associate Professor, Cornell University - USA. His involvement in my thesis work is a great coincidence, and his suggestions inspired me to explore new directions in my work. His ideas made a profound influence on my research interests.

I have special regards for Dr. R. Moona and Dr. P. Gupta. Publishing a paper with Dr. P. Gupta highly encouraged and motivated me for research work. Frequent discussions with Dr. R. Moona enriched my knowledge of computer architecture. I have been always surprised by his knowledge in almost every field. My whole-hearted thanks to the faculty members of Computer Science and Engineering Department for imparting me with invaluable knowledge and adding a lot of value-oriented growth to my career. Many thanks to the technical staff for providing an excellent working environment.

Without acknowledging my friends over here, this acknowledgement would only be a

nerd's voice. My stay at IITK without friends like Mayank, Avinash, Ashutosh, Raju, Gaurav, Alok and Mukul would have been an imprisonment. Each one of them are special for me. Mayank was always ready to help me and share my feelings. Avinash was a constant source of inspiration and helped me a lot to get me a touch of modern and intellectual society. Ashutosh and I had lots of habits in common and that strenghthen our friendship. One never gets bored of his unending chatter. I can never forget the smiling face of Gaurav. On the other hand Kingshuk, Prashanta, Sharath, Reddy were helping me without any hesitation, without their help this thesis would have been only a miracle. Sriram and Vikrant also remained quite close to me. I always preferred Sriram as my project or term paper partner. Vikrant was an unending source of help and entertainment. I would like to thank our phd brothers specially Atul and Rajiv. They had been an unfailing source of encouragement and support. My heartfelt thanks to all my mtech99 friends for making such cordial environment, the most suitable one for proper working. I can never forget the mail sequels, parties and refreshments I had with them.

Last but not least, I would like to pay my best gratitudes to my family. Their constant love and affection throughout my life made me to reach at this stage.

Finally thanks to all whom I forgot to acknowledge or unable to acknowledge.

(Raghavendra Maloo)

Abstract

Distributed memory machines have gained wide acceptance and different types of programming models are provided for such machines. Data parallel programming model has become popular for such machines. HPF and Fortran D support this programming model and require the user to specify the data layout in the program. Data layout choice is of key importance while programming in such languages. A bad data layout may degrade the performance of the system. A good data layout choice depends on various system parameters and hence makes such choice very difficult for a general user of these languages. To automate the process of selection of data layout, various techniques have been proposed. In this thesis we present issues related to remapping and distribution analysis of automatic data layout technique that were partially explored previously. We present a simple algorithm to determine points in the program where dynamic realignment and redistribution is beneficial. Our distribution analysis follows constructive approach, and is strong enough to find out BLOCK, CYCLIC and BLOCK_CYCLIC distributions. We refine the heuristic algorithm proposed by Li and Chen [21, 22] for alignment analysis and use it in our approach. A prototype tool based on our technique has been implemented and experimented.

Contents

List of Figures

Chapter 1

Introduction

Distributed memory architectures have gained wide acceptance due to some of its nice features like it can be built using off-the-shelf components and its scalablity in terms of number of processors. Different types of programming models are provided for such machines. In the data parallel programming model [19], data is distributed among available processors using compiler directives, and all the processors perform required computation on their assigned data (owner compute rule). HPF and Fortran D support this programming model [27, 13, 14]. Data parallel programming model has become popular because other programming models for such machines (explicit message passing model etc.), are too tedious for common users.

Languages like HPF and Fortran D require the user to specify the data layout. Data layout choice is of key importance while programming in such languages, because the compiler generates code and performs optimizations based on specified data layout. A bad data layout choice may incur overwhelming communication overheads and hence may degrade the performance of the system. A good data layout choice depends on various system parameters, including target machine configuration, target compilation system, number of available processors, program characteristics etc. For a user unfamiliar with these system parameters, selection of good data layout is a formidable task. HPF and Fortran D also provide directives for remapping of data at appropiate points in program. For a general user it is very difficult to find those points where remapping is beneficial.

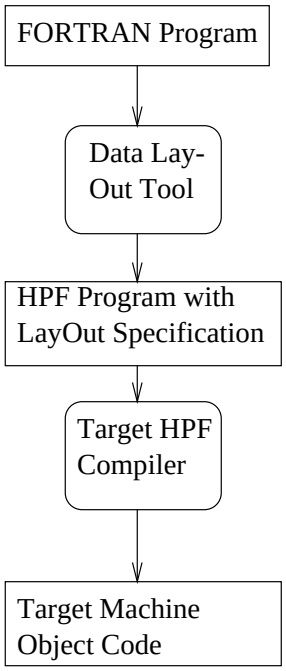


Figure 1.1: Application scenario

To automate the process of selection of data layout, various techniques have been proposed [20]. The goal of automatic data layout technique is to generate efficient data layout for sequential programs which do not have data layout specification. Application scenario of these tools is shown in Fig.1.1. Finding efficient data layout automatically may not be possible in all the cases and may require user interaction. Target inputs to automatic data layout techniques can be classified into regular programs and irregular programs based on computation structure of the program. Regular programs have a systematic computation structure and hence are amenable for automatic selection of data layout, with minimal of user interaction. Therefore regular programs have remained in focus as target inputs to these techniques.

Automatic data layout tools have two major phases: alignment and distribution. Alignment phase tries to find out appropiate alignment of all the arrays with respect to each other by mapping each array element to array elements of other arrays. A good alignment is based on alignment preferences of program computation and minimizes data movement i.e. interprocessor communication. Alignment preferences are extracted from

individual instruction of the program, and hence, depend on fine-grain parallelism available in the program. The distribution phase decides which dimension or dimensions of arrays are distributed, and number of processors allocated to each of them. A good distribution exploits coarse-grain parllelism available in the code.
In addition to these two phases, some tools also consider remapping between various program parts. These tools first partition program into program segments such that remapping between these program part may be beneficial. Intuitively remapping may be beneficial between different nested loops and hence are appropiate candiates for program parts. Benefit of remapping is calculated considering remapping cost, individual program part execution cost and merged program parts execution cost. Based on this value remapping is inserted or removed.

1.1 Related Work

The problem of automatic data layout have been discussed extensively in the past [1, 2, 5, 7, 9, 12, 18, 24, 29]. In this section we restrict ourself to compile-time automatic data layout selection approaches that use instances of program statements as basic units of computation. The proposed techniques for automatic data layout vary substantially in the assumptions made about the input languages, the possible set of data layout directives, the target compilation system, and the target machine configuration.

- Kremer proposes a framework for automatic data layout tool [18] for regular programs as a part of D System, a data parallel programming environment project [4]. His tool works in four steps: Initial step partition program into code segments, called phases. Data remapping is allowed only between phases. In the next step, data layout search space is constructed for each phase. A data layout for a phase is specified by the alignment and distribution of all the arrays referenced in that phase. First alignment analysis builds a search space of reasonable alignment schemes for each phase. Then distribution analysis uses the alignment search space to build candidate data layout search space of promising alignment

and distribution for each phase. In the next step cost of each candidate layout and remapping is estimated using compiler, execution and machine models. In the final step, data layout with appropiate remapping is selected for the program such that overall cost is minimal. Kremer uses exhaustive approach in his distribution analysis, it is computationally too expensive and only considers BLOCK and CYCLIC distribution.

- Gupta and Banerjee [12] propose a technique for automatic data layout as part of Parafrase-2 System [10, 11, 12]. The technique performs alignment and distribution analysis based on constraints for each statement in the program. Constraint represent data layout preferences and associated with a weight, a quality measure. Depending on selected overall data layout, constraints are either satisfied or not. The associated weight is a penalty function representing cost for not satisfying that constraint. Alignment preferences are mapped as constraints. Quality measures of constraints are parameterized function of some parameters including distribution scheme, and hence, represent distribution preferences. Alignment analysis is performed based on Li's and Chen's approach [21, 22]. The best distribution scheme for each dimension is determined based on communication cost of each statement. Dynamic realignment and redistribution are not supported.

- Palermo and Banerjee [24] extended work by Gupta and Banerjee to handle dynamic remapping. This appraoch is based on hierarchical decomposition of the program. In every decomposition step, a single component is split into two subcomponent if remapping between these subcomponents is beneficial.

- Garcia, Aygudae and Labarta [3, 9] propose an approach to handle alignment and distribution at the same time. Since alignement and distribution decisions may be mutually dependent, performing alignment and distribution analysis independent of each other may lead to inferior results. Both, alignment and distribution information is represented togther in a directed weighted graph, called communication-parallelism graph(CPG). Problem of automatic data layout is redefined as minimal path problem with some additional constraint on this graph.

- Anderson and Lam [2] discussed an algorithm for automatic data and computation

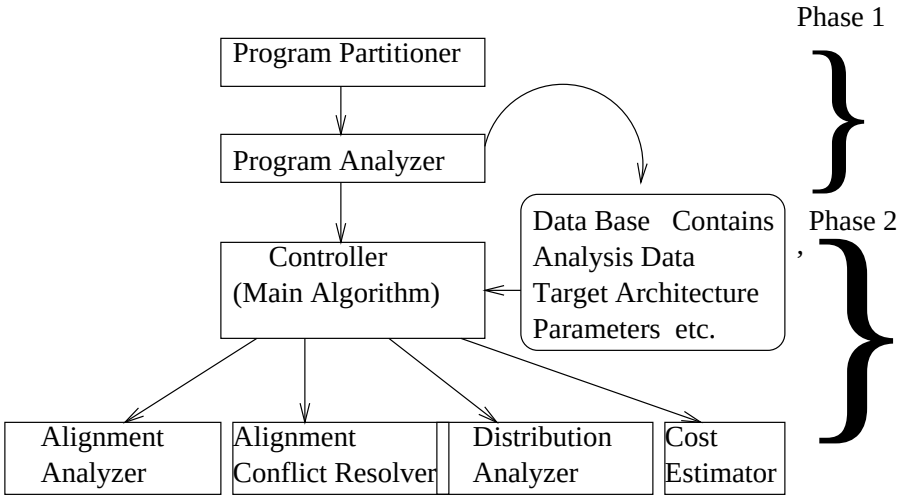


Figure 1.2: Components of Automatic Data Layout Tool

mapping for matrix computations. The focus of their work is on data and computation alignment. The central idea in their approach is mathematical formulation of data and computation mapping problem. This approach also supports dynamic data remapping.

Other techniques for automatic data layout are proposed by Li, Chen and Choo [7], Wholey [28, 29], Albert, Knobe, Lukas and Steele [1], Chatterjee, Gilbert, Schreiber, Teng [5] etc..

1.2 Our Approach

In our approach program is partitioned into program segments. Later some of these program segments are merged if remapping is not beneficial between them. The data layout for each program segment is determined while calculating the benefit of merging for these program segments. The tool based on our technique consists of two phases, as shown in Fig.1.2. The first phase partitions program and generates data access patterns for each program part. Data access pattern is set of array elements accessed in one iteration of a instruction. In the second phase, the controller produces the optimal data layout for the program with appropriate remapping. Controller first determines optimal data layout for each program part and later merges some program parts if remapping

is not beneficial between them. The optimal data layout of a program part consist of appropiate alignment and distribution for arrays of that program part. Controller performs its task iteratively. In each iteration of controller, data layout of the program is made more efficient. Initially data layout for each program segment is determined separately. In subsequent iterations consecutive program segments are merged into one if remapping between them is not beneficial and their combined optimal data layout is determined.

