1000-9825/2001/12(10)1434-13 ©2001 Journal of Software 软件学报 Vol. 12, No. 10

A Time-Slicing Optimization Framework of Computation Partitioning for Data-Parallel Languages*

YU Hua-shan, HU Chang-jun, HUANG Qi-jun, DING Wen-kui, XU Zhuo-qun

(*Department of Computer Science and Technology, Beijing University, Beijing* 100871, *China*)

E-mail: yuhs@ailab.pku.edu.cn

http://www.pku.edu.cn

Received June 28, 2000; accepted March 1, 2001

Abstract: Computation partitionings (CP) for the data-parallel statements in a program have a dramatic impact on its performance. Although the problem has been widely studied, previous approaches focus on improving spatial locality of the chosen CP. A time-slicing optimization framework is presented, which integrates many important optimization strategies, to select optimal CPs for parallel loop constructs. In the framework, a CP is represented by a directed graph, which not only represents a mapping of the operations in a parallel statement into processors, but also specifies the dependency constraints for operations in different processors. This approach is to evaluate the efficiency of each CP choice and to find the one with the best overall execution time. The evaluation method synthesizes the four aspects of load balance, operation-independence between processors, spatial locality and temporal locality for each CP. The framework has been implemented in a HPF compiler p_HPF for FORALL construct. Experimental results show that the framework is of generality with desired speedups for a wide variety of data-parallel applications. With a very little modification, it can also be applied to many other kinds of data-parallel statement.

Key words: data parallelism implementation; cluster parallel computing; computation partitioning; data dependency analysis; data reuse analysis; load-balancing; communication optimization

Data-parallel programming languages provide a simple, portable, abstract programming model applicable to parallel computation[1,2]. Programmers can simply specify the computing parallelism in a single-threaded program with its data-distribution directives and parallel statements. The task of computation decomposing is largely left to the language compilers. The compiler is also responsible for optimizing the overall performance of the program's execution. It is a challenge to partition computation for data-parallel loop constructs, such as FORALL construct in HPF[1], which usually contains multiple statements with masked and different iteration spaces. Figure 1 illustrates the complexity of data-parallel loop constructs by a FORALL construct with masked iteration space. The loop body contains 3 statements, s_1, s_2 and s_3. Statement s_1 assigns to variable $b[i]$ with a value of function fun(i),

---

* Supported by the National High Technology Development 863 Program of China under Grant No. 863-306-ZT01-02-3 (国家863高科技发展计划)

YU Hua-shan was born 1971. He is a Ph.D. student. His research interests are parallel compiling, parallel programming environment and parallel computer architecture. **HU Chang-jun** was born in 1963. He is a Ph.D. student. His research interests are data engineering, parallel computing and software integration. **HUANG Qi-jun** was born in 1973. He is a Ph.D. student. His research interests are parallel compiling and parallel algorithms. **DING Wen-kui** was born in 1946. He is an associate professor. His research interests are system software and parallel computing. **XU Zhuo-qun** was born in 1937. He is a professor and doctoral supervisor. His current research areas include parallel computing, geography information system and artificial intelligence.

© 中国科学院软件研究所 http://www.jos.org.cn

which can be a quite complex computation, and s_3 itself is also a data-parallel loop statement, introducing a new nested and masked iteration space. In terms of data dependency between statements, obviously there exist a number of instances in the iteration space between instances of s_1 and s_2, and between instances of s_1 and s_3.

Most parallel compilers to date[2~8] primarily use owner-computes rule[9] to partition computation for parallel statements. It specifies that a computation is executed by the owner of the value being computed. According to this rule, as well as other variants used in some compilers (e.g., integer set framework in dHPF[2] and affine mapping framework in SUIF[3]), a computation partitioning (CP) for a statement is a specification of which processor(s) must execute each dynamic instance of the statement. And a CP is typically expressed in terms of the processor(s) that own a particular set of data elements. For example, let *stmt* be a statement enclosed in a loop nest with index vector i and A be an array variable, the CP ON_HOME($A(f(i))$), specifies that the dynamic instance of stmt in iteration i will be executed by the processor(s) that own array element(s) $A(f(i))$. The affine mapping framework further restricts all statements in a loop to have the same CP. The integer set framework permits each statement in a program to have its own CP, and extends a CP to be the union of one or more ON_HOME terms: $\bigcup_{k=1,n}\{\text{ON_HOME}(A_k(f_k(i)))\}$. The fundamental disadvantage of the ON_HOME term (and its union) is that it cannot express the operation dependency in a statement as illustrated by Fig. 1, hence limits its generality and the optimizing strategies it supports.

```
    Integer a(5,5), b(6), c(5)
    ...
    FORALL (i=1:5, b[i]. NE. 0)
s1    b[i]=fun (i)
s2    c[i]=b[i+j]
s3    FORALL j=1:5, b[j]>2) a[i,j]=6
    END FORALL
```

