Vol. 40 No. SIG 9 (TOM 2) 情報処理学会論文誌 Dec. 1999

Regular Paper

A Combined Data and Program Partitioning Algorithm for Distributed Memory Multiprocessors

Tsuneo Nakanishi ,† Kazuki Joe ,††
Constantine D. Polychronopoulos ††† and Akira Fukuda†

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

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 compilers5),9),12).

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 parallelism11), 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 properties8), 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. Conflicted program and data partitioning will follow sequential optimization of these 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 propose the CDP^2 Algorithm to optimize program partitioning and data partitioning simultaneously with an extension of the dependence graph representing explicit data location and access information which we have proposed, the Data Partitioning Graph (DPG)10), as a common intermediate representation of a given program for data partitioning and transferring optimization techniques. The chief aim of this paper is to introduce an algorithm which optimizes program partitioning and data partitioning at the same time, there-

---

† Graduate School of Information Science, Nara Institute of Science and Technology
†† Faculty of Science, Nara Women's University
††† Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign

fore, we do not discuss implementation and application of the algorithm.

The rest of this paper is organized as follows. Chapter 2 gives brief introduction to the DPG. Chapter 3 formalizes the data-program partitioning problem on the DPG. Chapter 4 describes fundamental properties of the DPG and details of the CDP^{2}Algorithm utilizing those properties. Finally Chapter 5 concludes this paper.

2. Data Partitioning Graph (DPG)

In this chapter we summarize the dependence graph and its constitutional difficulties as an intermediate representation of the program executed on distributed memory multiprocessors. Moreover, we introduce an extension of the dependence graph which resolves those difficulties, the Data Partitioning Graph (DPG)$^{10)}$, as an intermediate representation for parallelizing compilers.

2.1 Dependence Graph

Parallelizing compilers build and transform the intermediate representation of a given program to manage analytic information, detect and exploit parallelism, and generate a parallel program which is semantically equivalent to the source program. The simplest and well-known intermediate representation will be the dependence graph which has been employed by parallelizing (or vectorizing) compilers since the dawn of the parallelizing compiler. The node of the dependence graph represents a task. The edge from a node u to a node v of the dependence graph, denoted by a tuple (u, v), represents that the task of v cannot begin its execution before completion of the task of u, that is, the dependence from the task of u to the task of v.

Advantageous features of the dependence graph are its simplicity and commonality. The dependence graph or its extensions$^{2),3)}$ are useful for many effective parallelization and code optimization techniques employed by parallelizing compilers$^{1)}$.

Although the dependence graph provides essential information for semantically correct parallelization and code optimization, it does not necessarily follow that the dependence graph provides information to achieve good speedup. Naive parallelization often fails in frequent interprocessor communications which should be avoided especially on distributed memory multiprocessors. It is a natural and possible approach to attach costs of communications raised by dependencies to the corresponding edges for dealing with problems of communication overheads on the dependence graph. However, the dependence graph represents no explicit information about location of variables which affects communication overheads extremely on distributed memory multiprocessors.

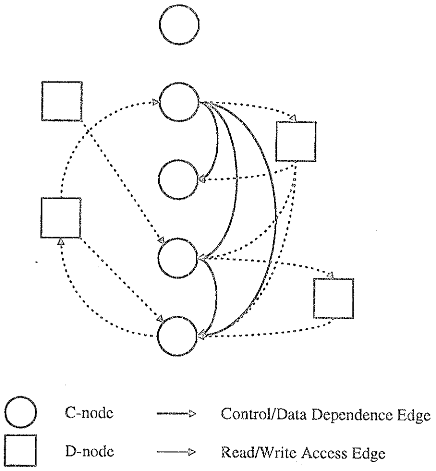


Fig. 1 Data Partitioning Graph (DPG)

2.2 Data Partitioning Graph (DPG)

Based on observation in the previous section, we have defined the Data Partitioning Graph, or the DPG, as an intermediate representation which reveals variables and accesses to them in its structure in reference 10).

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-node represents a task. 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 and a data dependence edge represent a control dependence$^{2)}$ and a data dependence from the task corresponding to the source of the edge to the task corresponding to the sink of the edge respectively. Note that the DPG contains the dependence graph in itself.

The D-node represents a set 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 access patterns of the variables and generate a D-node for each class of variables, namely a set of the variables whose access patterns are same. The access pattern is a property of the variable indicating 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.

3. Data-Program Partitioning

In this chapter we put assumptions concerning architectures on which run parallel programs, task execution model, and programs to be partitioned optimally with their data. Moreover, we formalize the data-program partitioning problem on the DPG under these assumptions.

3.1 Assumptions

In this paper we assume that parallel programs are executed on a distributed memory multiprocessor which satisfies the following requirements.

(1) Processors are provided as many as the executing parallel program requires.
(2) Each processor owns its local memory module to be accessible with no latency.
(3) 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 algorithm6),7) and each processor executes its own assigned tasks in the following manner.