Unique aspects of our framework are as follows:

- Algorithm for controller is quite simple and efficient
- Distribution analysis phase follows efficient constructive approach instead of previously used expensive exhaustive approaches [18]
- Distribution analysis is powerful enough to find out BLOCK, CYCLIC and BLOCK_CYCLIC layouts, whereas previous approaches were restricted to BLOCK and CYCLIC only.
- Alignment analysis in our framework uses technique proposed by Li and Chen [21, 22]. We refine the conflict resolution algorithm by Li and Chen [21, 22] for more accurate results.

Our technique is flexible enough and to port automatic data layout tool from one machine to another only some machine parameter files need to be replaced. Many of previously proposed techniques can also be partially implemented in our framework. In almost all these previous approaches efficient technique for remapping and distribution analysis were not discussed in detail. In our technique, we discuss these issues in somewhat more detail. Our framework uses efficient graph algorithms [8], and hence computationally less expensive than other proposed algorithms using expensive techniques, such as 0-1 integer programming.

1.3 Components of Automatic Data Layout tool

In this section we describe components of our data layout tool briefly and in the following chapters we discuss each of these components in detail, Fig. 1.2 shows various

components of our tool. In first phase program partitioner partitions program into program segments such that data remapping between these segments may be beneficial. Partitioned program is assigned to program analyzer. Program analyzer generates data access pattern of each program segment into its corresponding file. Database is a repository of data access patterns and contains all the files generated by program analyzer; it also contains target machine parameters such as communication cost, remapping cost, instruction execution cost. In second phase controller produces optimal data layout with appropriate remapping for the program. Controller uses alignment analyzer, alignment conflict resolver, distribution analyzer and cost estimator. Alignment analyzer identifies alignment preferences of program segment and represents these preferences in the form of a graph called component affinity graph (CAG). Alignment conflict resolver produces optimal conflict-free alignment from a given CAG. Distribution analyzer generates optimal distribution of a program segment for a given alignment of that segment. Distribution analyzer uses data access pattern of that program segment. Cost estimator estimates the execution cost of a program segment with alignment and distribution specified.

1.4 Organization of the report

A short introduction to High Performance Fortran can be found in Chapter 2. Our framework for automatic data layout consists of various components. Each of these components is discussed in detail in Chapters 3, 4 or 5. Our technique consists of two phases. In first phase program partitioner (Section 3.1) partitions program into program segments. Partitioned program is assigned to program analyzer (Section 3.2). Program analyzer generates data access pattern of each program segment into its corresponding file into database. In second phase controller(Chapter 5) produces optimal data layout with appropriate remapping for the program. Controller uses alignment analyzer(Section 3.3), alignment conflict resolver(Section 3.4), distribution analyzer(Chapter 4) and cost estimator.

A prototype data layout tool has been implemented, Chapter 6 describes the prototype

tool implementation details. Chapter 7 shows that the experimental results with prototype tool is encouraging and tool generates data layouts of desired quality. A summary of this work and a discussion of future work can be found in Chapter 8.

Chapter 2

Background

Languages such as High Performance Fortran (HPF) and Fortran D [27, 13, 14] provide a machine-independent programming model and support data parallelism. Section 2.1 explains those data layout directives of HPF which are relevant to our work on automatic data layout.

2.1 HPF Data Layout Directives

The task of distributing data across processors can be approached by considering the two aspects. First, there is the question of how arrays should be aligned with respect to one another, both within and across array dimensions. These are called intra-dimensional alignment and inter-dimensional alignement respectively. Alignment is determined by the the structure of the computation patterns in the program. Alignment is identified by seeing every instruction separately hence depend on fine-grain parallelism. It represents the minimal requirements for reducing data movement for the program, and is independent of any machine considerations. Second aspect is how arrays should be distributed onto the actual parallel machine. It is affected by the topology, communication mechanisms, size of local memory, and number of processors of the underlying machine.

HPF provides data layout specifications for these two levels of parallelism using TEMPLATE, ALIGN, DISTRIBUTE, and PROCESSORS statements. HPF also supports dynamic remapping, i.e., the data layout of arrays can change during the execution of the program. REALIGN and REDISTRIBUTE statements are used to specify remappings. Realignment and redistribution statements are executable statements and not declarative.

- TEMPLATE

  Template is simply a virtual array or index domain and no storage is allocated for it. Template enables user to group data arrays. In our work arrays are aligned to a common template so they aligned to each other automatically. Also this single template is distributed and all associated arrays are distributed automatically. TEMPLATE statement declares name, dimensionality and size of template. As following statement:

  TEMPLATE A(10, 10)

  declares a 2-dimensional 10×10 template with name A.

- ALIGN

  Align statement is used to align arrays with respect to each others based on alignment preferences. All the arrays are mapped to common template so that they aligned to each other automatically. Alignment of array to template is specified by the order of subscript variables in both array and template.

  REAL X(N, N)

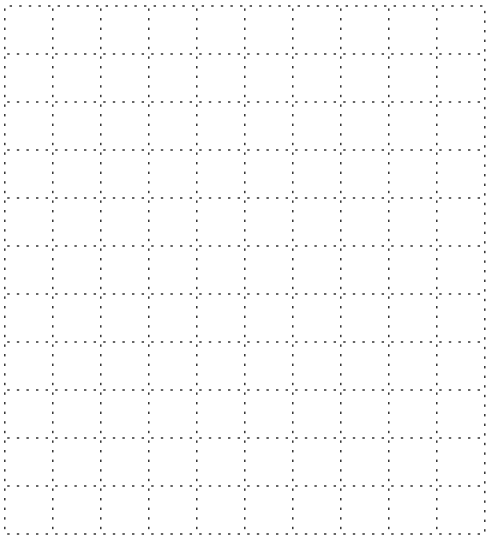
  TEMPLATE A(N, N)

  ALIGN X(I, J) with A(J, I)

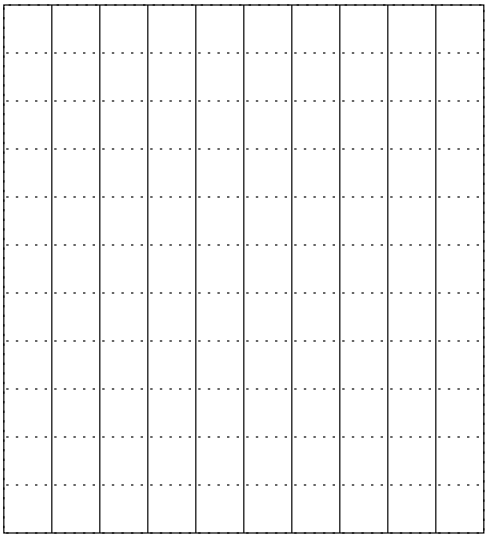
  In this example transpose of X is mapped to template A.

- DISTRIBUTE

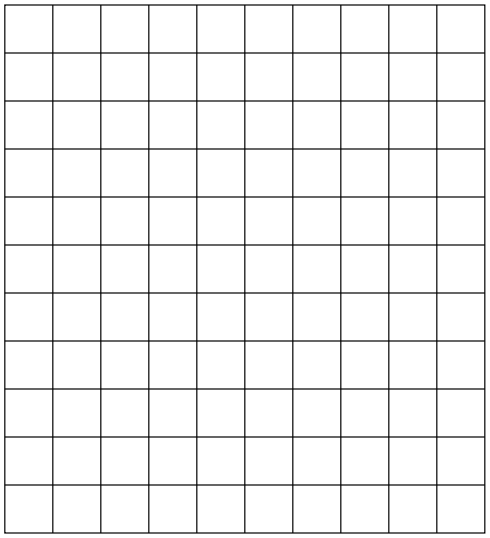
  Distribute statement is used to distribute group of arrays onto available processors. User specifies distribution to common template and compiler applies distribution to all the mapped arrays. Distribution statement takes name of the template and assign distribution type parameter for each dimension of the template. Each distribution type parameter contributes in mapping of data element to a processor.



TEMPLATE A(10, 10)



ALIGN X(I, J) with A(J, I)



REAL X(10, 10)

Figure 2.1: Align Example

Distribution type parameters are independent in each dimension.

DISTRIBUTE A(type, type)

Distribution type parameters:

The different type of parameters are: BLOCK, CYCLIC, BLOCK_CYCLIC. Suppose there are P processors and N elements in a template dimension, as in following example.

```
TEMPLATE A(N), B(N), C(N)
PROCESSORS PROCS(P)
DISTRIBUTE A(BLOCK) ONTO PROCS
DISTRIBUTE B(CYCLIC) ONTO PROCS
DISTRIBUTE C(BLOCK_CYCLIC(4)) ONTO PROCS
```

1. BLOCK: divides the template into contiguous segments of size N/P and assigns one segment to each processor.
2. CYCLIC: distribute template elements in round-robin fashion i.e. assign each ith element to i mod P processor.
3. BLOCK_CYCLIC(M): takes a parameter M. It first divides the dimension into contiguous segments of size M and then distribute each segment in round-robin fashion to processors, as in CYCLIC.

- REALIGN Same as ALIGN directive and can be used to realign a array with alignment differnt from previous alignment.

- REDISTRIBUTE Same as DISTRIBUTE directive and can be used to redeistribute a template with distribution differnt from previous distribution.

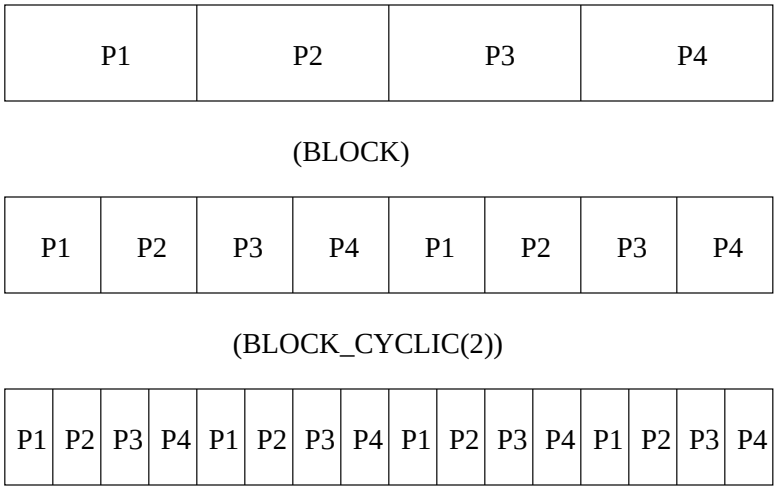


Figure 2.2: Distribution Example

Chapter 3

Program Analysis and Alignment Analysis

In this chapter we describe some components of our data layout tool. Some of these components uses existing techniques with no refinement and others uses existing techniques with slight refinement.

3.1 Program Partitioner

Function of program partitioner is to identify probable points of data remapping in the program. Program partitioner partitions programs into program segments called phases. Ulrich Kremer in his thesis [18] introduced a good definition for phase. This component works based on Kremer's phase definition. Kremer assumed that loop can be used as basis of phase definition and defined phase as follows:

- Definition 1 : A phase is the outermost loop in a loop nest such that loop defines a induction variable that occurs in a subscript expression of an array reference in loop body.

Figure 3.1 shows the partition of an example program into phases using above phase definition. In our framework phase is used as unit of program and remapping is considered only between these units.

