A Combined Data and Program Partitioning Algorithm for Distributed Memory Multiprocessors

Tsuneo Nakanishi * Kazuki Joe †
Constantine D. Polychronopoulos ‡ Akira Fukuda *

narafrase@is.aist-nara.ac.jp

* Graduate School of Information Science, Nara Institute of Science and Technology
8916–5 Takayamacho, Ikoma, Nara 630–0101, JAPAN
† Wakayama University, Wakayama, JAPAN
‡ University of Illinois at Urbana-Champaign, IL, U.S.A.

Abstract

In this paper we propose an algorithm to perform data partitioning and program partitioning simultaneously on the Data Partitioning Graph, an intermediate representation for parallelizing compilers. Conventional and, therefore, conservative parallelizing compilers usually activate program partitioning prior to data partitioning. However, on distributed memory multiprocessors, since communication costs change depending on a data partitioning and distribution decision, it is quite difficult to partition a program effectively with consideration of data partitioning. The proposed algorithm resolves this confliction by handling these inseparable partitioning problems simultaneously with a branch-and-bound based scheme.

1 Introduction

Introduction of distributed memory multiprocessors, which promise high performance computation because of their scalability, increases the complexity of manual parallel programming tremendously. The most serious problem will be how to partition and distribute data of the program to distributed memory modules not to raise expensive interprocessor communications frequently. Many researchers have been exploring methods to reduce interprocessor communication overheads automatically by parallelizing compilers[5, 6, 7].

Parallelizing compilers partition a given program to tasks executed concurrently on target machines. This compilation process is often referred to as *program partitioning*, or *partitioning* simply. Syntactical objects of the source language such as statements, basic blocks, loops, or procedures, organize tasks themselves in general. However, since excessive partitioning fails in frequent expensive interprocessor communications and insufficient partitioning exploits poor parallelism[4], it is often required to optimize program partitioning by fusing or splitting tasks after initial program partitioning. Moreover, parallelizing compilers for distributed memory multiprocessors must optimize *data partitioning* of the program, namely partition and distribute array variables in an optimal form, unless partitioning or distribution of the array variables are specified at programmers' responsibility with some means such as source language properties[9], compiler directives, and so on.

We cannot either find an optimal program partitioning without communication costs between tasks fixed after data partitioning or optimize data partitioning without any program partitioning decisions. Sequential optimization of these partitioning follows conflicted program and data partitioning. Therefore, program partitioning and data partitioning should be optimized simultaneously in a unified manner.

So far parallelizing compilers have often employed the dependence graph for parallelization and code optimization as a simple and convenient intermediate representation. However, the dependence

graph is not powerful enough for distributed memory multiprocessors, since the dependence graph is lack of explicit information on data locations and accesses which is essential to optimize data partitioning.

In this paper we define the Data Partitioning Graph (DPG), an extension of the dependence graph with explicit data location and access information, as a common intermediate representation of a given program for data partitioning and transferring optimization techniques. Furthermore, we propose the CDP^{2}Algorithm to optimize program partitioning and data partitioning simultaneously with the DPG.

2 Data Partitioning Graph

We mentioned the dependence graph has constitutional problems as an intermediate representation for parallelizing compilers whose targets are distributed memory multiprocessors, namely the dependence graph lack explicit data location and access information, in Chapter 1. We define the Data Partitioning Graph, or the DPG, as an intermediate representation which reveals variables and accesses to them in its structure.

Figure 1 shows an example of the DPG. The DPG has two kinds of nodes: *C-nodes* shown as circular nodes and *D-nodes* shown as square nodes.

The C-nodes represent tasks. There are two kinds of *dependence edges*, that is, the *control dependence edge* and the *data dependence edge*, between C-nodes. A control dependence edge or a data dependence edge represents a control dependence[2] or a data dependence from the task corresponding to the source of the edge to the task corresponding to the sink of the edge.

The D-nodes represent sets of scalar variables and array variable elements. Note that we will refer to a scalar variable or an array variable element as *variable* simply in the rest of this paper. We partition the set of all the variables into classes according to their access patterns and generate a D-node for each class of variables, namely a set of variables whose access patterns are same. The access pattern is a property of the variable indicates which tasks read the variable at what frequency and which tasks write the variable at what frequency. If the task of a C-node may read a variable in the class of a D-node, we set a *read access edge* from the D-node to the C-node. Similarly, if the task of a C-node may write a variable in the class of a D-node, we set a *write access edge* from the C-node to the D-node. We often refer to read access edges or write access edges as *data access edges* without distinction.