Fig. 1

Previous research has focused on compilation strategies to minimize the amount of remote data accesses and improve spatial locality in each partition[2,3,9], but paid little attention to the temporal locality[10] of data accesses and synchronization overhead incurred by operation dependency in the statements. This paper presents a time-slicing optimization framework for regular data-parallel applications on message-passing systems. In the framework, a CP for a statement is presented as a directed graph, the CP not only decomposes the computation into several partitions and assigns each partition to a node, but also specifies explicitly the dependency between partitions by its directed edges. The framework provides a formal method to partition computation for data-parallel loop constructs, and find an optimum over a set of possible CPs by evaluating the efficiency of each CP choice. For each possible CP, the evaluation concerns not only of its spatial locality, but also of the temporal aspects of its parallel execution over the iteration space. The model has been implemented in a data parallel compiler *p*_HPF for FORALL construct. With little technical modification, this implementation can also be used in other data-parallel languages and other major loop constructs such as INDEPENDENT DO construct.

In Section 1, we introduce the definitions of several notations, and give the representation of CP in the time-slicing optimization framework. In Section 2, we first present a formal description to outline the design of the framework, then we discuss its optimization strategies to construct an optimal CP. We describe an implementation of the framework and illustrate its performance evaluation with experimental results of two benchmarks in Section 3. Section 4 concludes with a summary.

1 The Definitions

In this section, we define our notations of parallel statements, its dependency and parallelism, and illustrate these concepts with corresponding examples. Figure 2 shows a parallel-statement fragment used in this section. In the example, the FORALL construct contains 4 assignments (component statements) s_1, s_2, s_3 and s_4. The s_3, which is contained in a component FORALL statement, has a different iteration space against others. There is a declaration of a group p of 4 processors, and data distribution of integer arrays a, b, c and d over the processor group.

© 中国科学院软件研究所 http://www.jos.org.cn

```
Integer a(100,100), b(100,100)
Integer c(100,100), d(100,100,100)
Processor p(4)
Distribute (*,block) onto p:a,b
Distribute c(block,*) onto p
Distributed d(*,block,*) onto p
...
FORALL (i=1:99, j=1:100)
s1   a[i,j]=fun (i,j)
s2   b[i,j]=a[i,j]+c[i+1,j]
       FORALL (k=1:99)
s3       d[i,j,k]=a[i,k]*b[k,j]
s4   c[i,j]=b(i+1,j)+a[i+1,j]
END FORALL
```

Fig. 2 An HPF example

• CD-Graph

A CD-Graph (Computation Dependency Graph) is a directed graph $\langle \mathscr{L}, \Psi \rangle$ that defines the computation dependency in a parallel loop construct (like FORALL construct), where

(1) $\mathscr{L}$ is the set of nodes. Each component-statement in the loop body is represented as a node. Actually, a node represents all instances over its iteration space of the same statement.

(2) Ψ is a set of directed edges in the CD-Graph . Every edge in Ψ belongs to one of two edge-types, labeled with α and β respectively.

An edge $s_1 \rightarrow s_2$ labeled with α specifies:

a. statement s_1 is lexically before statement s_2;

b. s_1 and s_2 have a common iteration space;

c. for any two instances i_1 of s_1 and i_2 of s_2, if a data is referred by one instance and updated by the other instance, then $i_1 = i_2$.

An edge $s_1 \rightarrow s_2$ labeled with β specifies:

a. statement s_1 is lexically before statement s_2;

b. there are such two instances i_1 of s_1 and i_2 of s_2, where $i_1 \neq i_2$, that a data is referred by one instance and updated by another instance.

Figure 3 is the CD-Graph for the HPF example.