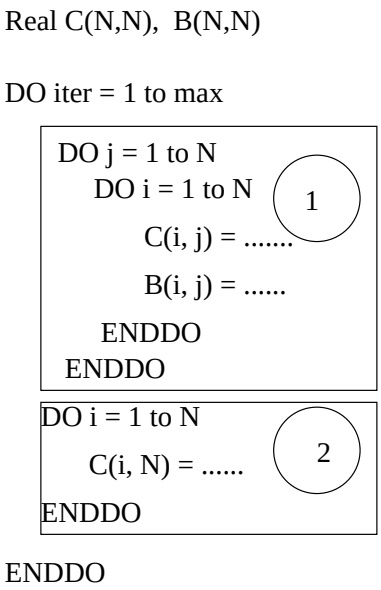


Figure 3.1: Partitioning of an example program

3.2 Program Analyzer

Program analyzer produces data access pattern of each program segment into their corresponding file into database. One data access pattern is generated corresponding to each iteration instance of a statement in the program segment. Data access pattern contains execution time of iteration instance of the statement, data size of data elements accessed in iteration instance of the statement and all data elements, accessed in the iteration instance of the statement. Our program analyzer works quite similar to traditional profilers [16]. Intuitively, to generate data access pattern for a statement in the program segment an additional statement is inserted after the statement. This additional statement writes required data access pattern to the corresponding file while program is executed by program analyzer to generate data access patterns.

3.3 Alignment Analyzer

Arrays are aligned to each other before distribution based on alignment preferences of program segment. Arrays are aligned by specifying a mapping of their elements to a common virtual array called template. Our discussion is restricted to mapping of array

dimensions to the template dimensions i.e. inter-dimensional alignment. Problem of inter-dimensional alignment is addressed by Li and Chen [21, 22]. Our alignment analyzer identifies alignment preferences of program segment and represents these preferences in the form of undirected, weighted graph, called component affinity graph. Alignment analyzer component of our tool works based on technique proposed by Li and Chen [21, 22]. In the next section we briefly describe technique proposed by Li and Chen for alignment analysis.

3.3.1 Li and Chen Approach of Alignment Analysis

A symbolic form notation is introduced to represent alignment preferences called reference pattern and defined as follows:

- Definition 2: Consider an array assignment statement appearing in a m-level multiple loop nest.

  for((i1,....,im):D)

  a(i1,....,im) = if (γ) then .... b(t1,....,tn)....else....

  for each array reference b(t1,....,tn) appearing on the right hand side of the assignment statement, the symbolic representation [a(i1,....im)← b(t1,....tn):γ] is called reference pattern. Where the indices (i1,....im) are quantified over index domain D and γ is the guard of the conditional branch that b(t1,....,tn) is in.

They introduced a graphical representation of relative alignment preferences, in the form of undirected, weighted graph, called component affinity graph (CAG). The nodes of the graph represent the components of index domains i.e. dimensions of the arrays to be aligned. For each distinct reference pattern in the program, an edge is generated between two nodes if the corresponding domain components have affinity. Affinity is defined as follows:

- Definition 3: Given a reference pattern [a(i1,..,ip,...,im)← b(t1,...,tq,...,tn):γ], two domain component dom(a,p) and dom(b,q) are said to have an affinity relation if $t_q \cong i_p + c$, where c is a small constant.

Each edge of CAG is assigned a weight to reflect the strength of preference.

3.3.2 Our Refinement: Preference strength

We introduced a new definition to determine the strength of preference.

- Definition : Consider an array assignment statement appearing in an m-level multiple loop nest.

  for((i1,....,im):d1 × .... × dm)

  a(i1,....,im) = if (γ) then .... b(t1,....,tn)....else....

  the quoted symbolic form [a(i1,....im)← b(t1,....tn):γ] is a reference pattern. Strength of this reference pattern is $R(d_1) \times .... \times R(d_m)$, where the index i_q is quantified over the index domain d_q and $R : D \rightarrow Z^+$ returns range of given index domain.

3.4 Alignment Conflict Resolver

Alignment problem in terms of CAG can be formulated as follows: partition the node set of CAG into n disjoint subsets $V_1, V_2, ..., V_{n-1}, V_n$, with the restriction that no two nodes belonging to the same index domain are allowed to be in the same subset. Here n is the maximum dimensionality of all index domains to be aligned. A CAG may have conflict if there is path between two nodes representing distinct dimensions of same index domain. Alignment conflict resolver partitions CAG such a way that total weight of edges that are between nodes that are in different subset is minimum. Li and Chen [21] have shown that alignment problem defined in terms of CAG can be proved NP-Complete and proposed a heuristic algorithm.

3.4.1 Li's and Chen's Heuristic Algorithm

Step 1: $C_T \leftarrow$ Index domain of CAG with maximum number of nodes

Step 2: For Each other index domain C_x do

//Set cost of edge between every pair of (dim(C_T),dim(C_x))

considering other alignment preferences in CAG

Step 2.1: $G_x \leftarrow$ Form_Bipartite_graph(C_T , C_x)

//Weighted Bipartite Matching

Step 2.2: M ← Optimal_alignment(G_x)

//Merging of Matched nodes

Step 2.3: CAG ← Reduce_graph(M, C_T, C_x, CAG)

3.4.2 Our Refinement: Modified Form_Bipartite_Graph

We modified Form_bipartite_graph part of previous algorithm slightly for more accurate results. Here we describe refined form_bipartite_graph procedure.

Form_bipartite_graph(C_T, C_X)

For each node pair (x, y) where x ∈ C_T and y ∈ C_X

Step 1: Construct Graph G by removing all the nodes in C_T and C_X except for x and y and all the edges that are incident on those nodes.

Step 2: Set up the edge between x and y if they are connected.

Step 3: If the edge exists assign the maximum of the weights to the *merged critical paths* in the connected component that x and y belongs to.

Merged critical path can be defined as follows:

- Definition 4: Merged critical path between two nodes x and y, is a path in the CAG that contains x and y with the restriction that no two nodes belonging to the same index domain can be on the path. Term merge enforces that weight of critical path is sum of all the edges that are between any two nodes of this critical path.

In older algorithm sum of edges in the connected component that x and y belongs is assigned as weight in step 3. Figure 3.2 explains how this refinement gives more accurate results.

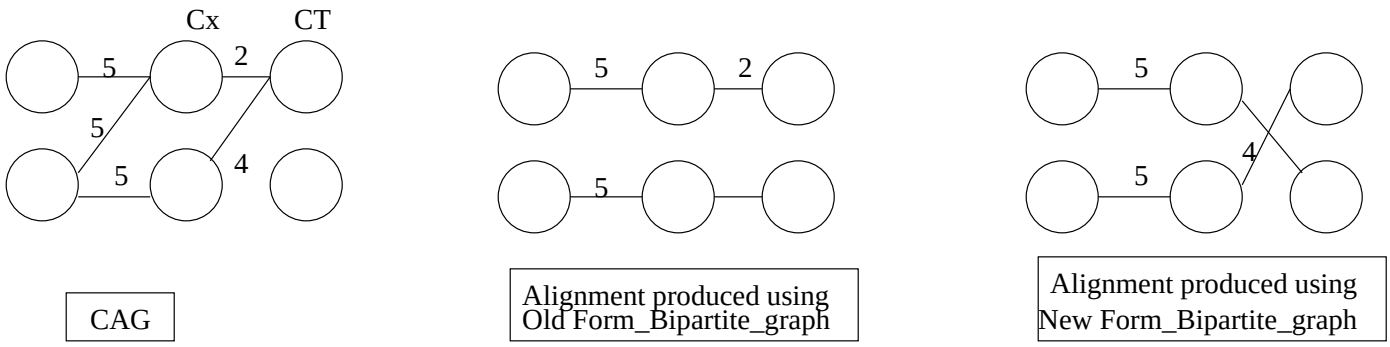


Figure 3.2: New Form_bipartite_graph procedure

Chapter 4

Distribution Analysis

After aligning arrays to a common template, this template is distributed over available processors. Each dimension of template can be replicated, localized or distributed blockwise or block-cyclically [13]. Our heuristic to find out optimal distribution doesn't consider replication. Techniques to find out optimal distribution can be classified into two types. Exhaustive heuristic first construct set of all possible distribution candidates without considering program characteristics. Cost is evaluated for each candidate in set and based on cost optimal one is choosed. Constructive heuristics find out the most optimal distribution choice considering program characteristic. To improve performance it may create set of some promising distribution candidates and later choses best one.

4.1 Heuristic Algorithm

In this section we present our constructive heuristic algorithm to find out optimal data distribution. Concept of partitioning and scheduling program for multiprocessors is intensively used in our algorithm. Sarkar [26] presents efficient approximation algorithm for partitioning and scheduling program for multiprocessor problem. Multiprocessor model proposed by Sarkar exactly matches with our distributed memory machine model. We have used his algorithm to determine parallel execution time of a partitioned and scheduled program. We simplified and modified this algorithm according to our requirements. Algorithm for program partitioning and scheduling from Sarkar is embedded in

our heuristic algorithm in simplified form. After describing Some notations and distributed memory machine model we defined data structures and operations used in our algorithm, then we presented modified algorithm to calculate parallel execution time. At the end of this section we discuss our heuristic algorithm to find out optimal distribution.

4.1.1 Notations

- Z^+ Set of positive integers
- Q^+ Set of non-negative rational numbers
- D Domain of data elements

4.1.2 Distributed Memory Machine Model

This section describes our abstraction of Distributed Memory Machine multiprocessor organisations, which is the target machine model used for automatic data layout tool. A distributed memory machine can be modeled as a collection of identical, communicating processors. The only inter-processor interaction in this model is data communication through message passing.

Definitions:

- Rc:$Z^+ \times Z^+ \times Q^+ \times Q^+ \rightarrow Q^+$ is the communication overhead function for reading data. Rc(i, j, s, l) is the overhead incurred by processor j to receive s units of data from processor i, j$\neq$i, assuming a communication load of l per processor.
- Wc:$Z^+ \times Z^+ \times Q^+ \times Q^+ \rightarrow Q^+$ is the communication overhead function for writing data. Wc(i, j, s, l) is the overhead incurred by processor i to send s units of data to processor j, i$\neq$j, assuming a communication load of l per processor.
- Dc:$Z^+ \times Z^+ \times Q^+ \times Q^+ \rightarrow Q^+$ is the communication delay overhead function. Dc(i, j, s, l) is the delay time which must elapse when processor i sends s units of data to processor j, j$\neq$i, assuming a communication load of l per processor.

4.1.3 Operations and Data Structures

- ANALYSIS: is collection of data access patterns of a program segment and is generated by Program Analyzer.
- TRACE: is transformed data access pattern of a program segment.
- Transform: ANALYSIS $\rightarrow$ TRACE Data access pattern generated for program segment must be transformed according to specified alignment. Transformation is needed because different arrays in data access pattern will be aligned to a single template with different specified alignments. It can easily be transformed by the following technique : if there is an array element a($i_1, ...i_n$) in data access pattern and array a is aligned with alignment function g:D$\rightarrow$D and g($i_1, ..., i_n$) = ($i_{q_1}, ...i_{q_n}$) where ($q_1, ..., q_n$) is permutaion of (1,....,n), then replace a($i_1, ..., i_n$) by T($i_{q_1}, ..., i_{q_n}$) in the generated trace, here T is single template on which all the arrays aligned. When this template is distributed all arrays aligned to this template are distributed automatically.
- Create_TDG:TRACE $\rightarrow$ TDG(Trace Dependency Graph) This graph represents generated trace pattern of program segment and dependecies between them in a data structure. Each trace pattern is represented by one node in the graph and there are edges between nodes representing communication requirement. Each node is assigned a number in increasing order, called ID. For every template element t_k in the a trace pattern Tr_j, if any nearest previous trace pattern Tr_i (ID(Tr_i) $<$ ID(Tr_j)) contains the element t_k, an edge is marked, from Tr_i to Tr_j with t_k as attribute of edge. Formally TDG = (V, E), where V is set of trace patterns. E = { $(Tr_i, Tr_j, t_k) \mid t_k \in Tr_i \wedge t_k \in Tr_j \wedge ID(Tr_i) < ID(Tr_j) \wedge \forall m, ID(Tr_i) < ID(tr_m) < ID(Tr_j)(t_k \ni Tr_m)$ }.
- TIME:V$\rightarrow Q^+$ is the execution time of a trace pattern.
- SIZE:D$\rightarrow Q^+$ is the data size of array element.
- PA:V$\rightarrow Z^+$ is processor number to which this trace pattern is assigned for computation.