In this paper we denote the DPG by a tuple $(\{CV, DV\}, \{DDE, CDE, RAE, WAE\})$, where CV, DV, CDE, DDE, RAE, and WAE are a set of C-nodes, a set of D-nodes, a set of control dependence edges, a set of data dependence edges, a set of read access edges, and a set of write access edges respectively for formal description.

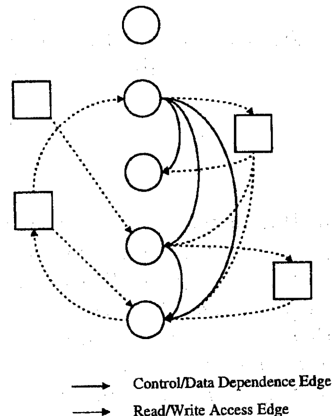

Figure 1: Data Partitioning Graph (DPG)

3 Data-Program Partitioning

3.1 Assumptions

In this paper we assume that parallel programs are executed on a distributed memory multiprocessor which satisfies the following requirements: i) Processors are provided as many as the executing parallel program requires; ii) Each processor owns its local memory module to be accessible with no latency; iii) Each processor can access remote memory modules with some latency to be independent of the locations of the remote memory modules.

Also we assume that all tasks are assigned to processors by an appropriate static scheduling algorithm[1] and each processor executes its assigned tasks in the following manner: i) Choose a task being ready to run; ii) Read values of variables on remote memory modules required for the

task execution into copies of the variables on the local memory module; iii) Execute the task non-preemptively; iv) Write values of the copied variables back to their original on remote memory modules; v) Go to Step i). In this execution model variables on remote memory modules are accessed collectively before and after each task execution, while variables on the local memory module are accessed at any time.

For portion of the program to be partitioned optimally we set the following assumptions: i) The dependence graph of the portion is acyclic; ii) There is no control dependence between tasks. The second assumption states there is no branch in the portion and all the task of the portion are executed.

3.2 Data-Program Partitioning

We achieve optimization of both program partitioning and data partitioning simultaneously, or *data-program partitioning*, by fusing C-nodes and D-nodes of the DPG. The data-program partitioning problem is formalized as a problem of grouping C-nodes and D-nodes of the DPG. The tasks of C-nodes in each group are fused, moreover, the variables of D-nodes in each group are located at the local memory module of the processor which executes the task constructed by fusing the tasks of the C-nodes in the same group. Therefore, data access edges whose sources and sinks are in different groups represent accesses to remote memory modules.

Data-program partitioning on the DPG reproduces a new dependence graph from the original DPG. At first C-nodes and D-nodes in the same group are fused. Self-loop data dependence edges are removed and parallel data dependence edges are bundled. The DPG has execution times of tasks as the costs of their corresponding C-nodes. Besides each data access edge has time of the interprocessor communication required for the corresponding data access as its cost. Data access edges are removed but their costs are used to compute edge costs of the new dependence graph with considering whether the data access edges represent accesses to remote memory modules or the local memory module. In this way costs of nodes and edges of the new dependence graph are computed. It is known that the cost of the critical path of the dependence graph, whose nodes and edges have execution times of the corresponding tasks and times required for the corresponding communications, gives minimum parallel execution time in the field of scheduling. Thus the critical path cost of the dependence graph derived from the DPG by the CDP^{2}Algorithm gives minimum parallel execution time under the program and data partitioning decision defined by the grouping of C-nodes and D-nodes. We can formulate the data-program partitioning problem as a problem of finding the *optimal* grouping of C-nods and D-nodes of the DPG, namely the grouping such that minimizes the critical path cost of the resulting dependence graph.

We must respect the *convex constraint*[4], which was originally defined for program partitioning on the dependence graph, on grouping C-nodes to avoid dead locks. The convex constraint is redefined for the DPG as a constraint on grouping C-nodes of the DPG such that for any two different C-nodes cv_u and cv_v in each group all the nodes in all the paths from cv_u to cv_v consisting of only data dependence edges must be in the same group. Data-program partitioning without the convex constraint produces a cyclic dependence graph which causes a dead lock.