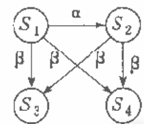

Fig. 3 CD-graph of the HPF program in Fig. 2

• LIC

An LIC (Loop-Independent Component) is a sub-set of $\mathscr{L}$ in the corresponding CD-Graph, where two of any node v_1 and v_2 confirm the followings:

(1) statements in v_1 and v_2 have a common iteration space;

(2) if v_2 is reachable from v_1 in the CD-Graph by path l, all edges in the path l are labeled with α.

There are five possible LICs in the example in Fig. 2: $\{s_1, s_2\}$, $\{s_1\}$, $\{s_2\}$, $\{s_3\}$, $\{s_4\}$.

• ISP

An ISP (Iteration Space Partitioning) is a mapping from an iteration space to processors, and the mapping is specified by one data reference description within the related statement (or statements). An ISP$(A_k(f_k(i)))$ specifies that iteration i is mapped to the processor(s) that own the array element(s) $A_k(f_k(i))$, where $A_k(f_k(i))$ is a data-references in iteration space i. Figure 4 shows two ISPs for the iteration space defined by the outer FORALL construct in Fig. 2, the first is defined by ISP$(a[i,j])$ and the second is specified by ISP$(c[i+1,j])$.

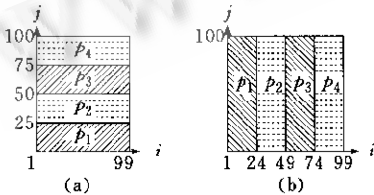

Fig. 4 Two ISPs

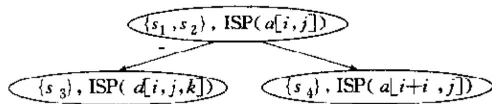

Fig. 5 A PD graph for the example in Fig. 2

When there is a m-dimension affine mapping $\Phi(i)$ for ISP$(A_k(f_k(i)))$, such that the owner(s) of $A_k(f_k(i_1))$ and $A_k(f_k(i_2))$ are different if and only if $\Phi(i_1) \neq \Phi(i_2)$, we say ISP$(A_k(f_k(i)))$ is a m-dimension ISP, and m is its rank. All iterations mapped to processor proc by ISP $(A_k(f_k(i)))$ are represented as *LocSpace* (*proc*,

© 中国科学院软件研究所 http://www.jos.org.cn

$ISP(A_k(f_k(i)))$.

• PD-Graph

A PD-Graph (Partition Dependency Graph) is a directed graph $\langle \mathscr{L}, \psi \rangle$ for a parallel loop construct, where

(1) $\mathscr{L}$ is the set of nodes. Each node consists of a LIC and an ISP. The (ISP, LIC) specifies an mapping of each instance i of s in LIC to processor(s) $ISP^{-1}(i)$. LICs in different nodes are disjoint, and every component statement in the loop construct must have a corresponding LIC in the PD-Graph.

(2) ψ is a set of directed edges in the PD-Graph. An edge $(ISP_1, LIC_1) \rightarrow (ISP_2, LIC_2)$ belongs to ψ if and only if there are at least one statement s_1 in ISP_1 and one statement s_2 in ISP_2, such that $s_1 \rightarrow s_2$ is an edge in the corresponding CD-Graph.

Each PD-Graph is associated with a cost, which is the amount of weights of its nodes and edges. Figure 5 shows a PD-Graph for the example in Fig. 2, but all weights are omitted.

2 A Time-Slicing Optimization Framework for Data-Parallel Compilation

In general, the efficiency of a parallel program in a message-passing system is affected by

• communication for non-local accesses (i. e. send and receive messages)

In the SPMD codes, explicit messages are required when remote data is read/written. The cost of communication is determined by message number, message size and communication patterns. A statement's CP is communication-free if and only if each partition accesses only local data.

• load-balance

A CP for a statement partitions its iteration space to several sub-spaces. When the sub-spaces on all available processors are disjoint and the sizes of the sub-spaces are equal to each other, computation of the statement is evenly distributed. The CP is considered load-balanced if it evenly distributes the computation into available processors.

• the number of guards executed on each processor

Guards are such conditions introduced into the SPMD codes, that they are executed on each processor to determine which statement instances it must execute in each iteration. The number of guards and their positions in the SPMD codes are indications of the overhead for determining the local iteration-spaces.