- TOTALTIME:V$\rightarrow Q^+$ is the total execution time of a trace pattern node including overheads of all non-local communication.

- EST:V$\rightarrow Q^+$ is the earliest start time at which trace pattern node can start execution.

- ID:V$\rightarrow Z^+$ is a unique identification number (>P) given to each trace pattern node. Sometimes used as a assigned virtual processor number to that node.

- FIRST:V $\times$ D $\times$ Z^+ $\rightarrow$ V is the first node on given processor which need given data element from given trace pattern node i.e. FIRST(Tr_i, t_k, p) = Tr_j s.t. PA(Tr_j) = p $\wedge \exists (Tr_i, Tr_j, t_k) \in E \wedge \forall Tr_l \in V((Tr_i, Tr_l, t_k) \in E \wedge PA(Tr_l) = p) \rightarrow ID(Tr_l) \geq ID(Tr_j))$

- Merge:(V,E) $\times$ Z^+ $\times$ Z^+ $\rightarrow$ (V, E)To remove a communication edge source and destination trace pattern nodes of the edge must have same assigned processor. Merge(G, SRC, DESTI) works as follows:

  FOR each n $\in$ V DO

  OLD_PA(n) $\leftarrow$ PA(n)

  IF(PA(n) = DESTI)

  PA(n) $\leftarrow$ SRC

- Restore_graph:(V, E) is used to restore graph to its state before the latest Merge() call. Restore_graph() works as follows:

  FOR each n $\in$ V DO

  PA(n) $\leftarrow$ OLD_PA(n)

- IDX:D $\times$ Z^+ $\rightarrow$ Z^+ is the index value of given dimension of the data element i.e. IDX($a(I_1, ...I_q, ...I_m)$, q) = I_q.

- Accomodate: DC $\times$ D $\rightarrow$ Z^+ given Distribution candidate and data element returns processor number, where this data element will reside with this distribution choice. Processor number is calculated similar to address calculation of an array

element. Suppose $((b_1, p_1, t_1), ..., (b_{maxdims}, p_{maxdims}, t_{maxdims}))$ is a distribution choice, where (b_q, p_q, t_q) represents block size, number of processors allocated and type of distribution i.e. blockwise or block-cyclically respectively for q^{th} dimension of the template. Then $offset_i$ for i^{th} dimension is calculated as $offset_i \leftarrow$ IDX(a, i) - 1 / b_i MOD p_i. Finally Accomodate $\leftarrow (....((offset_1) * p_2 + offset_2) * p_3 + ... + offset_{maxdims})$

- LHS:V$\rightarrow$D is data element being updated in given trace pattern.
- COMM: Set of communication edges which actually require communication for a given processor assignment of nodes in TDG i.e. $\subseteq$ E.
- AAE_{proc}: Set of template elements s.t. their corresponding node in TDG is assigned to proc.
- LOAD: is raw estimate of communication load.
- P: is the number of available processors.
- ALIGNMENT: represent alignment of all the arrays of a program segment, to a single template.
- MAXDIMS: number of dimensions of the program template.
- PARTIME: is parallel execution time of a TDG with given PA, TIME, SIZE etc.
- SSPACE: is set of guessed distribution for one dimension. Guess is made considering program characteristics. Elements of the set are 3-tuple. representing assigned processor number, dimension and distribution specification respectively. Distribution specification is again 3-tuple representing block-size, number of processors and type of distribution respectively.
- SEARCH_SPACE: is set of all promising distribution candidates.
- CURRENT_SSN: represents one distribution candidate of SEARCH_SPACE.
- LDB: is a set of 3-tuple elements. Each element represents a association between a distribution choice, a node in TDG and processor assignment of this node for this distribution choice.

4.1.4 Algorithm to Determine Parallel Execution Time of T-DG

To compare performance of two different processor assignments to the nodes of TDG, parallel execution time of TDG is needed. Procedure Determine_time initializes COMM with communication edges that actually require communication. For each node in TDG TOTALTIME is calculated and it is equal to execution time plus communication overheads. For each node in TDG Earliest Start time(EST) is calculated and using this PARTIME is calculated.

Procedure Determine_time

Inputs:

1. Trace dependency Graph G = (V, E)
2. Execution time of each node N $\in$V, TIME:N$\rightarrow Q^+$
3. Communication size of each edge C $\in$E, SIZE:C$\rightarrow Q^+$
4. Processor Assignment to each node N $\in$V, PA:N$\rightarrow Z^+$
5. The Communication overhead function, R_c, W_c, D_c.

Outputs:

1. The Parallel execution time, PARTIME:Q^+

Algorithm

$$COMM(G) \leftarrow \{(Tr_i, Tr_j, t_k) \mid (Tr_i, Tr_j, t_k) \in E, PA(Tr_i) \neq PA(Tr_j) \wedge Tr_j = FIRST(Tr_i, t_k, PA(Tr_j))\}$$

LOAD $\leftarrow \Sigma_{(Tr_i,Tr_j,t_k)\in COMM}$ SIZE(t_k) / $\Sigma_{n\in V}$ TIME(n)

FOR every node n $\in$V DO
  TOTALTIME(n) = TIME(n)
    + $\Sigma_{(Tr_i,Tr_j,t_k)\in COMM \wedge Tr_j = n}$ Rc(PA(Tr_i), PA(Tr_j), SIZE(t_k), LOAD)

+ $\Sigma_{(Tr_i,Tr_j,t_k)\in COMM \wedge Tr_i=n}$ Wc(PA(Tr_i), PA(Tr_j), SIZE(t_k), LOAD)

FOR every node n $\in$ v DO
EST(n) = max ({0} $\cup$ { EST(Tr_i) + TOTALTIME(Tr_i)
+ Dc(PA(Tr_i), PA(Tr_j), SIZE(t_k), LOAD)
| $(Tr_i, Tr_j, t_k) \in$ COMM $\wedge$ Tr_j = n}
$\cup$ { EST(m) + TOTALTIME(m) | PA(m) = PA(n) $\wedge$ ID(m) < ID(n)})

PARTIME $\leftarrow$ max({EST(n) + TOTALTIME(n) | n $\in$ V})

4.1.5 Algorithm to Determine Parallel Execution Time with Distribution Specified

To compare performance of two distribution candidates, parallel execution time of the Trace Dependency Graph is needed. Processor assignment of nodes of TDG is made taking in account corresponding entries in LDB of distribution choice. Initially COMM set is calculated as in previous algorithm. PA of a TDG node is either processor number of LHS of TDG, determined by distribution choice or ID of TDG node. There may be cases in which processor number of elements other than LHS of TDG, determined by distribution choice, may not be same as PA of TDG, and hence require communication. For these cases some additional edges is added into COMM set. To represent this communication requirement as edges some dummy nodes are inserted into TDG, PA of dummy node $\top_i$ is i and execution time, TIME of $\top_i$ is zero. Now onwards PARTIME calculation is same as previous procedure.

Procedure Determinetime_withDistribution

Inputs:

1. Trace dependency Graph G = (V, E)
2. Execution time of each node N $\in$V, TIME:N$\rightarrow Q^+$
3. Communication size of each edge C $\in$E, SIZE:C$\rightarrow Q^+$

4. Distribution choice, d ∈ SEARCH_SPACE

5. The Communication overhead function, Rc, Wc, Dc.

Outputs:

1. The Parallel execution time, PARTIME:Q^+

Algorithm

FOR each n ∈ V
  IF((d, n, proc) ∈ LDB)
    PA(n) ← proc
  ELSE
    PA(n) ← ID(n)

$COMM(G) \leftarrow \{(Tr_i, Tr_j, t_k) \mid (Tr_i, Tr_j, t_k) \in E, PA(Tr_i) \neq PA(Tr_j) \wedge$
$Tr_j = FIRST(Tr_i, t_k, PA(Tr_j))\}$

$COMM(G) = COMM(G) \bigcup \{(\top_i, Tr_j, t_k) \mid Tr_j \in V, t_k \in Tr_j,$
$\forall Tr_l (Tr_l, Tr_j, t_k) \ni E, Accomodate(d, t_k) \neq PA(Tr_j) \wedge Accomodate(d, t_k) = i$

LOAD ← $\Sigma_{(Tr_i, Tr_j, t_k) \in COMM}$ SIZE(t_k) / $\Sigma_{n \in V}$ TIME(n)

FOR every node n ∈V DO
  TOTALTIME(n) = TIME(n)
    + $\Sigma_{(Tr_i, Tr_j, t_k) \in COMM \wedge Tr_j = n}$ Rc(PA(Tr_i), PA(Tr_j), SIZE(t_k), LOAD)
    + $\Sigma_{(Tr_i, Tr_j, t_k) \in COMM \wedge Tr_i = n}$ Wc(PA(Tr_i), PA(Tr_j), SIZE(t_k), LOAD)

FOR every node n ∈ v DO
  EST(n) = max ({0} ∪ { EST(Tr_i) + TOTALTIME(Tr_i)
    + Dc(PA(Tr_i), PA(Tr_j), SIZE(t_k), LOAD)
    | (Tr_i, Tr_j, t_k) ∈COMM ∧ Tr_j = n}
    ∪ { EST(m) + TOTALTIME(m) | PA(m) = PA(n) ∧ ID(m) < ID(n)})

PARTIME ← max({EST(n) + TOTALTIME(n) | n ∈ V})

4.1.6 Main Algorithm to Find Out Optimal Distribution

Inputs:

1. File containing data access pattern of the program segment, ANALYSIS
2. Suggested alignment for that segment, ALIGNMENT
3. Number of available Processors, P

Outputs:

1. The optimal Distribution layout for that program segment, CURRENT_SSN.

Algorithm

//Transform data access patterns using specified alignment

Step1: TRACE ← Transform(ANALYSIS, ALIGNMENT)

//Create TDG G using transformed data access patterns

Step2: G ← Create_TDG(TRACE)

//Assign PA of each node to its own id

Step3: FOR each Trace pattern $Tr_i \in$ V

PA(Tr_i) ← ID(Tr_i)

//Find out parallel execution time for G

Step4: PARTIME ← Determine_time(G)

//find out COMM set in the G

Step5: COMM(G) ← {(Tr_i, Tr_j, t_k) | $(Tr_i, Tr_j, t_k) \in$ E, PA(Tr_i) ≠ PA(Tr_j)
$\wedge\ Tr_j$ = FIRST($Tr_i, t_k, PA(Tr_j)$)}

// Assign processor to each trace pattern, efficiently

Step6: FOR each communication edge $(Tr_i, Tr_j, t_k) \in$ COMM DO

Merge(G, PA(Tr_i), PA(Tr_j))

NEW_PARTIME ← Determine_time(G)

IF (NEW_PARTIME $<$ PARTIME)
PARTIME $\leftarrow$ NEW_PARTIME
ELSE
Restore_graph(G)

//Schedule Graph nodes on P processors

Step 7: For every trace pattern node n $\in$ V DO
PARTIME $\leftarrow \infty$
FOR proc = 1 TO P DO
Merge(G, proc, PA(n))
NEW_PARTIME $\leftarrow$ Determine_time(G)
IF(NEW_PARTIME $<$ PARTIME)
PARTIME $\leftarrow$ NEW_PARTIME
$P_{min} \leftarrow$ proc
Restore_graph()
Merge(G, P_{min}, PA(n))