4 The CDP^{2}Algorithm

The CDP^{2}Algorithm, which is extended from Girkar's program partitioning algorithm[3] on the dependence graph, searches for the optimal grouping of nodes of the DPG by a branch-and-bound based scheme.

Girkar's partitioning algorithm performs grouping of nodes of the dependence graph to optimize program partitioning. For a grouping of the nodes of the dependence graph, if nodes of each group and edges between the nodes are organizing a connected subgraph of the dependence graph, we refer to the partitioning given by the grouping as *connected.* Girkar proved the following theorem concerning optimality and connectivity of partitioning on the dependence graph in [3].

Theorem 1 *There exists a connected partitioning.* □

Girkar classifies all the edges of the dependence graph into a class Π or a class π to define a grouping of the nodes of the dependence graph gives a connected partitioning in his algorithm. The subgraph consisting of nodes of the dependence graph

and edges in π contains connected components. Girkar's algorithm regards the nodes of each of the connected components as to be in the same group to define a grouping of the nodes of the dependence graph gives a connected partitioning. The problem is to find a classification of edges of the dependence graph into Π and π which minimizes the critical path cost of the dependence graph produced by grain packing defined by the classification. We extend his idea to the data-program partitioning on the DPG.

4.1 Properties of the DPG

The CDP^{2}Algorithm accepts the DPG of a given program, performs grouping of C-nodes and D-nodes of the DPG to decide which tasks are fused and where variables are located, and produces a dependence graph of the program optimized its partitioning and data partitioning based on the aforementioned formalization in the previous chapter.

The CDP^{2}Algorithm classifies all the data dependence edges of a given DPG into a class Π and a class π in a similar way of Girkar's program partitioning algorithm. Simultaneously the CDP^{2}Algorithm classifies all the read access edges of the DPG into a class Π_{RAE} and a class π_{RAE} and all the write access edges of the DPG into a class Π_{WAE} and a class π_{WAE}. A C-node and a D-node which are the end points of each data access edge in π_{RAE} and π_{WAE} are enclosed in the same group. To the contrary, C-nodes and D-nodes corresponding to the end points of data access edges in Π_{RAE} or Π_{WAE} are in different groups. Thus data accesses corresponding to the data access edges in Π_{RAE} and Π_{WAE} raise interprocessor communications. Figure 2 describes the basic idea of the CDP^{2}Algorithm.

The CDP^{2}Algorithm must classify data access edges not to conflict with the classification of the data dependence edges. Two properties of the described below are essential to guarantee conflictless classification.

Firstly, for any data dependence edges there exists a D-node which contain variables concerning with the dependence of the edge. Figure 3 explains that. We denote a set of D-nodes which contain variables concerning with the dependence of any data dependence edge (cv_u, cv_v) by $DV_f((cv_u, cv_v))$, $DV_o((cv_u, cv_v))$, or $DV_a((cv_u, cv_v))$ in case (cv_u, cv_v) represents a flow

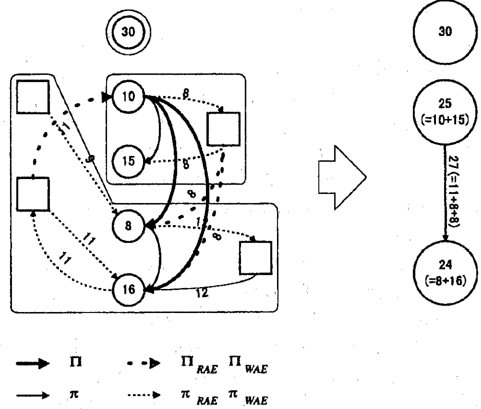

Figure 2: The Basic Idea of the CDP^{2}Algorithm

dependence, an output dependence, or an anti-dependence respectively.

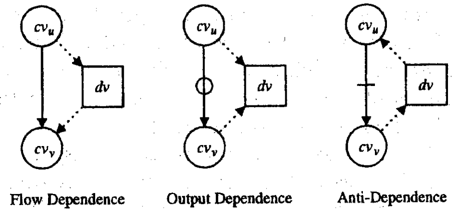

Figure 3: Dependence Edges and D-nodes

Secondly, if any data dependence edge (cv_u, cv_v) is classified into Π, there exists three legal cases of classification of data access edges between $dv \in DV_f((cv_u, cv_v)) \cup DV_o((cv_u, cv_v)) \cup DV_a((cv_u, cv_v))$ and each of cv_u and cv_v. In case of (cv_u, cv_v) is classified into π, the legal classification is two cases. Figure 4 enumerates these legal classifications.