(1) Choose a task being ready to run.
(2) Read the values of the variables on remote memory modules required for the task execution into the copies of the variables on the local memory module.
(3) Execute the task non-preemptively.
(4) Write the values of the copied variables back to their original on remote memory modules.
(5) Go to Step 1.

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 put the following assumptions.

(1) The dependence graph of the portion is acyclic.
(2) There is no control dependence between tasks.

The second assumption states there is no branch in the portion and all the tasks of the portion are executed.

3.2 Grain Packing

Grain packing is one of well-known methods to optimize program partitioning by fusing tasks of finer granularity. Grain packing reduces communication overheads since data exchanges between fused tasks never cause interprocessor communications. However, grain packing spoils parallelism at the same time in case some of fused tasks can run in parallel.

We can formalize grain packing as a problem of grouping nodes of the dependence graph and fusing the tasks corresponding to the nodes in each group. Under this formalization grain packing reproduces a new dependence graph from the original dependence graph. At first the nodes in each group are fused. Edges of the original dependence graph whose sources and sinks are in the same group appear as self-loops at this moment. These self-loop edges are removed. The remained edges are ones of the original dependence graph whose sources and sinks are in different groups. Some of them may be parallel, namely their sources and sinks are identical respectively. These parallel edges are bundled into one edge.

We have often assigned execution time of each task to the corresponding node of the dependence graph and time required for the interprocessor communication raised by each dependence to the corresponding edge as their costs for task scheduling. These costs should be recomputed for the new dependence graph. The cost of each node of the new dependence graph is the total cost of the nodes of the original dependence graph fused on grain packing. Similarly, the cost of each edge of the new dependence graph is the total cost of the edges of the original dependence graph bundled on grain packing. It is known the cost of the critical path of a dependence graph, the path such that the total cost of the nodes and the edges in itself is not less than any other paths in the dependence graph, gives the minimum parallel execution time required to complete all the tasks. Therefore, we can formulate grain packing as a

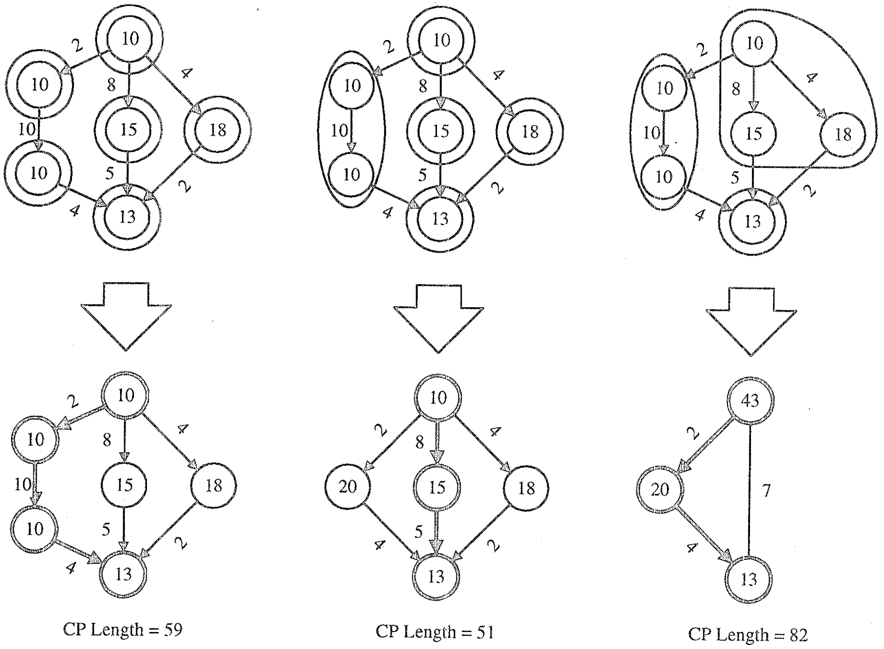


Fig. 2 Grain Packing

problem of grouping nodes of the dependence graph to minimize the critical path cost of the resulting dependence graph.

Figure 2 shows three examples of grain packing. The first example makes no fusing of nodes. The second example fusing two nodes reduces the critical path cost of the resulting dependence graph than the first example owing to elimination of interprocessor communication by grain packing. However, the last example fusing three plus two nodes turns the result unfavorable because of loss of parallelism.

To avoid dead locks under our task execution model grain packing must respect a constraint on grouping nodes of the dependence graph such that for any two different nodes u and v in each group all the nodes in all the paths from u to v must be in the same group, the *convex constraint*$^{11)}$. Grain packing without the convex constraint reproduces a cyclic dependence graph including a loop in itself and causes a dead lock among the tasks of the nodes in the loop.

3.3 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 likewise for program partitioning on the dependence graph. The tasks of the C-nodes in each group are fused, moreover, the variables of the 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 the C-nodes and the D-nodes in the same group are fused. Self-loop data dependence edges are removed and parallel data dependence edges are bundled likewise for grain packing. The DPG has execution times of tasks as the costs of their corresponding C-nodes. The DPG has communication times as costs of data access edges instead of data dependence edges. 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 and the critical path cost of the dependence graph 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-nodes and D-nodes of the DPG, namely the grouping such that minimizes the critical path cost of the resulting dependence graph.