//Start creating search space of candidate distributions

Step8: FOR proc = 1 TO P DO
//Compute ordered Set of assigned array elements for this proc
AAE_{proc} = { $a \mid a \in$ n $\wedge$ PA(n) = proc, $\forall n \in V$}
// Generate candidate distribution for each dimension
FOR DIM = 1 TO MAXDIMS DO
LASTELEMENT $\leftarrow$ NULL
// Sort array elements in AAE_{proc}
based on index values of DIM dimension
Sort_AAE(AAE_{proc}, DIM)
FOR every data element da $\in AAE_{proc}$, starting from first DO
IF (LASTELEMENT = NULL)
LASTELEMENT $\leftarrow$ da
BLOCKSIZE $\leftarrow$ 1

ELSE

DIFF ← ABS(IDX(LASTELEMENT, DIM) - IDX(da, DIM))

IF(DIFF = 1)

LASTELEMENT ← da

BLOCKSIZE ← BLOCKSIZE + 1

ELSE

BSIZE ← BLOCKSIZE

PROCS ← DIFF + BLOCKSIZE - 1/ BLOCKSIZE

TYPE ← BLOCK_CYCLIC

SSPACE = SSPACE ⋃ {(proc, DIM, (BSIZE, PROCS, TYPE))}

LASTELEMENT ← da

BLOCKSIZE ← 1

//Consider Block distribution of this dimension

IF(BLOCKSIZE > 1)

BSIZE ← BLOCKSIZE

PROCS ← Number_elements(DIM) / BLOCKSIZE

TYPE ← BLOCK

SSPACE = SSPACE ⋃ {(proc, DIM, (BSIZE, PROCS, TYPE))}

//Try all combination of candidate distribution for each dimension

SEARCH_SPACE = $\{((b_1, p_1, t_1), ..., (b_{maxdims}, p_{maxdims}, t_{maxdims}))$

$| (pr_i, i, (b_i, p_i, t_i)) \in SPACE \wedge pr_1 = pr_2 = ... = pr_{maxdims}\}$

//Insert all possible BLOCK and CYCLIC(1) distribution candidate

SEARCH_SPACE = SEARCH_SPACE ⋃ $\{((b_1, p_1, t_1), ..., (b_{maxdims}, p_{maxdims}, t_{maxdims}))$

$| b_i = 1 \wedge t_i = CYCLIC \wedge \prod_{i=1}^{maxdims} p_i \leq P \wedge \exists i(p_i > 1)\}$

SEARCH_SPACE = SEARCH_SPACE ⋃ $\{((b_1, p_1, t_1), ..., (b_{maxdims}, p_{maxdims}, t_{maxdims}))$

$| b_i = Number\_elements(i)/p_i \wedge t_i = BLOCK$

$\wedge \prod_{i=1}^{maxdims} p_i \leq P \wedge \exists i(p_i > 1)\}$

//Modify infeasible Distribution Candidates

Step 9:FOR each distribution candidate

$((b_1, p_1, t_1), ..., (b_{maxdims}, p_{maxdims}, t_{maxdims})) \in$ SEARCH_SPACE DO

MOD_COUNT ← 1

WHILE ($\prod_{i=1}^{maxdims} p_i > P$) DO

DIM2MOD $\leftarrow$ (MOD_COUNT MOD MAXDIMS) + 1

K $\leftarrow$ Min_Prime_Divisor($p_{DIM2MOD}$)

$b_{DIM2MOD} \leftarrow b_{DIM2MOD}$ * K

$p_{DIM2MOD} \leftarrow p_{DIM2MOD}$ / K

MOD_COUNT $\leftarrow$ MOD_COUNT + 1

//Now try to find out optimal distribution

Step10: FOR each Trace pattern $Tr_i \in$ V

PA(Tr_i) $\leftarrow$ ID(Tr_i)

//Find out parallel execution time for G

PARTIME $\leftarrow$ Determine_time(G)

//find out COMM set in G

COMM(G) $\leftarrow$ {$(Tr_i, Tr_j, t_k) \in$ E | PA(Tr_i) $\neq$ PA(tr_j)

$\wedge Tr_j$ = FIRST(Tr_i, t_k, PA(Tr_j))}

// Try each distribution for every COMM edge merging

FOR each communication edge $(Tr_i, Tr_j, t_k) \in$ COMM DO

Merge(G, PA(Tr_i), PA(Tr_j)

NEW_PARTIME $\leftarrow$ Determine_time(G)

IF (NEW_PARTIME $<$ PARTIME)

FOR each candidate distribution d $\in$ SEARCH_SPACE DO

IF(Accomodate(d, LHS(Tr_i)) = Accomodate(d, LHS(Tr_j)))

LDB = LDB $\cup$ {(d, Tr_i, Accomodate(d, LHS(Tr_i)))}

LDB = LDB $\cup$ {(d, Tr_j, Accomodate(d, LHS(Tr_j)))

NEW_PARTIME $\leftarrow$ Determinetime_withDistribution(G, d)

IF(NEW_PARTIME $<$ PARTIME)

PARTIME $\leftarrow$ NEW_PARTIME

CURRENT_SSN $\leftarrow$ d

FOR each n $\in$ V

IF((d, n, proc) $\in$ LDB)

PA(n) $\leftarrow$ proc

ELSE

PA(n) $\leftarrow$ ID(n)

ELSE

Restore_graph(G)

//Desired distribution candidate is represented by CURRENT_SSN

Step11: EXIT with CURRENT_SSN

The above algorithm is an efficient heuristic algorithm to find out optimal data distribution for a program segment. In step 1 Data access patterns of program segment are transformed into trace patterns. These trace patterns are used to create Trace Dependency Graph(TDG). Each trace pattern is assigned to a virtual processor and parallel execution time of TDG is calculated using Determine_time. In next step COMM set is calculated for TDG. In step 6 processor assignment for each trace pattern is determined s.t. parallel execution time is minimum possible . In step 7 trace pattern are assigned to actual processors, efficiently. In step 8 search space of promising distribution candidates is created. For all processors assigned array elements are computed in their AAE sets and based on that desired distribution guessed for each dimension of template. These guessed distribution are used to create search space of promising distribution candidates, in the last part of step 8 all possible BLOCK and CYCLIC(1) distribution for available number of processors are inserted in search space. Kremer [18] in his work proved that total number of possible BLOCK and CYCLIC(1) distribution for procs = p^k, where p is some prime number and d-dimensional program template is $\Sigma_{i=1}^{d} 2^i * \binom{d}{i} * \binom{k-1}{i-1}$. In step 9 infeasible distribution choices i.e. requiring more processors than available are modified s.t. data element resides as desired and processor required reduced. In step 10 processor assignment for each node in TDG is reinitialized to virtual processor . COMM set is computed and for each edge in COMM set it is checked that removal of this edge is beneficial. If beneficial then all distribution choices in search space is tried that whether it accomodate source and destination nodes of edge on same processor. Since we are assuming owner compute rule for computation, accomodate simply means LHS of both source and destination nodes of the edge reside on same processor. If a distribution choice accomodates that edge entries are made in LDB representing association between distribution choice, source and destination nodes and their assigned processors.

Also parallel execution time of TDG with this distribution choice is determined. it uses LDB set to assign PA of source and destination nodes of all previously accomodated edges. If parallel execution time is better then this distribution choice is made CURRENT_SSN. This work is repeated for each edge in COMM set. In step 11 algorithm exits with desired distribution choice. It has 3-tuple element for each dimension of template representing distribution layout for that dimension. Example in Fig. 4.1 illustrates the functioning of above algorithm

MAXELEMENT = 18

Real A(MAXELEMENT)

Stride = MAXELEMENT / 3

FOR I = 1 TO 11 in Steps of 2 DO
A(I) = A(I + 1) * A(I + Stride)
A(I+1) = A(I) * A(I + Stride +1)

AAE(1)={ A(1), A(2), A(7), A(8)}

AAE(2)={ A(3), A(4), A(9), A(10)}

AAE(3)={ A(5), A(6), A(11), A(12)}

SSPACE

(BSIZE = 2 ,
PROCS = 3 ,
TYPE = BLOCK_CYCLIC)

ANALYSIS and TRACE

s, t, A(1), A(2), A(7)
s, t, A(2), A(1), A(8)

s, t, A(3), A(4), A(9)
s, t, A(4), A(3), A(10)

s, t, A(5), A(6), A(11)
s, t, A(6), A(5), A(12)
s, t, A(7), A(8), A(13)
s, t, A(8), A(7), A(14)
s, t, A(9), A(10), A(15)
s, t, A(10), A(9), A(16)

s, t, A(11), A(12), A(17)
s, t, A(12), A(11), A(18)

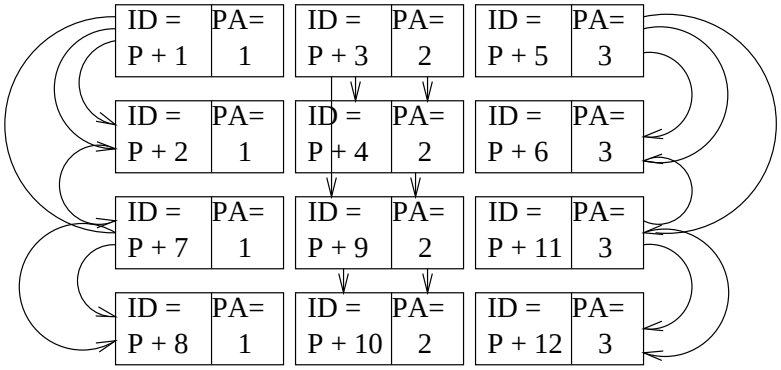
SEARCH_SPACE

BSIZE = 2
PROCS = 3
TYPE = BLOCK_CYCLIC

BSIZE = 1
PROCS = P
TYPE = CYCLIC

BSIZE = 6
PROCS = P
TYPE = BLOCK

TDG After Processor Assignment



CURRENT_SSN after Final step of Algorithm

BSIZE = 2
PROCS = 3
TYPE = BLOCK_CYCLIC

Figure 4.1: Functioning of Distribution Analyzer

Chapter 5

Remapping Analysis

The algorithm described in this chapter generates the optimal data layout of the program with appropiate remapping and is implemented in controller component of our data layout tool. It uses Alignment Analyzer, Alignment Conflict Resolver, Distribution Analyzer, Cost estimator, Data base files etc. All the operations in the algorithm are performed on Phase Affinity Tree(PAT). PAT is a tree where root is a dummy node representing whole program and each phase i.e. program segment is represented by its children nodes. If phases are merged they are replaced by a subtree with a dummy node as its root and all merged phases as its children nodes.

Main Algorithm

G $\leftarrow$ Phase Affinity Tree(PAT)

dist $\leftarrow$ 1

Step 1: GENERATE conflict-free alignment for every phase using Alignment Analyzer and Alignment Conflict Resolver

Step 2: GENERATE distribution for every phase using alignment from Step 1 and Distribution Analyzer.