Given a conflictless classification of data dependence edges and data access edges we can define a dependence graph as described in Section 3.2. We have to recompute the costs of nodes and edges of the dependence graph to evaluate minimum parallel execution time at program and data partitioning given by the classification. The cost of a node of the dependence graph is the total cost of C-nodes fused on constructing the node. On the other hand the cost of a data dependence edge (cv_u, cv_v) denoted by $\omega_{DDE}((cv_u, cv_v))$ is given as follows based on the

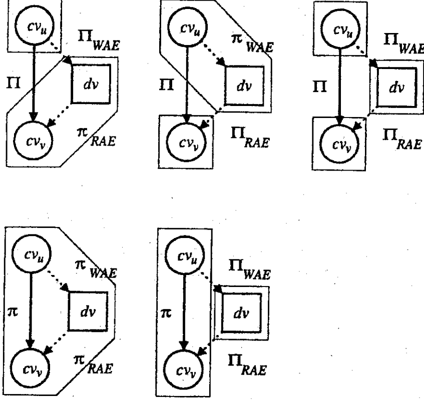

Figure 4: Conflictless Classification

classification of data access edges. In the following expression the cost of a read access edge (dv, cv) and the cost of a write access edge (cv, dv) are denoted by $\omega_{RAE}((cv_u, dv))$ and $\omega_{WAE}((cv, dv))$ respectively.

$$
\begin{aligned}
&\omega_{DDE}((cv_u, cv_v)) \\
= &\sum_{dv \in \{dv' \in DV_f((cv_u, cv_v)) | (cv_u, dv') \in \Pi_{WAE}\}} \omega_{WAE}((cv_u, dv)) \\
+ &\sum_{dv \in \{dv' \in DV_f((cv_u, cv_v)) | (dv', cv_v) \in \Pi_{RAE}\}} \omega_{RAE}((dv, cv_v)) \\
+ &\sum_{dv \in \{dv' \in DV_o((cv_u, cv_v)) | (cv_u, dv') \in \Pi_{WAE}\}} \omega_{WAE}((cv_u, dv)) \\
+ &\sum_{dv \in \{dv' \in DV_o((cv_u, cv_v)) | (cv_v, dv') \in \Pi_{WAE}\}} \omega_{WAE}((cv_v, dv)) \\
+ &\sum_{dv \in \{dv' \in DV_a((cv_u, cv_v)) | (dv', cv_u) \in \Pi_{RAE}\}} \omega_{RAE}((dv, cv_u)) \\
+ &\sum_{dv \in \{dv' \in DV_a((cv_u, cv_v)) | (cv_v, dv') \in \Pi_{WAE}\}} \omega_{WAE}((cv_v, dv))
\end{aligned}
$$

Note that we assume zero latency for local memory module accesses and the costs of the only data access edges in Π_{RAE} and Π_{WAE} contribute the costs of data dependence edges. The costs of data access edges are assigned times required for interprocessor communication of the corresponding data accesses as described in Section 3.2.

4.2 The CDP^{2}Algorithm

The CDP^{2}Algorithm employs a branch-and-bound scheme to find a classification of data dependence edges and data access edges which minimizes the critical path cost of the dependence graph given by the classification. At each step the CDP^{2}Algorithm chooses a halfway classification, picks a data dependence edge of unknown class, and derives two cases of new halfway classification, namely cases when the picked data dependence edge is classified into Π and π. Moreover, for each cases the CDP^{2}Algorithm derives classifications of the related data access edges not to conflict with the current halfway classification of data dependence edges and data access edges based on the property shown in Figure 4. Of course, since it requires exponential computation time to examine all the possible classifications, the CDP^{2}Algorithm produces derived classifications only from prospective classifications which will generate the optimal classification later and suspends the other halfway classifications at each step.