The convex constraint must be respected on data-program partitioning 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$^{4)}$ on the dependence graph, is an A* algorithm☆ which searches for the optimal grouping of nodes of the DPG. In this chapter we make brief introduction of the A* algorithm in a generalized form and rewrite Girkar's program partitioning algorithm as an A* algorithm. Furthermore, we present fundamental properties of the DPG to derive the CDP^{2}Algorithm and describe details of the CDP^{2}Algorithm.

Through this chapter we denote the dependence graph, which is the input of Girkar's program partitioning algorithm, by a tuple (V, E) where V and E are a set of the nodes and a set of the edges respectively. Similarly we denote the DPG, which is the input of the CDP^{2}Algorithm, by a tuple $(\{CV, DV\}, \{DDE, CDE, RAE, WAE\})$, where CV, DV, CDE, DDE, RAE, and WAE are a set of the C-nodes, a set of the D-nodes, a set of the control dependence edges, a set of the data dependence edges, a set of the read access edges, and a set of the write access edges respectively.

4.1 The A* Algorithm

The A* algorithm is an algorithm which searches state space such that state transition in the space is represented by a directed graph. Each state has its own evaluation value equal to or greater than those of the previous states. The A* algorithm finds out the final state whose evaluation value is minimum. Algorithm 1 describes the A* algorithm in a generalized form.

Algorithm 1 (Generalized A* Algorithm)

(1) Insert the initial state to a list λ.
(2) Remove the state s whose evaluation value is minimum from λ.
(3) Terminate the algorithm if s is one of final states.
(4) Insert all the states derived immediately from s to λ with their evaluation values.
(5) Go to Step 2. □

As shown in Algorithm 1, the A* algorithm is characterized by i)the definition of the state, ii)the initial state, iii)the final states, iv)the state transition procedure, and v)the definition of the evaluation value of the state. They are defined according to the problem to be solved with the A* algorithm.

The A* algorithm can use a biased state evaluation value $f = g + h$, where f is an actual evaluation value and h is a non-negative bias, for each state. h must be a lower bound of the differences of the evaluation values of the reachable final states from the current state evaluation value. Larger h reduces the number of states to be examined and contributes to reduce computation time of the algorithm. Girkar does not discuss biasing state evaluation values of his program partitioning algorithm in reference 4). We also do not bias state evaluation values of the CDP^{2}Algorithm in this paper to keep discussion simple. But both algorithms can improve their performance by biasing state evaluation values effectively.

4.2 Girkar's Program Partitioning Algorithm

For a grouping of the nodes of the dependence graph, if the nodes of each group and the edges between these 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 reference

---

☆ Girkar refers to his partitioning algorithm as a branch-and-bound algorithm in reference 4), however, his algorithm applies no bounding operations. As written in this paper, the A* algorithm will be a more proper category of Girkar's algorithm and our CDP^{2}Algorithm rather than the branch-and-bound algorithm.

4).

Theorem 1 There exists a connected and optimal partitioning. □

Girkar's program partitioning algorithm accepts a dependence graph of the program to be partitioned and finds a grouping of its nodes which gives the optimal partitioning of the program out of groupings of its nodes which give connected partitionings. Girkar's algorithm classifies all the edges of the dependence graph into a class Π or a class π. Girkar's algorithm regards the end point nodes of each edge in π as to be in the same group and end point nodes of each edge in Π as to be in different groups. There exist at least one or more connected components in the subgraph consisting of the nodes of the dependence graph and the edges in π. These connected components can define a grouping of the nodes of the dependence graph, that is, the nodes in each of the connected components organize one group of the grouping. Obviously the partitioning given by the grouping is connected. Therefore, the problem is reduced to find a classification of the edges of the dependence graph into Π and π which minimizes the critical path cost of the dependence graph reproduced by the grain packing which the edge classification defines. Girkar characterizes the A* algorithm as follows to find such an edge classification.

- **The Definition of the State.** Girkar's algorithm defines its state as a tuple (ρ, χ) where ρ is a set of the edges classified into π and χ is a set of the edges classified into neither Π nor π yet. Let P be the set of the edges classified into Π, i.e. $E - (\rho \cup \chi)$.
- **The Initial State.** Girkar's algorithm defines the initial state of the problem as state $(\emptyset, E)$. At the initial state no edge is classified into Π and π.
- **The Final States.** A state (ρ, χ) is a final one if $\chi = \emptyset$. At final states all the edge is classified into Π or π.
- **The State Transition Procedure.** Let us suppose Girkar's algorithm removes a state (ρ, χ) from the list λ. Girkar's algorithm picks an arbitrary edge $e = (u, v)$ in χ and derives two new states: $(\rho, \chi - \{e\})$ and $(\rho \cup B, \chi - B)$ where B is a set of the edges of the dependence graph in all the paths from u to v. The new state $(\rho, \chi - \{e\})$ is always inserted to λ. However, the new state $(\rho \cup B, \chi - B)$ is not inserted to λ if there exists an edge (a, b) such that $(a, b) \in B \cap P$. Note that the state transition which this procedure defines is represented by a directed tree.
- **The Definition of the State Evaluation Value.** The evaluation value of a state (ρ, χ) is the critical path cost of the dependence graph reproduced by the halfway edge classification which (ρ, χ) defines. Edges in χ are dealt as if they did not exist in the dependence graph on computation of the critical path cost. States derived by the above transition procedure never have less evaluation values than their original state since the dependence graphs reproduced by the derived states have more edges, which can contribute to increase the critical path cost, than that of their original state.