Step 3: IF (dist $\geq$ #phases) GOTO step 11

Step 4: FOR every phase i $\in$ G DO

Step 4.1: IF (i+dist $>$ #phases) GOTO Step 5

Step 4.2: Compute ($inter\_align_{i,i+dist}$, $inter\_distri_{i,i+dist}$)

by merging phases i and i+dist and computing their composite alignment and distribution

Step 4.3: Compute merge_benefit using formula

merge_benefit = $\Sigma_{j=i}^{i+dist}$ COST(j, $align_j$, $distri_j$) + $\Sigma_{j=i}^{i+dist-1}$ REMAP_COST($align_j$, $distri_j$, $align_{j+1}$, $distri_{j+1}$) - COST(merge_phase(I,....,I+dist),$inter\_align_{i,i+dist}$, $inter\_distri_{i,i+dist}$)

Step 4.4: IF merge_benefit > 0 put an edge ed_i between phase i and i+dist with weight merge_benefit

Step 5: Choose the edge ed_j with highest weight

Step 6: IF (ed_j = NULL) GOTO Step 10

Step 7: G $\leftarrow$ merge(j,....,j+dist)

Step 8: FOR every phase k, s.t. j-dist $\leq$ k $\leq$ j

IF (k+dist $\leq$ #phases)

PERFORM steps 4.2, 4.3, 4.4 for phase k

Step 9: GOTO Step 5

Step 10: dist $\leftarrow$ dist+1 and GOTO Step 3

Step 11: EXIT with desired data layout

The above algorithm is to find out optimal data layout with dynamic realignment and redistribution for the program. In this algorithm affinity between phases i.e. program segments are tested and if beneficial phases are merged into a single phase. Step 1 Generates conflict-free alignment for each phase, in step 2 optimal distribution is detemined for each phase using alignment of previous step. At first affinity between neighbour phases is tested i.e. when dist is 1. Affinity is measured by value of merge_benefit i.e. benefit on merging corresponding phases. In step 4.2 phases are merged by combining their CAGs and data access patterns. Optimal alignment i.e. inter_align for merged phases is calculated by resolving conflicts in combined CAG using alignment conflict resolver. Optimal distribution i.e. inter_distri for merged phases is calculated by determining optimal distribution for combined data access patterns using distribution analyzer. Phases with highest affinity are chosen in step 5 and merged in step 7. For all effected phases merge_benefit is recalculated in step 8, now again phase merging is

tried. All these above steps are performed with dist increased. In step 11 algorithm exits with desired data layout.

Chapter 6

Implementation

6.1 Programming Environment

6.1.1 SUIF Compiler

SUIF is a base system that can support collaborative research and development efforts. It has been structured as a small kernel plus a toolkit consisting of various compilation analyses and optimizations built using the kernel. The kernel defines the intermediate representation and the interface between passes of the compiler. This interface is always the same so that the passes in the toolkit can easily be enhanced, replaced, or rearranged. The intermediate program representation is a hierarchy of data structures defined in an object-oriented class library. This intermediate representation retains almost all the high-level information from the source code. Accessing and manipulating the data structures are generally straightforward due to the modular design of the kernel. The SUIF kernel performs three major functions: It defines the intermediate representation of programs. This representation supports both high-level program-restructuring transformations as well as low-level analyses and optimizations. It provides functions to access and manipulate the intermediate representation. Hiding the low-level details of the implementation makes the system easier to use and helps maintain compatibility if the representation is changed. It structures the interface between compiler passes. SUIF passes are separate programs that communicate via files. The format of these files is the same for all stages of a compilation. The system supports experimentation by allowing

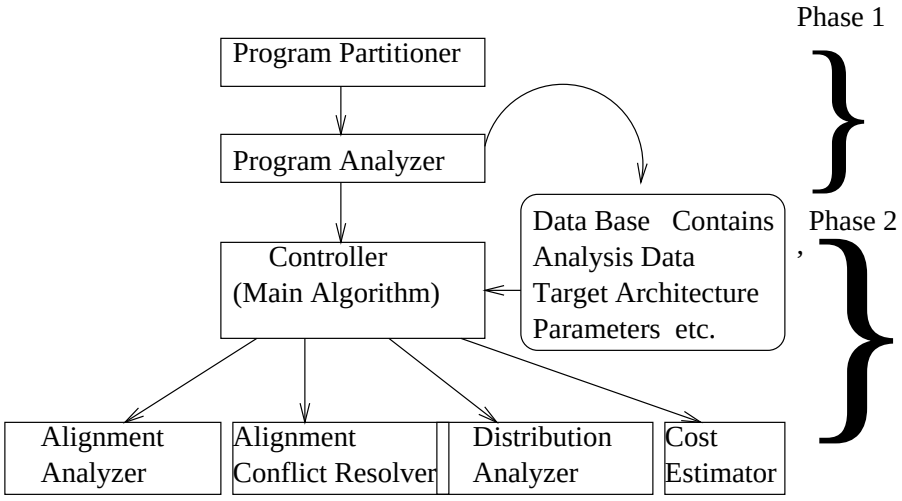


Figure 6.1: Components of Automatic Data Layout Tool

user-defined data in annotations.

6.2 Implementation Overview

We implemented a prototype automatic data layout tool based on our technique, to verify the results. We choosed IBM/SP2 as our target architecture, HPF as the target compiler and Fortran 77 programs as the target input. The tool based on our technique analyzes Fortran programs and generates optimal HPF data layout specifications. Program instructions execution time, communication cost, remapping cost and available number of processors for the above system, were made available to the prototype tool. Our automatic technique considers dynamic alignment and distribution. The data layout tool is used to generate a data layout with appropiate remapping for a sequential Fortran program without any data layout statements. The data layout is optimized for given system parameters i.e. data communication cost, remapping cost, number of available processors etc. It consists of two phases as shown in Fig 6.1. The first phase partitions program and generates data access patterns for each program part. In the second phase, the controller produces the optimal data layout for the program, with appropriate remapping.

Figure 6.2 shows various components of our tool separately and their corresponding

inputs and outputs, we describe their functioning here briefly. Our tool uses SUIF library functions and works on SUIF intermediate format of the program. SUIF compiler driver scc generates SUIF intermediate format for the given input program in the corresponding .spd file. As described previously our technique consists of two phases. In first phase program partitioner partitions program into program segments s.t. data remapping between these segments may be beneficial. Program partitioner takes SUIF .spd file as input and generates program segment information into a data structure called phase_tab. Partitioned program is assigned to program analyzer. Program analyzer generates data access pattern of each program segment into its corresponding file into database. Program analyzer uses information about phase segments, stored in phase_tab and ordered array list. Ordered array list is ordered list of all arrays declared in input program. Ordered array list is generated by program partitioner. In second phase controller produces optimal data layout with appropriate remapping for the program. Controller uses alignment analyzer, alignment conflict resolver, distribution analyzer and cost estimator. Alignment analyzer identifies alignment preferences of given program segment and represents these preferences in the form of a graph called component affinity graph (CAG), stored in phase_cag# file, # is replaced by program segment number. Alignment conflict resolver produces optimal conflict-free alignment in phase_align# file from a given CAG stored in phase_cga# file. Distribution analyzer generates optimal distribution of a program segment into phase_distri# file for a given alignment file phase_align# of that segment. Distribution analyzer also uses data access pattern of that program segment which is stored in corresponding phase_ana# file. Trace analyzer of distribution analyzer produces trace of program segment into file phase_trace# for the given data access patterns (phase_ana# file) and alignment(phase_align# file). Cost estimator estimates the execution cost of a program segment with alignment and distribution specified. In our tool cost estimator is embedded into the distribution analyzer. System parameters such as communication cost, remapping cost etc. are supplied to distribution analyzer.

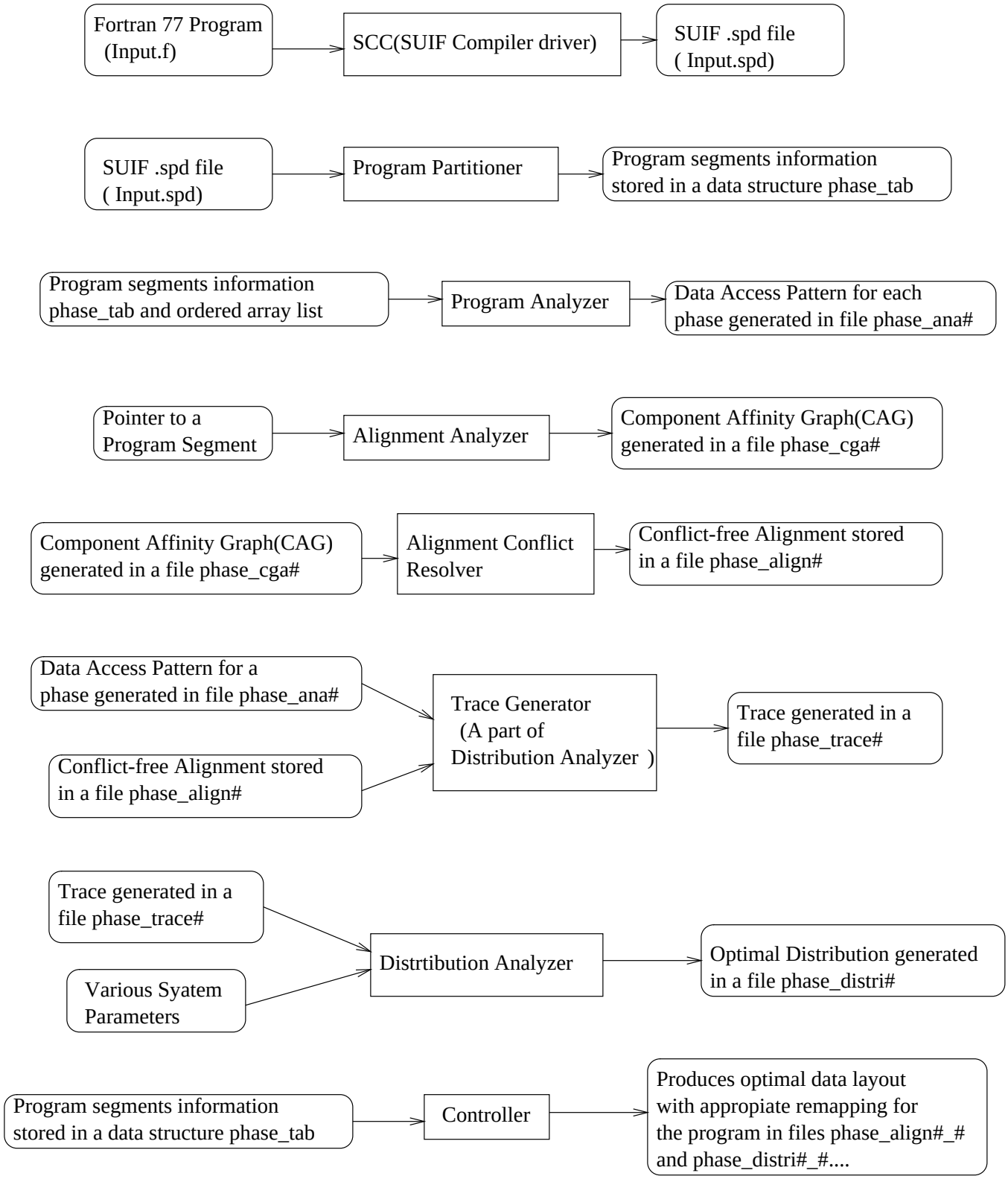


Figure 6.2: Implementation Overview

6.3 Implementation details

6.3.1 Program Partitioner

Program Partitioner is implemented in prog.cc file. Program partitioner is called separately for each procedure of the program. Program partitioner works on SUIF intermediate format (.spd format) of the program and uses SUIF library. It checks for each FOR_LOOP node that whether its index is being used in the inside computation. If index is used in the computation the subtree represented by this FOR_LOOP is marked as phase by storing sufficient information about this node in phase_tab data structure. If index is not used, children nodes of this FOR_LOOP are checked similarly. To implement this map() function of SUIF library is used intensively.

6.3.2 Alignment Analyzer

Alignment anlyzer genreates Component affinity graph(CGA) for each phase, which represents alignment preferences of the phase. This graph is stored in corresponding phase_cga# file of the phase. To generate CGA phase_cga() function is called for the phase. phase_cga function is defined in is_lhs.cc file. This function uses SUIF library exhaustively. For each TREE_INSTR tree node of corresponding SUIF intermediate form of phase alignment preferences extracted. To identify alignment preference of a instruction its LHS is identified using is_lhs() function. For remaining arrays of instruction reference patteern is determined w.r.t. LHS and represented in CGA.