The CDP^{2}Algorithm defines the halfway classification, or the *incumbent* of the solution, as a septuplet:

$$(\rho, \chi, \rho_{RAE}, \chi_{RAE}, \rho_{WAE}, \chi_{WAE}, \omega).$$

Here ρ are a set of data dependence edges classified into π and χ are a set of data dependence edges which are classified into neither Π nor π yet. ρ_{RAE}, χ_{RAE}, ρ_{WAE}, and χ_{WAE} are similar sets but of data access edges. ω is a critical path cost of the dependence graph under the program and data partitioning decision given by this current edge classification. Edges in χ, χ_{RAE}, and χ_{WAE} are dealt as if they were not in the DPG on computing the critical path cost. The CDP^{2}Algorithm picks an incumbent with minimum ω on derivations of incumbents. Since derived incumbents never have less ω than their original incumbent, the first found incumbent such that there is no data dependence edge in χ must be an edge classification which defines optimal program and data partitioning.

We omit the detail flow of the CDP^{2}Algorithm since we do not have enough space to describe it. See [8] instead. The CDP^{2}Algorithm itself is a well-known branch-and-bound algorithm but we impose a restriction on selection of data dependence edge at each step to keep the convex constraint. The CDP^{2}Algorithm does not apply active bounding

but only suspends inexpectant incumbents since we do not have confident ideas on effective bounding schemes to the DPGs of real applications.

Some data access edges may be left unclassified and some D-nodes may not be fused after an application of the CDP^2 Algorithm. Read access edges from D-nodes whose variables are read by some tasks but not written by any tasks are left unclassified, since the variables never relate to any data dependencies. Write access edges to D-nodes whose variables are written by one task but not read by any tasks are also left unclassified for the same reason. We should duplicate those variables pro re nata and locate variables or distribute their copies over local memory modules of the processors which execute tasks referring the variables. This process keeps data references of these data access edges from raising interprocessor communications. The variables of isolated D-nodes should be removed, since they are redundant variables to be read or written by any tasks.

5 Conclusion

In this paper we formalized the data-program partitioning problem, a problem to partition a given program and its data in an optimal form, as a problem grouping nodes of the DPG which is an extension of the dependence graph with explicit data location and access information. The CDP^2 Algorithm described in this paper is a branch-and-bound based algorithm which solves the data-program partitioning problem.

The current CDP^2 Algorithm does not bound hopeless incumbents actively but only suspends incumbents of unknown expectancy at each step. Since computation time of a branch-and-bound algorithm depends on how the algorithm bounds hopeless incumbents effectively in earlier steps, it will be our prior future work to find an effective bounding method for the CDP^2 Algorithm. Although the current CDP^2 Algorithm picks data dependence edges in an arbitrary order, picking data dependence edges in a sophisticated order will contribute to reduce computation time of the CDP^2 Algorithm by suppressing the number of branchings. It is also a future work to develop a heuristics on the order of picking data dependence edges to suppress the number of branchings. For both future works we consider utilizing evaluation measures used in list scheduling algorithms[1] such as the critical path cost to the bottom of the dependence graph, the number of children or descendants of each node of the dependence graph, and so on.

References

[1] H. Kasahara and S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," *IEEE Trans. on Computers*, Vol.C-33, No.11, pp.1023–1029, Nov. 1984.

[2] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and Its Use in Optimization," ACM Trans. on Programming Languages and Systems, Vol.9, No.3, pp.319–349, Jul. 1987.

[3] M. B. Girkar and C. D. Polychronopoulos, "Partitioning Programs for Parallel Execution," *Proc. of the 1988 Int. Conf. on Supercomputing*, pp.216–229, 1988.

[4] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*, The MIT Press, 1989.

[5] J. Li and M. Chen, "Index Domain Alignment: Minimizing Cost of Cross-Referencing between Distributed Arrays," *Proc. Frontier '90: The 3rd Symp. on the Frontiers of Massively Parallel Computation*, pp.424–433, Oct. 1990.

[6] M. Gupta and P. Banerjee, "Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputer," *IEEE Trans. on Parallel and Distributed Systems*, Vol.3, No.2, pp.179–193, Mar. 1992.

[7] P. Tu and D. Padua, "Automatic Array Privatization," *Proc. of the 6th Int. Workshop on Languages and Compilers for Parallel Computing*, pp.500–521, 1993.

[8] T. Nakanishi, K. Joe, H. Saito, A. Fukuda, and K. Araki, "The CDP^2 Algorithm: A Combined Data and Program Partitioning Algorithm on the Data Partitioning Graph," *Proc. of the 1995 Int. Conf. on Parallel Processing*, Vol.II, pp.177–181, Aug. 1995.

[9] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr., and M. E. Zosel, *The High Performance Fortran Handbook*, The MIT Press, 1994.