The final state obtained by Girkar's algorithm gives the edge classification which partitions a given program optimally. The complexity of the algorithm is exponential in the worst case.

4.3 Fundamental 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 or a class π in a similar way to Girkar's program partitioning algorithm. Simultaneously the CDP^{2}Algorithm classifies all the read access edges of the DPG into a class Π_{RAE} or a class π_{RAE} and all the write access edges of the DPG into a class Π_{WAE} or a class π_{WAE}. The C-node and the 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, the C-node and the D-node which are the end points of each data access edge in Π_{RAE} or Π_{WAE} are in different groups. Thus data accesses corresponding to the data access edges in Π_{RAE} or Π_{WAE} raise interprocessor communications. **Figure 3** 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. We show two

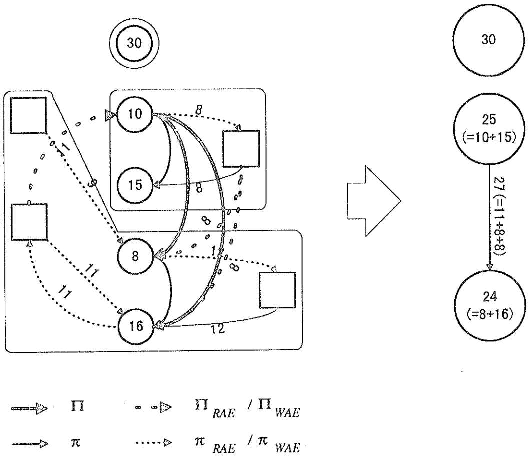


Fig. 3 The Basic Idea of the CDP^2 Algorithm

fundamental properties of the DPG to guarantee conflictless classification of the data access edges below. The first property states for any data dependence edge there exist D-nodes which contain variables concerning its corresponding dependence.

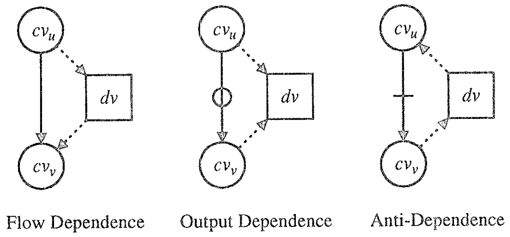
Property 1 For any data dependence edge $dde = (cv_u, cv_v) \in DDE$ of a given DPG,

- If dde represents a flow dependence from the task of cv_u to the task of cv_v, there exists a D-node $dv \in DV$ such that $(cv_u, dv) \in WAE$ and $(dv, cv_v) \in RAE$.
- If dde represents an output dependence from the task of cv_u to the task of cv_v, there exists a D-node $dv \in DV$ such that $(cv_u, dv) \in WAE$ and $(cv_v, dv) \in WAE$.
- If dde represents an anti-dependence from the task of cv_u to the task of cv_v, there exists a D-node $dv \in DV$ such that $(dv, cv_u) \in RAE$ and $(cv_v, dv) \in WAE$.

(It is trivial according to definitions of these dependencies. See **Fig. 4** instead of the proof.) □

We denote a set of the D-nodes which contain variables concerning with the dependence of any data dependence edge (cv_u, cv_v) by $DV_f((cv_u, cv_v))$, $DV_o((cv_u, cv_v))$, and $DV_a((cv_u, cv_v))$ in case (cv_u, cv_v) represents a flow dependence, an output dependence, and an anti-dependence respectively. $DV_f((cv_u, cv_v))$, $DV_o((cv_u, cv_v))$, and $DV_a((cv_u, cv_v))$ are defined as follows in formal.

$$DV_f((cv_u, cv_v)) = \{dv \in DV | \\ (cv_u, dv) \in WAE, \\ (dv, cv_v) \in RAE\}$$



Flow Dependence Output Dependence Anti-Dependence

Fig. 4 Dependence Edges and D-nodes

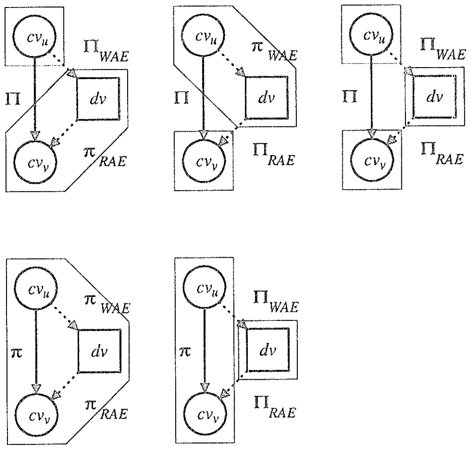


Fig. 5 Conflictless Classifications

$$DV_o((cv_u, cv_v)) = \{dv \in DV | \\ (cv_u, dv) \in WAE, \\ (cv_v, dv) \in WAE\}$$

$$DV_a((cv_u, cv_v)) = \{dv \in DV | \\ (dv, cv_u) \in RAE, \\ (cv_v, dv) \in WAE\}$$

The second property guarantees conflictless classification of data access edges.