• the number of executed synchronization barriers

A synchronization barrier is such an event introduced into the SPMD codes, that two or more processors belonging to some group are blocked until all members of the group have been blocked. For a statement containing multiple components, if two of its operations op_1 and op_2 are assigned by the chosen CP to different processors, and op_2 depends on op_1, a blocking barrier is required to synchronize op_1 and op_2. The frequency of synchronization determines the overhead of synchronization barriers.

2.1 Overview of the time-slicing optimization framework

The time-slicing optimization framework is based on the concept of PD-Graph. For a parallel statement, the PD-Graph divides its component instances into several sets, corresponding to nodes in the PD-Graph. Each node specifies a space partitioning for the computation of one component (or a series of multiple components), while each partition can be executed independently and concurrently on different processors. And each directed-edge expresses a data dependency constraint between two connected component-statements. All edges in the PD-Graph specify a time partitioning (or a dependency relationship between partitions in different nodes), e. g., an edge $(ISP_1, LIC_1) \rightarrow (ISP_2, LIC_2)$, specifies that instances in LIC_1 must execute before those in LIC_2. According to this CP representation, the parallel execution of the statement is divided into several time slices, and each time slice

© 中国科学院软件研究所 http://www.jos.org.cn

contains one or more PD-Graph nodes that are not reachable to each other. Partitions in the same time slice execute concurrently if they are assigned to different processors. In particular, for a data parallel loop construct containing multiple statements, c_1 and c_2 are two of its component-statements, $\boldsymbol{G}=\langle\mathscr{L},\psi\rangle$ is one of its PD-Graphs, $c_1\in LIC_1 \wedge (ISP_1,LIC_1)\in\mathscr{L}$ and $c_2\in LIC_2\wedge(ISP_2,LIC_2)\in\mathscr{L}$, if (ISP_1,LIC_1) and (ISP_2,LIC_2) are not reachable to each other in $\boldsymbol{G}$, $\boldsymbol{G}$ specifies that c_1 and c_2 can be executed concurrently. Furthermore, this concurrency sacrifices no locality if data accessed by c_1 and c_2 is located in disjoint sub-groups of processors, hence improves the performance of the statement's parallel execution.

For a data-parallel loop construct, more than one PD-Graph can be constructed from its CP-Graph. The problem of partitioning computation for a parallel statement is to select an optimal PD-Graph from all of its possible PD-Graphs. Note that the cost of a PD-Graph is the sum of the weights associated with its nodes and edges. The weight associated with edge $(ISP_1,LIC_1)\rightarrow(ISP_2,LIC_2)$ is an estimation of the overhead to synchronize the instances in LIC_1 and LIC_2. For each node (ISP, LIC) in the PD-Graph, it specifies a space partitioning for instances in LIC, and its weight, which is a trade-off between the communication cost, load balance and guard overhead required by the computation in (ISP, LIC), evaluates the efficiency of the space partitioning. The cost of a PD-Graph is a synthetic evaluation of its communication overhead, load-balance, guard and synchronization overhead discussed above. Therefore, the PD-Graph with the minimal cost can be expected to have the best performance, and the problem of partitioning computation for a data-parallel statement can be stated as selecting a PD-Graph that has the minimal cost over all of its PD-Graphs. The mechanism used in the time-slicing optimization framework can be stated formally as that, for each data-parallel loop construct, it performs the following two steps:

1. construct the set of possible PD-Graphs from the parallel statement's CD-Graph;
2. evaluate the cost of each PD-graph in the set and choose the one with the minimal cost.

In Step 2, several important strategies are performed to improve the performance of each PD-Graph. These strategies are introduced in Sections 2.2, 2.3 and 2.4 respectively.

In principle, any data-reference in the iteration space i can define an ISP for i, and a node in the CD-Graph can choose any valid ISP (the ISP defined by a data-reference in its iteration space) to specify its space partitioning. Furthermore, given an ISP for each node in the CD-Graph, the set of combinations (ISP, LIC) is still nondderministic, and hence the PD-Graph is not decided. Therefore, the set of possible PD-Graphs for a parallel statement can be very large, and the problem of searching an optimal CP over these set is NP-complete. To improve the efficiency of selecting an optimal CP for a parallel statement, the following heuristics are important:

1. maximizing ISP rank for each iteration space

Load-balance is an important indicator of the performance for space partition. ISPs with the maximal rank likely map the iteration space to more processors than others can do, and hence they likely distribute the computation in a LIC more even. Therefore, in the optimal PD-Graph, the ISP for each LIC must have the maximal rank.