6.3.3 Alignment Conflict Resolver

Alignment Analyzer is implemented in two files, namely align_ana.c and mod_tdm.c. The program align_ana.c, first reads CGA from the input phase_cga# file. For each array to be aligned, optimal weighted matching procedure is applied to a bipartite graph. Bipartite graph is constructed from the corresponding nodes of array and target array. Target array is array with maximum dimension and all the arrays are aligned to this array. Two nodes in a bipartite graph is connected if there is path between these two nodes. The weight of such an edge is maximum of weights of all merged critical path of corresponding

connected component subgraph. Matching produced by the this procedure is optimal alignment for this arbitrary array and target array. The file mod_tdm.c contains optimal weighted matching procedure and used by the program align_ana.c. Both programs, align_ana.c and mod_tdm.c are C programs.

6.3.4 Distribution Analyzer

This procedure is implemented in part_graph.c file. Trace generator part of distribution analyzer is implemented in prog.cc file with procedure name gen_trace(). The procedure gen_trace() generates transformed data access pattern (traces) into phase_trace# file, for the given phase_ana# and phase_align# files. In the part_graph.c file first trace dependency graph(TDG) is created by reading traces from phase_trace# file. read_gr() function performs this task. In next step processor assignment for each trace pattern is determined s.t. parallel execution time is minimum possible. Function det_time() is used to determine parallel execution time for a processor assignment of TDG. Distribution for each dimension of template is guessed based on best processor assignment of TDG nodes and stored in their corresponding sspace distribution candidate list. These guessed distribution are used to create search space of promising distribution candidates using recursive merge_sspace() procedure. Promising distribution candidate search space is pointed by SEARCH_SPACE pointer. All possible BLOCK and CYCLIC(1) distribution for available number of processors are inserted in search space using add_bc_perm() function. In the next step infeasible distribution choices i.e. requiring more processors than available, are modified by invoking remove_unfeasible() procedure. For each communication edge function mod_current() is called that modifies CURRENT_SSN, s.t. it represents the optimal distribution candidate of distribution candidate search space. The function mod_current() uses accomodate() function to find out processor number of a data element for the given distribution. The function mod_current() also uses add_pa() procedure to add entry in LDB and mod_det_time() to determine parallel execution time with distribution considered.

Chapter 7

Experimental Results

Experiments were done to show that our approach is efficient and generates good data layouts. We implemented a prototype automatic data layout tool based on our technique, to verify the results. We chose IBM/SP2 as our target architecture, HPF as the target compiler and Fortran 77 programs as the target input. Our tool generates HPF data layout specifications for the target input program. Program instructions execution time, communication cost, remapping cost and available number of processors for the above system, were made available to the prototype tool.

The goal of experiments was to show the ability of tool to correctly generate optimal data layout. It is important to note that experiments were not aimed to analyse absolute performance of target input programs. Performance of generated programs are always relative to the quality of target compilers.

The goal of this thesis is to explore some issues related to distribution analysis and remapping analysis. Our object in this thesis was not to find techniques that are specific to some application domain and works efficiently for this domain only. We presented techniques that are based on sound intuitive ideas and works for a general program. We focused our research on checking legitimacy of integration of these ideas. We feel the need to just verify new aspects of our work. Also time constraint didn't allow us to experiment our tool with many applications. We mainly used five programs for our experiments.

- Following is an Alternating Direction Implicit (ADI) integration kernel.

```
Example 1 : ADI integration kernel:

  Real c(N, N), a(N, N), b(N, N)

  \\Read c, a, b

  Do iter = 1, max
  \\Forward and Backward sweep along rows
    Do j = 2, N
      DO i = 1, N
        c(i, j) = c(i, j) - c(i, j-1) * a(i, j) / b(i, j-1)
        b(i, j) = b(i, j) - a(i, j) * a(i, j) / b(i, j-1)
      ENDDO
    ENDDO

    DO i = 1, N
      c(i, N) = c(i, N) / b(i, N)
    ENDDO

    DO j = N-1, 1, -1
      DO i = 2, N
        c(i, j) = ( c(i,j) - a(i, j+1) * c(i, j+1)) / b(i, j)
      ENDDO
    ENDDO

    \\ Downward and Upward sweep along the columns
    Do j = 1, N
      DO i = 2, N
        c(i, j) = c(i, j) - c(i-1, j) * a(i, j) / b(i-1, j)
```

```
      b(i, j) = b(i, j) - a(i, j) * a(i, j) / b(i-1, j)
    ENDDO
  ENDDO

  DO i = 1, N
    c(N, i) = c(N, i) / b(N, i)
  ENDDO

  DO j = 2, N
    DO i = N-1, 1, -1
      c(i, j) = ( c(i,j) - a(i+1, j) * c(i+1, j)) / b(i, j)
    ENDDO
  ENDDO
```

ADI integration is a technique frequently used to solve partial differential equations (PDEs). ADI program remained in focus as target input of lots of previous automatic data layout approaches. We used ADI as our input target program. The execution of the ADI integration kernel consists of a repeated sequence of forward and backward sweeps along rows, followed by downward and upward sweeps along columns. For the sweeps along the rows, a row layout has the best performance and column layout has the best performance for the column sweeps. Transposing the arrays between the all row and all column sweeps eliminates communication within the sweeps. In contrast, choosing the same data layout for both, row and column sweeps will avoid communication between the sweeps but will make communication necessary either in the row or column sweeps. The best data layout choice will depend on the speed of the communication hardware and software of the target distributed-memory machine. The actual size N of the arrays and the number of available processors may influence the data layout choice.We applied our technique to Adi kernel for few test cases differing in program size and available number of processors. Following figure shows two different data layout specifications that is generated by our automatic data tool assistant tool.

`Generated output for ADI integration kernel:`

```
Case I : Static column-wise data layout generated for smaller N
  Real c(N, N), a(N, N), b(N, N)

  \\Static column-wise layout
  !HPF$ TEMPLATE X(N, N)
  !HPF$ ALIGN c(i, j), a(i, j), b(i, j) WITH X(i, j)
  !HPF$ DISTRIBUTE X(BLOCK, *)

    ........

  DO iter = 1, max
    \\Forward and Backward sweep along rows

     ......

    \\Downward and Upward sweep along columns


     .....

  ENDDO

Case II : Dynamic data layout generated for larger N
  Real c(N, N), a(N, N), b(N, N)

  !HPF$ TEMPLATE X(N, N)
  !HPF$ ALIGN c(i, j), a(i, j), b(i, j) WITH X(i, j)
  !HPF$ DISTRIBUTE X(BLOCK, *)

    ........
```

```
  DO iter = 1, max
    \\Forward and Backward sweep along rows

     ......

    \\Downward and Upward sweep along columns
    !HPF$ REDISTRIBUTE X(*, BLOCK)


     ......

  ENDDO
```

The first shows a static, column-wise data layout and generated for smaller values of N. The second depicts a dynamic data layout where transpose operations will be performed between the row and column sweeps is generated for larger values of N.

- Following is the another example program Jacobi.f

```
Example 2 : Jacobi.f

PROGRAM JACOBI
REAL A(500, 500), B(500, 500)
INTEGER I, J, K, TIME
TIME = 100

  DO 50 K = 1, TIME
    DO 20 J = 2, 499
      DO 10 I = 2, 499
        A(I, J) = ( B(I, J-1) + B(I-1, J) +B(I+1, J) + B(I, J+1)) / 4
10    CONTINUE
```

```
20    CONTINUE
    DO 40 J = 2, 499
      DO 30 I = 2, 499
        B(I, J) = A(I, J)
30    CONTINUE
40  CONTINUE
50 CONTINUE

END
```

Jacobi.f is an standard program. In every iteration an array element and all its four neighbours are accessed. Intuitively, (BLOCK, *), (*, BLOCK) and (BLOCK, BLOCK) are optimal distributions. Our tool generates (BLOCK, *) distribution for this code which is among the optimal one.

Generated output for Jacobi.f

```
  PROGRAM JACOBI
  REAL A(500, 500), B(500, 500)
  INTEGER I, J, K, TIME
  TIME = 100

  !HPF$ TEMPLATE X(500, 500)
  !HPF$ ALIGN A(I, J), B(I, J) WITH X(I, J)
  !HPF$ DISTRIBUTE X(BLOCK, *)

  //Code accessing all four neighbours of a element

  END
```

- Following is the another test program Mult.f

Example 3 : Matrix-Matrix Multiplication program

```
  PROGRAM MATMULT
  INTEGER A(300, 300), B(300, 300), C(300, 300)
  INTEGER I, J, K

//Initialize matrices

  DO 30 K = 1, 300
    DO 20 J = 1, 300
      DO 10 I = 1, 300
        C(I, J) = C(I, J) + A(I, K) * B(K, J)
10    CONTINUE
20   CONTINUE
30 CONTINUE
```

Mult.f is standard matrix multiplication program and used as kernel of many scientific applications. For the above permutation of the loops C and A are being accessed by column wise. Although B is being accessed by row wise but change in its indices are less frequent than A and C, and hence exploits temporal locality. Intuitively optimal distribution is (*, BLOCK) and that is produced by our tool.

Generated output for Mult.f:

```
  PROGRAM MATMULT
  INTEGER A(300, 300), B(300, 300), C(300, 300)
  INTEGER I, J, K

  !HPF$ TEMPLATE X(300, 300)
  !HPF$ ALIGN A(I, J), B(I, J), C(I, J) WITH X(I, J)
  !HPF$ DISTRIBUTE X(*, BLOCK)
```

```
  //Initialize matrices
  //Multiplication Computation
```

We also experimented our tool with a program to verify the capabiltiy of our distribution analyzer to generate BLOCK_CYCLIC distribution.

- Input program and its corresponding generated data layout (block-cyclic distribution) is shown in following figure.

Example 4 :Input Program to test BLOCK_CYCLIC feature :

```
  INTEGER TIMES, MAXELEMENT
  INTEGER iter, i

  MAXELEMENT = 18
  TIMES = 10

  REAL A(MAXELEMENT)

  REAL Stride = MAXELEMENT / 3

 DO iter = 1, TIMES
  DO i = 1, 11, 2
    A(i) = A(i+1) * A(i+Stride)
    A(i+1) = A(i) * A(i+Stride+1)
  ENDDO
```

Generated output :

```
  INTEGER TIMES, MAXELEMENT
```

```
  INTEGER iter, i

  MAXELEMENT = 18
  TIMES = 10

  REAL A(MAXELEMENT)

  REAL Stride = MAXELEMENT / 3



  !HPF$ TEMPLATE X(MAXELEMENT)
  !HPF$ ALIGN A(i) WITH X(i)
  !HPF$ PROCESSORS PROCS(3)
  !HPF$ DISTRIBUTE X(BLOCK_CYCLIC(2)) ONTO PROCS

 DO iter = 1, TIMES
  DO i = 1, 11, 2
    A(i) = A(i+1) * A(i+Stride)
    A(i+1) = A(i) * A(i+Stride+1)
  ENDDO
```

we also found the most optimal data layout for this program by solving on paper. We made a graph with nodes representing data elements of array A and edges show affinity between nodes. If two elements are accessed in one iteration instance of the loop then there is an edge between their corresponding nodes. We tried to find out different connected components of the graph. If we assign the data elements of different connected components of the graph to different processors it maximizes parallelism and minimizes communication. The graph and different connected components are shown in Fig. 7.1.