Property 2 For any flow dependence edge (cv_u, cv_v), let a D-node contains variables concerning the flow dependence of (cv_u, cv_v) be dv. Conflictless classifications of the write access edge (cv_u, dv) into Π_{WAE} or π_{WAE} and the read access edge (dv, cv_v) into Π_{RAE} or π_{RAE} in case (cv_u, cv_v) is classified into Π or π are as follows.

- If $(cv_u, cv_v) \in \Pi$,
  - $(cv_u, dv) \in \Pi_{WAE}$, $(dv, cv_v) \in \pi_{RAE}$, or
  - $(cv_u, dv) \in \pi_{WAE}$, $(dv, cv_v) \in \Pi_{RAE}$, or
  - $(cv_u, dv) \in \Pi_{WAE}$, $(dv, cv_v) \in \Pi_{RAE}$.
- If $(cv_u, cv_v) \in \pi$,
  - $(cv_u, dv) \in \pi_{WAE}$, $(dv, cv_v) \in \pi_{RAE}$, or
  - $(cv_u, dv) \in \Pi_{WAE}$, $(dv, cv_v) \in \Pi_{RAE}$.

Likewise for output dependence edges and anti-dependence edges. (It is trivial. See **Fig. 5** instead of the proof.) □

Given a conflictless classification of data dependence edges and data access edges we can define a dependence graph as described in Section 3.3. We have to recompute the costs of the nodes and the edges of the dependence graph to evaluate minimum parallel execution time at program and data partitioning given by the classification. The recomputation is straightforward. 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 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 $\omega_{RAE}((dv, cv))$ 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 cost of a data access edge is time required for interprocessor communication of the corresponding data access as described in Section 3.3.

4.4 The CDP^2 Algorithm

The CDP^{2}Algorithm is an A* algorithm to find the classification of data dependence edges and data access edges which minimizes the critical path cost of the dependence graph reproduced by the classification.

- **The Definition of the State.** The CDP^{2}Algorithm defines its state as a sextuplet $(\rho, \chi, \rho_{RAE}, \chi_{RAE}, \rho_{WAE}, \chi_{WAE})$. Here ρ is a set of the data dependence edges classified into π and χ is a set of the data dependence edges classified into neither Π nor π yet. ρ_{RAE}, χ_{RAE}, ρ_{WAE}, and χ_{WAE} are similar sets but of data access edges. We denote a set of the data dependence edges classified into Π, a set of the read access edges classified into Π_{RAE}, and a set of the write access edges classified into Π_{WAE} by P, P_{RAE}, and P_{WAE} respectively. They are represented as follows.

$$\begin{aligned}
P &= DDE - (\chi \cup \rho) \\
P_{RAE} &= RAE - (\chi_{RAE} \cup \rho_{RAE}) \\
P_{WAE} &= WAE - (\chi_{WAE} \cup \rho_{WAE})
\end{aligned}$$

- **The Initial State.** The CDP^{2}Algorithm defines the initial state of the problem as state $(\emptyset, DDE, \emptyset, RAE, \emptyset, WAE)$. At the initial state no data dependence edge is classified into Π and π. It is likewise for data access edges.
- **The Final State.** A state $(\rho, \chi, \rho_{RAE}, \chi_{RAE}, \rho_{WAE}, \chi_{WAE})$ is a final one if $\chi = \emptyset$. At final states all the data dependence edges are classified into Π or π.
- **The State Transition Procedure.** Let us suppose the CDP^{2}Algorithm removes a state $(\rho, \chi, \rho_{RAE}, \chi_{RAE}, \rho_{WAE}, \chi_{WAE})$ from the list λ. The CDP^{2}Algorithm picks an arbitrary data dependence edge $dde = (cv_u, cv_v)$ in χ to derive new states. Here we define sets of data access edges denoted by RAE_u, WAE_u, RAE_v, and WAE_v as follows.

$$\begin{aligned}
RAE_u &= \{(dv, cv_u) \in RAE | \\
&\qquad dv \in DV_a(dde)\} \\
WAE_u &= \{(cv_u, dv) \in WAE | \\
&\qquad dv \in DV_f(dde) \cup DV_o(dde)\} \\
RAE_v &= \{(dv, cv_v) \in RAE | \\
&\qquad dv \in DV_f(dde)\} \\
WAE_v &= \{(cv_v, dv) \in WAE | \\
&\qquad dv \in DV_a(dde) \cup DV_o(dde)\}
\end{aligned}$$

We also define a set of the data dependence edges in all the paths from cv_u to cv_v consisting of only data dependence edges as convex(dde).