2. minimizing the number of nodes in a PD-Graph

LIC number in a PD-Graph is one key to improve temporal locality and reduce guard overhead. Given an ISP for each node in the CP-Graph, the PD-Graph with the minimal cost must have the minimal number of LICs.

3. eliminating PD-Graphs with cycles

For each data-parallel loop construct, we can construct a PD-Graph containing no cycles directly from its CD-Graph by assigning each component-statement an ISP. Obviously this PD-Graph is more optimal than any other PD-Graph that contains one or more cycles, since any cycle in a PD-Graph means that instances i_1 and i_2 of s are transitively dependent, where s is one of nodes in the cycle. Therefore, eliminating PD-Graphs with cycles don't affect the result of computation partitioning.

© 中国科学院软件研究所 http://www.jos.org.cn

With these three heuristics, the search scope for the optimal PD-Graph is greatly reduced while the optimal PD-Graph is not missed. In the time-slicing optimization framework, every possible PD-Graph is constructed and evaluated by the following steps:

(1) Selecting an ISP with the maximal rank for each node in the CD-Graph.

(2) Creating nodes for a PD-Graph: constructing a minimal number of combinations (ISP, LIC), where LICs are created by its definition from the CD-Graph, and the associated ISP choices, such that the nodes in each LIC have a common ISP.

(3) Determining edges in the PD-Graph: if edge $s_1 \rightarrow s_2$ belongs to the CD-Graph and $s_1 \in LIC_1 \neq LIC_2 \ni s_2$, adding the edge $(ISP_1, LIC_1) \rightarrow (ISP_2, LIC_2)$ to PD-Graph, where ISP_1 is the space partition for LIC_1 and ISP_2 is the space partition for LIC_2.

(4) Assigning weight for each node (ISP, LIC)

(5) Assigning weight for each edge.

In each possible PD-Graph $G=\langle \mathscr{L}, \psi \rangle$, for node $v \in \mathscr{L}$, the framework performs communication analysis and optimization, and determines its communication overhead.

2.2 Evaluating the cost of PD-graph

Now we start to introduce a heuristic method to evaluate the weights for a PD-Graph.

For each node $v=(ISP, LIC)$ in the PD-Graph, ISP decomposes the set of index vectors in i, which is the common iteration space for statements in LIC. The node specifies a space partitioning for the computation in LIC, and communications are required when non-local data is accessed. Its weight, evaluated by the following Eq. (1)~Eq. (4), is a trade-off between load-balance, communication cost and guard overhead. Eq. (1) says that *weight* (v) is the sum of Eq. (2) and Eq. (3). Eq. (2) computes the minimal communication cost of this partitioning. Eq. (3) synthesizes the effect of its load-balance and guard overhead on each processor. $Lsize(i)$ in Eq. (4) is an indicator of load-balance of the space partitioning. Since the computational complexity of LIC on processor proc is decided by the number of iterations in LocSpace(proc, ISP), the smaller the $Lsize(i)$ is, the less the computational complexity of LIC on each processor, and the better the load-balance. Due to the diversities of network architectures and memory architectures, the four quantities of w_{ini}, w_{com}, w_{ctrl} *and* w_{loc} in the equations may be different in different parallel systems.

$$weight(v) = cost_comm(v) + cost_ctrl(i) \quad (1)$$

$$cost_comm(v) = w_{ini} \times num_{\mathscr{N}} + \sum_{com \in \mathscr{N}} size_{com} \times w_{com} \quad (2)$$

$$cost_ctrl(i) = w_{loc} + w_{ctrl} \times Lsize(i) \quad (3)$$

$$Lsize(i) = \mathop{\mathrm{Max}}_{proc \in \mathscr{P}} \{ |LocSpace(proc, ISP)| \} \quad (4)$$

where,

$\mathscr{N}$	the set of communications required by non-local references in v
$\mathscr{P}$	the set of processors
w_{ini}	average cost of initializing one communication
w_{com}	average cost to move one unit of data between processors by the communication com $\in \mathscr{N}$
w_{loc}	average cost to compute local iteration space
w_{ctrl}	average control cost of one iteration
$Lsize(i)$	the maximal size of local iteration spaces on each processor
$num_{\mathscr{N}}$	the number of communications in $\mathscr{N}$
$size_{com}$	the amount of data moved between processors by the communication $com \in \mathscr{N}$