Connected components suggest that A(1), A(2), A(7), A(8) must be on one processor,A(3), A(4), A(9), A(10) must be on another processor,A(5), A(6), A(11), A(12)

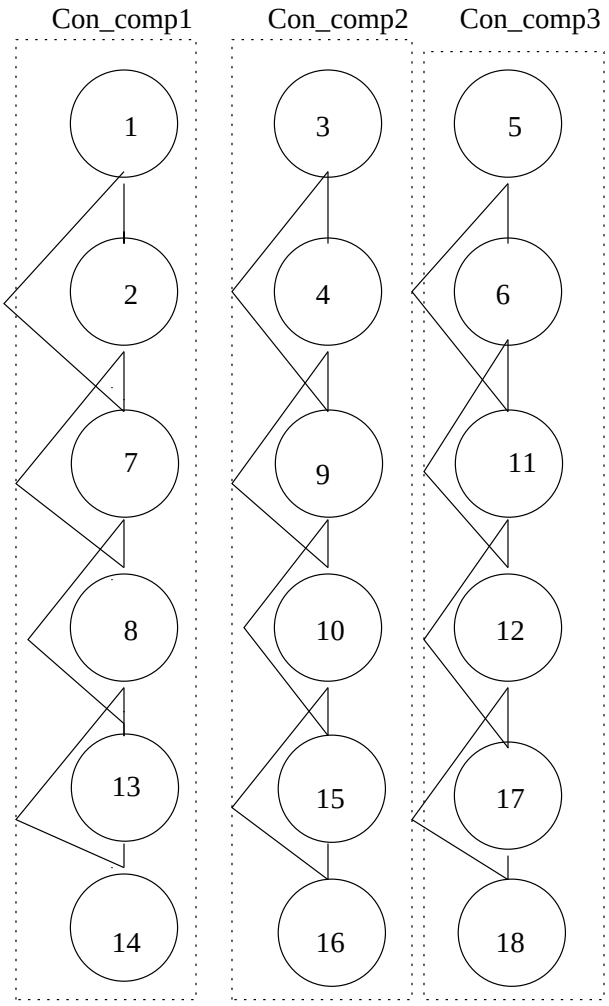


Figure 7.1: Sample program graph and connected components

must be on third processor. The tool generates data layout for the program that perfectly fulfills this requirement.

Another program is tested to verify that cyclic distribution is also being generated properly by the tool.

- Input program and its corresponding generated data layout (cyclic distribution) is shown in following figure.

Example 5 :Input Program to test CYCLIC feature:

```
  REAL A(13)

  DO i = 1, 10
    A(i) = A(i) + A(i+3)
  ENDDO
```

Generated output data layout:

```
  Real A(13)

  !HPF$ TEMPLATE X(13)
  !HPF$ ALIGN A(i) WITH x(i)
  !HPF$ PROCESSORS PROCS(3)
  1HPF$ DISTRIBUTE X(CYCLIC) ONTO PROCS

  DO i = 1, 10
    A(i) = A(i) + A(i+3)
  ENDDO
```

Intuitively, to minimize communication and maximize parallelism A(1), A(4), A(7), A(10), A(13) must reside on one processor, A(2), A(5), A(8), A(11) must reside on

another processor and A(3), A(6), A(9), A(12) must reside on third processor. The tool generates data layout for the program that exactly matches with our intuition.

These results are encouraging and show that our approach generates desired data layout.

Chapter 8

Conclusions and Future Work

8.1 Conclusion

Selection of good data layout with appropiate remapping while programming in data parallel languages is difficult for a general user. It requires knowledge about the target machine configuration, about the target compiler etc. We have developed a new framework for automatic data layout for regular problems. Our approach produces HPF like data layout for a given program and number of available processors. We also have implemented a prototype tool based on this approach. The prototype tool verified the claimed efficiency and quality of generated data layouts.

Our framework consists of two phases. The first phase is performed in two steps; the first step partitions program into program segments such that remapping between these program segments may be beneficial. We partitioned program based on program segment definition presented by Kremer [18]. In the second step data access patterns is genterated for each program segment. In the second phase, the controller produces the optimal data layout with appropriate remapping, for the program. Controller initially determines optimal data layout for each program segment and later tries to merge some program segments if remapping between them is not beneficial. The optimal data layout of a program part consist of appropiate alignment and distribution for arrays of that program part. To find proper alignment of arrays we used technique proposed by Li and Chen with slight refinement for more accurate results [21, 22]. To find distribution of

arrays we devised a constructive approach, that is strong enough to find out BLOCK, BLOCK_CYCLIC, CYCLIC distributions, supported by HPF. This new technique for distribution analysis is our main contribution. In the approach for distribution analysis we used intensively the technique proposed by Sarkar for program partitioning and scheduling for multiprocessors [26].

Our technique is general enough and to port automatic data layout tool from one machine to another only some machine parameter files need to be replaced. Presently we are trying to make it more general. Some efforts in this direction are; we can compute execution cost and remapping cost using abstract multiprocessor model instead of running it directly on machine. Also we expect this approach will work for irregular problems. The reason behind this claim is that, our technique considers data access pattern along with computation structure of the program.

8.2 Future Work

Many problems that are expected to be used as target input of automatic data layout tools, have irregular computation structure. Non-uniformity in data access patterns of these programs makes it difficult to extract needed information for automatic data layout selection, at compile-time. Intensive work has been done to use data parallel programming model for such problems. Almost all work done consider few problems and propose technique suitable for these problems only. A systematic general framework for such problems may be of significant importance.

Our current framework does not allow replication of data. One more phase can be added before the second phase. In this new phase we analyse data access pattern of program segments and find out read-only arrays. In the second phase, we can also consider replication of read-only arrays while constructing promising distribution candidate search space.

Technique proposed by us will work efficiently for unmodular programs or kernels of larger modular programs. To make this technique work efficiently for programs with many procedures, framework should be extended by including inter procedural analysis of data layout preferences. Call graph can be used to insert data layout preferences of called

procedure into the data layout preferences of callee procedure.

Distributed shared memory systems (DSM) are quite prevalent now-a-days. To develope automatic data layout technique for such system, unit of communication should be page or cache lines it should not be array element. In DSM system various communication optimization techniques like delayed update, has been implemented assuming release consistency model. All such issues should be considered while estimating performance of data layout for these systems.

Our distribution analyzer can be made more general such that it can adapt itself automatically for target compiler optimizations. An absract compiler model specification language can be made and these specifications are used by distribution analyzer. For some programs, set of data access patterns generated for program segments may be very large. Technique can be proposed to transform this set of data access patterns to a smaller set of data access patterns, such that the optimal data layout for new set will also be the optimal data layout for original set.

Bibliography

[1] Albert E., Knobe K. et.al. *Compiling Fortran 8x array features for the Connection Machine computer system.* Proceedings of the ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems (PPEALS), New Haven, CT, July 1988.

[2] Anderson J., Lam M. et.al. *Global optimizations for parallelism and locality on scalable parallel machines.* Proceedings of the SIGPLAN '93 Conference on Program Language Design and Implementation, Albuquerque, NM, June 1993.

[3] Ayguade E., Garcia J. et.al. *Detecting and using affinity in an automatic data distribution tool .* Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing, Ithaca, New York, August 1994.

[4] Bixby R., Kennedy K. et.al. *Automatic data layout using 0-1 integer programming.* Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT94), Montreal, Canada, August 1994.

[5] Chatterjee S., Gilbert J. et.al. *Automatic array alignment in data-parallel programs.* Proceedings of the Twentieth Annual ACM Symposium on the Principles of Programming Languages, Albuquerque, NM, January 1993.

[6] Chatterjee S., Gilbert J. et.al. *Optimal evaluation of array expressions on massively parallel machines.* Proceedings of the Second Workshop on Languages, Compilers, and Runtime Environments for Distributed Memory Multiprocessors, Bolder, CO, October 1992.

[7] Chen M., Choo Y. et.al. *Theory and pragmatics of compiling efficient parallel code.* Technical Report YALEU/DCS/TR-760, Dept. of Computer Science, Yale University, New Haven, CT, December 1989.

[8] Cormen T., Leiserson C. et.al. *Introduction to Algorithms.* The MIT Press, 1990.

[9] Garcia J., Ayguad'e E. et.al. *A novel approach towards automatic data distribution.* Proceedings of the Workshop on Automatic Data Layout and Performance Prediction (AP'95), Houston, TX, April 1995.

[10] Gupta M. and Banerjee P. et.al. *Automatic data partitioning on distributed memory multiprocessors.* Proceedings of the 6th Distributed Memory Computing Conference, Portland, OR, April 1991.

[11] Gupta M. and Banerjee P. et.al. *Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers.* IEEE Transactions on Parallel and Distributed Systems, April 1992.

[12] GuptaM. and Banerjee P. et.al. *Automatic data partitioning on distributed memory multiprocessors.* Technical Report CRHC-90-14, Center for Reliable and High-Performance Computing, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, October 1990.

[13] High Performance Fortran Forum. *High Performance Fortran language specification, version 1.0.* Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, May 1993.

[14] Hiranandani S., Kennedy K. et.al. *An overview of the Fortran D programming system.* Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing, Santa Clara, CA, August 1991.

[15] Knobe K., Lukas J. et.al. *Data optimization: Allocation of arrays to reduce communication on SIMD machines.* Journal of Parallel and Distributed Computing, 8(2):102-118, February 1990.

[16] Knuth D. *The art of compiler programming Volume 3: sorting and searching.* *Addision-Wesley*, 1973.

[17] Kremer U. *NP-completeness of dynamic remapping.* *Proceedings of the Fourth Workshop on Compilers for Parallel Computers, Delft, The Netherlands*, December 1993.

[18] Kremer U. *Automatic data layout for distributed-memory machines.* *Technical Report CRPC-TR93-299-S, Center for Research on Parallel Computation, Rice University*, February 1993.

[19] Kremer U., Mellor-Crummey J. et.al. *Requirements for data-parallel programming environments.* *IEEE Parallel and Distributed Technology*, 2(3):48-58, 1994.

[20] Kremer U., Zima H. et.al. *Advanced tools and techniques for automatic parallelization.* *Parallel Computing*, 7:387-393, 1988.

[21] Li J. and Chen M. *The data alignment phase in compiling programs for distributed-memory machines.* *Journal of Parallel and Distributed Computing*, 13(4):213-221, August 1991.

[22] Li J. and Chen M. *Compiling communication-efficient programs for massively parallel machines.* *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361-376, July 1991.

[23] Li J. and Chen M. et.al. *Index domain alignment: Minimizing cost of cross referencing between distributed arrays.* *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation, College Park, MD*, October 1990.

[24] Palermo D. and Banerjee P. *Automatic selection of dynamic data partitioning schemes for distributed-memory multicomputers.* *Technical Report CRHC95-09, Center for Reliable and High-Performance Computing, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign*, April 1995.

[25] Palermo D., Su E. et.al. *Communication optimizations used in the PARADIGM compiler for distributed-memory multicomputers.* *Proceedings of the 1994 International Conference on Parallel Processing, St. Charles, IL*, August 1994.

[26] Sarkar V. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* IBM Thomas J Watson research center, MIT press, Cambridge, Massatusetts.

[27] Tseng C. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* *PhD thesis, Rice University, Houston, TX*, January 1993.

[28] Wholey S. *Automatic data mapping for distributed-memory parallel computers.* *In Proceedings of the 1992 ACM International Conference on Supercomputing, Washington, DC*, July 1992.

[29] Wholey S. *Automatic Data Mapping for Distributed-Memory Parallel Computers.* *PhD thesis, School of Computer Science, Carnegie Mellon University*, May 1991.