The CDP^{2}Algorithm considers two cases, namely the case dde is classified into Π and the case dde is classified into π.

For the case dde is classified into Π we can derive at most three conflictless states as follows according to Property 2, if there exists no data dependence edge $dde' \in \rho$ such that $dde \in$ convex(dde').

(1) $\rho' := \rho$, $\chi' := \chi - \{dde\}$.
(2) $\rho'_{RAE1} := \rho_{RAE}$, $\rho'_{RAE2} := \rho_{RAE}$, $\rho'_{RAE3} := \rho_{RAE}$, $\chi'_{RAE} := \chi_{RAE}$
(3) $\rho'_{WAE1} := \rho_{WAE}$, $\rho'_{WAE2} := \rho_{WAE}$, $\rho'_{WAE3} := \rho_{WAE}$, $\chi'_{WAE} := \chi_{WAE}$
(4) If (cv_u, cv_v) represents a flow dependence,
  (a) $\rho'_{RAE1} := \rho'_{RAE1} \cup RAE_v$
  (b) $\rho'_{WAE2} := \rho'_{WAE2} \cup WAE_u$
  (c) $\chi'_{RAE} := \chi'_{RAE} - RAE_v$
  (d) $\chi'_{WAE} := \chi'_{WAE} - WAE_u$
(5) If (cv_u, cv_v) represents an output dependence,
  (a) $\rho'_{WAE1} := \rho'_{WAE1} \cup WAE_v$
  (b) $\rho'_{WAE2} := \rho'_{WAE2} \cup WAE_u$
  (c) $\chi'_{WAE} := \chi'_{WAE} - (WAE_u \cup WAE_v)$
(6) If (cv_u, cv_v) represents an anti-dependence,
  (a) $\rho'_{WAE1} := \rho'_{WAE1} \cup WAE_v$
  (b) $\rho'_{RAE2} := \rho'_{RAE2} \cup RAE_u$
  (c) $\chi'_{RAE} := \chi'_{RAE} - RAE_u$
  (d) $\chi'_{WAE} := \chi'_{WAE} - WAE_v$
(7) If $RAE_u \subseteq \chi_{RAE} \cup P_{RAE}$, $WAE_u \subseteq \chi_{WAE} \cup P_{WAE}$, $RAE_v \subseteq \chi_{RAE} \cup \rho_{RAE}$, and $WAE_v \subseteq \chi_{WAE} \cup \rho_{WAE}$, insert a new state $(\rho', \chi', \rho'_{RAE1}, \chi'_{RAE}, \rho'_{WAE1}, \chi'_{WAE})$ to λ.
(8) If $RAE_u \subseteq \chi_{RAE} \cup \rho_{RAE}$, $WAE_u \subseteq \chi_{WAE} \cup \rho_{WAE}$, $RAE_v \subseteq \chi_{RAE} \cup P_{RAE}$, and $WAE_v \subseteq \chi_{WAE} \cup P_{WAE}$, insert a new state $(\rho', \chi', \rho'_{RAE2}, \chi'_{RAE}, \rho'_{WAE2}, \chi'_{WAE})$ to λ.
(9) If $RAE_u \subseteq \chi_{RAE} \cup P_{RAE}$, $WAE_u \subseteq \chi_{WAE} \cup P_{WAE}$, $RAE_v \subseteq \chi_{RAE} \cup P_{RAE}$, and $WAE_v \subseteq \chi_{WAE} \cup P_{WAE}$, insert a new state $(\rho', \chi', \rho'_{RAE3}, \chi'_{RAE}, \rho'_{WAE3}, \chi'_{WAE})$ to λ.

For the case *dde* is classified into π we can derive at most two conflictless states as follows according to Property 2, if there exists no data dependence edge $dde' \in P \cap \text{convex}(dde)$.