For each edge, its weight represents the number of synchronization barriers required to synchronize the

© 中国科学院软件研究所 http://www.jos.org.cn

dependent computations in the connected node pair. There are two kinds of edges: edges belonging to non-cyclic paths, and edges belonging to cycles. For the first kind, edge $e_l=(\mathrm{ISP}_1,\mathrm{LIC}_1)\rightarrow(\mathrm{ISP}_2,\mathrm{LIC}_2)$ indicates a dependency constraint: there is at least one instance in LIC_2 that must be executed after some instances in LIC_1. A synchronization barrier, which blocks the execution of any instance in LIC_2 before all instances in LIC_1 have been executed, is enough to synchronize the dependent computations. Therefore, we assign a weight 1 to the edge e_1. In contrast, for the second kind of edges, a cycle $\Gamma=(\mathrm{ISP}_1,\mathrm{LIC}_1)\rightarrow(\mathrm{ISP}_2,\mathrm{LIC}_2)\rightarrow\ldots\rightarrow(\mathrm{ISP}_k,\mathrm{LIC}_k)\rightarrow(\mathrm{ISP}_1,\mathrm{LIC}_1)$ indicates: there is a dependency sequence of statements $s_1\in\mathrm{LIC}_1,s_2\in\mathrm{LIC}_2,\ldots,s_k\in\mathrm{LIC}_k$ and $s_{k+1}\in\mathrm{LIC}_1$, such that iteration-dependent synchronizations between consecutive statements are required. The number $O(n)$ of synchronizations required by computations in Γ depends on the number of iterations in these ISPs. We usually assign a maximum of iteration number to each edge as its weight.

Followings are two important lemmas derived from the definition of edges in a PD-Graph. They are the basis of the time-slicing optimization framework for choosing optimal CP. The first lemma says that PD-Graphs containing no cycles preserve the parallelism in the responding parallel statement. And the second ensures that at least one of such PD-Graphs can be constructed with the five steps in Section 2.1 for each parallel statement. Therefore, given a statement's CD-Graph, we can construct some PD-Graphs without cycles, and each of them decomposes the statement's operation into several sets of concurrent operations. When load-balance is considered and appropriate optimization strategies are exploited to reduce communication and loop-control overhead, some of them decompose the statement's computation in a way that can achieve desired performance. With the weight definition of edges in a PD-Graph, the lemma 2 further confirms that the PD-Graph with the minimal cost contains no cycles.

Lemma 1. If the computation for a statement is partitioned by a PD-Graph containing no cycles, then partitions containing instances of the same assignment statement are independent.

Proof. Assuming computation in partition p_1 and p_2 are dependent, p_1 belongs to $(\mathrm{ISP}_1,\mathrm{LIC}_1)$ and p_2 belongs to $(\mathrm{ISP}_2,\mathrm{LIC}_2)$. Therefore in the PD-Graph, there is a path from $(\mathrm{ISP}_1,\mathrm{LIC}_1)$ to $(\mathrm{ISP}_2,\mathrm{LIC}_2)$. Since there is no cycle in the PD-Graph, $(\mathrm{ISP}_1,\mathrm{LIC}_1)$ and $(\mathrm{ISP}_2,\mathrm{LIC}_2)$ are different, hence assignment statement contained in LIC_1 and LIC_2 are different. It is impossible for p_1 and p_2 to contain instances for the same assignment statement. □

Lemma 2. Each data-parallel statement has at least one PD-Graph that contains no cycles.

Proof. This lemma can be proved by constructing a PD-Graph that containing no cycle. First we construct the statement's CD-Graph $G_c=\langle\mathcal{S},\Psi\rangle$. Let PD-Graph $G_p=\langle\mathcal{S},\psi\rangle$, where

(1) $\mathcal{S}=\{(\mathrm{ISP}(A(f(i)),\{s\}):s\in\mathcal{S}$ and $A(f(i))$ is the left-hand side variable of $s)\}$

(2) $\psi=\{(\mathrm{ISP}(A_1(f_1(i))),\{s_1\})\rightarrow(\mathrm{ISP}(A_2(f_2(i))),\{s_2\}):s_1\rightarrow s_2\in\Psi\}$

Any cycle in the G_p means that instances i_1 and i_2 of s are dependent, where s is one of nodes in the cycle. □