(1) $\rho' := \rho \cup \{dde\}$, $\chi' := \chi - \{dde\}$.
(2) $\rho'_{RAE4} := \rho_{RAE}$, $\rho'_{RAE5} := \rho_{RAE}$, $\chi'_{RAE} := \chi_{RAE}$
(3) $\rho'_{WAE4} := \rho_{WAE}$, $\rho'_{WAE5} := \rho_{WAE}$, $\chi'_{WAE} := \chi_{WAE}$
(4) If (cv_u, cv_v) represents a flow dependence,
  (a) $\rho'_{WAE4} := \rho'_{WAE4} \cup WAE_u$, $\rho'_{RAE4} := \rho'_{RAE4} \cup RAE_v$
  (b) $\chi'_{RAE} := \chi'_{RAE} - RAE_v$
  (c) $\chi'_{WAE} := \chi'_{WAE} - WAE_u$
(5) If (cv_u, cv_v) represents an output dependence,
  (a) $\rho'_{WAE4} := \rho'_{WAE4} \cup (WAE_u \cup WAE_v)$
  (b) $\chi'_{WAE} := \chi'_{WAE} - (WAE_u \cup WAE_v)$
(6) If (cv_u, cv_v) represents an anti-dependence,
  (a) $\rho'_{RAE4} := \rho'_{RAE4} \cup RAE_u$, $\rho'_{WAE4} := \rho'_{WAE4} \cup WAE_v$
  (b) $\chi'_{RAE} := \chi'_{RAE} - RAE_u$
  (c) $\chi'_{WAE} := \chi'_{WAE} - WAE_v$
(7) If $RAE_u \subseteq \chi_{RAE} \cup \rho_{RAE}$, $WAE_u \subseteq \chi_{WAE} \cup \rho_{WAE}$, $RAE_v \subseteq \chi_{RAE} \cup \rho_{RAE}$, and $WAE_v \subseteq \chi_{WAE} \cup \rho_{WAE}$, insert a new state $(\rho', \chi', \rho'_{RAE4}, \chi'_{RAE}, \rho'_{WAE4}, \chi'_{WAE})$ to λ.
(8) If $RAE_u \subseteq \chi_{RAE} \cup P_{RAE}$, $WAE_u \subseteq \chi_{WAE} \cup P_{WAE}$, $RAE_v \subseteq \chi_{RAE} \cup P_{RAE}$, and $WAE_v \subseteq \chi_{WAE} \cup P_{WAE}$, insert a new state $(\rho', \chi', \rho'_{RAE5}, \chi'_{RAE}, \rho'_{WAE5}, \chi'_{WAE})$ to λ.

For both cases a new state is not inserted to λ when the new classification of data access edges specified by the state overrides a decided classification of some data access edges.

◎ **The Definition of the State Evaluation Value.** The evaluation value of a state $(\rho, \chi, \rho_{RAE}, \chi_{RAE}, \rho_{WAE}, \chi_{WAE})$ is the critical path cost of the dependence graph reproduced by the halfway edge classification which $(\rho, \chi, \rho_{RAE}, \chi_{RAE}, \rho_{WAE}, \chi_{WAE})$ defines. The edges in χ, χ_{RAE}, and χ_{WAE} are dealt as if they did not exist in the DPG on reproduction of the dependence graph and computation of the critical path cost of the reproduced dependence graph. States derived by the above transition procedure never have less evaluation values than their original state since dependence graphs reproduced by the derived states have more edges, which can contribute to increase the critical path cost, than that of their original state.

The final state obtained by the CDP^2 Algorithm gives the edge classification which partitions the given program and its data optimally. The complexity of the algorithm is exponential in the worst case.

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 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 no task.

5. Conclusion

In this paper we formalized a problem to partition a given program and its data in an optimal form, the data-program partitioning problem, 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 an A^* algorithm which solves the data-program partitioning problem.

In this paper we employ an A^* algorithm to solve the data-program partitioning problem. It is because we concentrate on importing the existing algorithm on the traditional dependence graph, namely Girkar's program partitioning algorithm, to the DPG with some extension and improvement to consider data partitioning and transferring optimization simultaneously.

As described in Section 4.1 the A^* algorithm can reduce its computation time by biasing state evaluation values and suppressing derivation of less attractive states. It will be our prior future work to find an effective biasing method for the CDP^2 Algorithm.

Although the current CDP^2 Algorithm picks unclassified data dependence edges in an arbitrary order, sophisticating the order of data dependence edge picking will contribute to reduce computation time of the CDP^2 Algorithm by deriving less number of states. It is also a future work to develop a heuristics on the order of picking data dependence edges to suppress the number of derived states. For both future works we consider utilizing evaluation measures used in list scheduling algorithms6),7) 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.

Besides the A^* algorithm there can be more efficient algorithms to solve the data-program partitioning problem under our formalization. However, it is intractable to evaluate actual performance of the algorithms, which changes depending on their inputs, without experiments applying algorithms to real applications. It should be another future work to compare algorithms to solve the data-program partitioning problem with actual inputs of real applications.

Acknowledgments The authors would like to acknowledge helpful suggestions and observations by Prof. Keijiro Araki of Kyushu University and anonymous referees. Several stimulating discussions with Prof. Masafumi Yamashita of Kyushu University and Prof. Satoshi Fujita of Hiroshima University are also greatly acknowledged.

References

1) Bacon, D. F., Graham, S. L. and Sharp, O. J.: Compiler transformations for high-performance computing, *ACM Computing Surveys*, Vol. 26, No. 4, pp. 345–420 (1994).

2) Ferrante, J., Ottenstein, K. J. and Warren, J. D.: The program dependence graph and its use in optimization, *ACM Trans. on Programming Languages and Systems*, Vol. 9, No. 3, pp. 319–349 (1987).

3) Girkar, M. and Polychronopoulos, C. D.: The hierarchical task graph as a universal intermediate representation, *Int. Journal of Parallel Programming*, Vol. 22, No. 5, pp. 519–551 (1994).

4) Girkar, M. B. and Polychronopoulos, C. D.: Partitioning programs for parallel execution, *Proc. of the 1988 Int. Conf. on Supercomputing*, pp. 216–229 (1988).