2.3 Strategies to improve the performance of parallel statements

As mentioned at the beginning of this section, for a parallel program, its efficiency mainly depends on locality, load balance, guards and synchronization barriers of the parallel codes. The time-slicing optimization framework exploits the following strategies to address these four challenges respectively:

1) Improving locality

Locality is improved by permitting each statement select its own ISP independently and remote data reuse among different statements. For each statement, the time-slicing optimization framework supports a broad class of ISP choices, and in fact some of these choices have the optimal spatial locality. Each statement selects its space partitioning independently to minimize the accessed remote data. And the time-slicing optimization framework improves temporal locality by grouping all component statements into several LICs where all statements have same ISP, thus remote data can be reused when it is accessed by more than one statements in the same LIC.

© 中国科学院软件研究所 http://www.jos.org.cn

2) Improving load-balance

The time-slicing framework improves load-balance by constraining that only the ISPs with the maximal rank can be chosen by component-statements in the responding iteration space. The ISPs with the maximal rank is likely to map the iteration space to the most number of processors, and hence likely to divide the parallel operations in a statement more even among a multiple processor system.

3) Reducing Guards

Given an ISP for each node in the CD-Graph, the framework tries to reduce guards in the SPMD codes by minimizing the nodes in each PD-Graph. A PD-Graph node (ISP, LIC) decomposes its operations into several partitions such that in each partition, all statements in the LIC have a common local iteration space and can share a common loop-control. Therefore in the SPMD codes, redundant guards in the same partition can be deleted by loop fusion[11], and the necessary guards are minimized when the PD-Graph has the minimal number of nodes.

4) Minimizing Synchronization Barriers

The strategies exploited to minimize synchronization barriers in the SPMD codes is the PD Graph weight measuring methods discussed in Section 2.2. The edge-weight measuring method ensures that the chosen CP contains no cycles, and this in turn make it possible to move any synchronization out of the local loops of a parallel statement. The node-weight measuring method ensures that, given an ISP for each node in the CD-Graph, the PD-Graph with the minimal cost has the minimal number of nodes. Since all dependency is partition-to-partition in a PD-Graph containing no cycles, the number of necessary synchronization barriers is determined by the edges in the PD-Graph. Therefore, the PD-Graph with the minimal cost is likely to require fewer synchronization barriers in the SPMD codes.

2.4 Improving communication performance within a chosen CP

Communication overhead for a parallel statement depends on the amount of data moved between processors and the communication strategy in SPMD codes. Reducing communication overhead is crucial on distributed-memory machines. And many strategies have been designed in two principle categories: (1) strategies to improve spatial locality of accessed data on each processor by CP selection, and (2) strategies to reduce the amount of data moved between processors and to improve communication efficiency in SPMD codes. In this section, we discuss the strategies to reduce the amount of time consumed by inter-processor data exchange. Given a PD-Graph, our model exploits the following optimization strategies, which have been implemented in our p_HPF compiler:

1) Message vectorization

Message vectorization moves communication out of loops thus replacing element-wise messages with fewer but larger messages. In the generated code, statements in a LIC share one common loop nest, and the loop body is loop-independent, it is straightforward to vectorize the element-wise communication over the local loop range.

2) Message coalescing

Message coalescing combines messages for multiple non-local references to the same or different variables, in order to reduce the total number of messages and to eliminate redundant communication. Since all statements in a LIC share one common loop nest in the generated code and the loop body is loop-independent, in each partition, redundant communication can be eliminated when two non-local references are one of the following cases:

- both are reading or updating the same non-local location
- first updating a non-local location, then reading the same location

In each partition of a LIC, when the data accessed by two non-local references locates in the same processor and both are reading references or updating references, then messages for the two references can be combined. Especially when data referred by both references belongs to the same array, the benefit is very significant.

3) Minimizing checks of buffer access via array redistribution

© 中国科学院软件研究所 http://www.jos.org.cn

Access checks are required when the same reference may access local data from an array or non-local data from a separate buffer on different loop iterations. By redistributing the array accessed by the reference, all data referred by the reference is local during iterating. In compilers that don't support redistribution, this can be achieved by copying local and non-local data into a common buffer.

4) Overlap areas for shift communication

For a non-local reference, if only a range of boundaries of array section need to be communicated between neighboring processors, extra boundary area is added to the sections. When non-local data is read in a LIC, non-local data is written into the extra boundary before executing loop nest for the LIC. When non-local data is updated, the new value is stored locally during iterating, at the end of each loop nest for LICs, the new value is written back to the neighboring processors via shift communication. Therefore, non-local data and local data can be accessed uniformly.