5) Gupta, M. and Banerjee, P.: 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 (1992).

6) Kasahara, H., Honda, H. and Narita, S.: Parallel processing of near fine grain tasks using static scheduling on OSCAR, *Proc. Supercomputing '90*, pp. 856–864 (1990).

7) Kasahara, H. and Narita, S.: Practical multiprocessor scheduling algorithms for efficient parallel processing, *IEEE Trans. on Computers*, Vol. C-33, No. 11, pp. 1023–1029 (1984).

8) Koelbel, C. H., Loveman, D. B., Schreiber, R. S., Steele Jr., G. L. and Zosel, M. E.: *The High Performance Fortran Handbook*, The

MIT Press (1994).

9) Li, J. and Chen, M.: 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 (1990).

10) Nakanishi, T., Joe, K., Saito, H., Polychronopoulos, C. D., Fukuda, A. and Araki, K.: The data partitioning graph: extending data and control dependencies for data partitioning, *Proc. of the 7th Int. Workshop on Languages and Compilers for Parallel Computing*, pp. 170–185 (1994).

11) Sarkar, V.: *Partitioning and Scheduling Parallel Programs for Multiprocessors*, The MIT Press (1989).

12) Tu, P. and Padua, D.: Automatic array privatization, *Proc. of the 6th Int. Workshop on Languages and Compilers for Parallel Computing*, pp. 500–521 (1993).

(Received October 30, 1998)
(Accepted January 11, 1999)

Tsuneo Nakanishi received the B.Eng. degree in communication engineering from Osaka University, Osaka, Japan, in 1993, and the M.Eng. and D.Eng. degrees in information science from Nara Institute of Science and Technology, Nara, Japan, in 1995 and 1998 respectively. He was a Research Fellow of the Japan Society for the Promotion of Science in 1996–1998. Since 1998 he has been an Assistant Professor in Nara Institute of Science and Technology. His current research interests are on compiler optimizations, parallel and distributed computing, and visualization and he is currently engaged in implementation of a parallelizing compiler for distributed memory multiprocessors. He is a member of the ACM, the IEEE Computer Society and the Information Processing Society of Japan (IPSJ).

Kazuki Joe received the B.S. degree in mathematics from Osaka University, Osaka, Japan, in 1984, and the M.S. and Ph.D. degrees in information science from Nara Institute of Science and Technology, Nara, Japan, in 1995 and 1996 respectively. From 1984 to 1986, he was a Software Engineer of Japan DEC, Osaka, Japan. From 1986 to 1990, he was a Researcher of ATR Auditory and Visual Perception Research Lab, Kyoto, Japan, where he developed off-line handwritten Japanese Kanji character recognition system with using neural networks. From 1991 to 1993, he was a Senior Researcher of Kubota Corporation, Osaka, Japan, and worked for scalable shared memory multiprocessor computer research and development. From 1996 to 1997, he was an Assistant Professor in Nara Institute of Science and Technology, Nara, Japan. From 1997 to 1999, he was an Associate Professor in Wakayama University, Wakayama, Japan. He is currently a Professor in Nara Women's University, Nara, Japan. His research interests include parallel computer architectures, analytic modeling for parallel computers, parallelizing compilers, neural networks, and image processing. He is a member of the IEEE Computer Society and the Information Processing Society of Japan (IPSJ).

Constantine D. Polychronopoulos received the Dipl. degree in mathematics from the University of Athens, Athens, Greece, in 1980, the M.S. degree in computer science from Vanderbilt University, TN, USA, in 1982, and the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign, IL, USA, in 1986. Since 1986 he has been a faculty member of the Department of Electrical and Computer Engineering (ECE) and the Center for Supercomputing Research and Development (CSRD) of the University of Illinois. He is currently a Professor of the ECE and a Director of the CSRD. At CSRD he participated in the design of the Cedar compiler and the Parafrase II parallelizing compiler project and manages the PROMIS parallelizing compiler project in these years. His research interests are on parallel computer architectures, their compilers, and operating systems. He was a Fulbright Scholar in 1981–1982 and a recipient of the Presidential Young Investigator Award in 1989. He is a member of the ACM and the IEEE.

Akira Fukuda received the B.E., M.E., and Dr.Eng. degrees in computer science and communication engineering from Kyushu University, Fukuoka, Japan, in 1977, 1979, and 1985 respectively. From 1979 to 1981, he worked for the Nippon Telegraph and Telephone Corporation, Tokyo, Japan, where he engaged in research on performance evaluation of computer systems and queueing theory. From 1981 to 1991 and from 1991 to 1993, he worked for the Department of Information Systems and Department of Computer Science and Communication Engineering of Kyushu University respectively. Since 1994 he has been a Professor in Nara Institute of Science and Technology, Nara, Japan. His research interests include parallel and distributed operating systems, parallel processing, and performance evaluation of computer systems. He is a member of the ACM, the IEEE Computer Society, the Information Processing Society of Japan (IPSJ), and the Operating Research Society of Japan (ORSJ).