5) Exploiting collective communication

Exploiting collective communication is essential for achieving good speedup in important cases such as reductions, broadcasts and array redistribution, and it also may provide significant benefits for patterns such as shift communication[2]. Within these four optimization strategies above, when the problem is regular, only collective communication is considered in our implementation.

3 Implementation and Performance Evaluation

The time-slicing framework has been implemented in a HPF compiler p_HPF to support FORALL construct. This section will first describe the implementation briefly. In the second part of this section, we will use two benchmarks to illustrate performance and effectiveness of the implementation.

3.1 Implementation of the time-slicing optimization framework

Figure 6 is the framework of p_HPF, which composes of a preprocessor, an analyzer, a parallelizing framework, a set of CP constructors, a set of optimizing tools, a communication analyzer, a generator and an unparsor. For each HPF program, the preprocessor performs an analysis of syntax and semantics, then translates the program to an AST (Abstract Syntax Tree) with Sage++[12]. The analyzer's task is to get the information necessary to parallelize the HPF program and optimize its execution from the AST, such as the shape of the objective processors, data distribution and its references, dependency constraints, the iteration space of each parallel statement, and interfaces of each procedure and its instances. With all these necessary information, the parallelizing framework then decomposes the computation into each processor, optimizes the execution of the partitioning, and specifies the necessary communication and synchronization between the processors. The generator implements the result of the parallelizing framework by introducing necessary guards, communication and synchronization statements into the AST, translating the HPF statements in the AST into Fortran 77 statements. Finally, the unparsor translates the result SPMD from the form of AST into the code of Fortran77+MPI.

As discussed above, the parallelizing framework is the key component of the p_HPF. With the information collected by the analyzer, it (1) uses the CP constructors to construct at least one CP for each parallel statement, selects one optimal CP when more than one CP is available; (2) uses the communication analyzer to detect the non local references of the chosen CP; (3) uses the optimizing tools to minimize the overhead of guards, communication and synchronization. The parallelizing framework uses the time-slicing optimization framework to parallelize FORALL constructs and uses the owner-computes rule to parallelize FORALL statements respectively. Figure 7 illustrates the implementation of the time-slicing framework. Given a FORALL construct s, from the analyzer component, the time-slicing framework can get the shape of the objective processors, dependency constraints in s,

© 中国科学院软件研究所 http://www.jos.org.cn

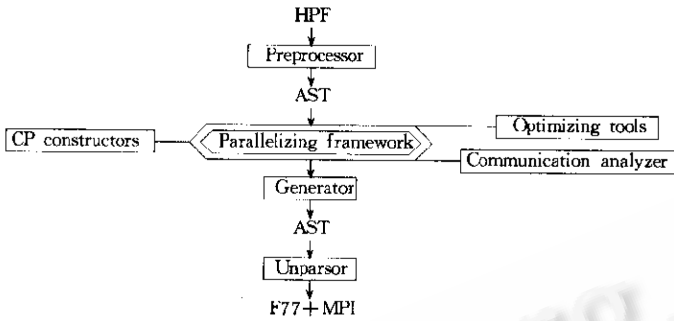

Fig. 6 The framework of p_HPF

the iteration spaces defined by s, the distribution of each data referred by s computation and data references in each iteration space. With this information, the implementation first constructs a CD-Graph for s. In Step 2, it uses the heuristics discussed in Section 2.1 to reduce the scope of the ISP for each iteration space, hence to reduce the scope of the optimal CP. Step 3 further reduces the scope of the optimal CP by decomposing the computation into some partitions that have inherit dependency between each other. Then the implementation constructs all the possible optimal PD-Graphs, performances optimization and evaluates the effectiveness respectively, and selects one optimal PD-Graph as the CP for s. Since the selected CP is a directed graph, it doesn't exact the exact execution order of each partition, communication and communication, the implementation performs an ordering for each node and edge in the chosen CP. For an edge $v_1 \to v_2$ in the chosen CP, v_1 is assigned a time slice that is ahead of the slice owning $v_1 \to v_2$, which is ahead of that occupied by v_2 too. To support the concurrency of statements in a parallel loop construct and the overlap of communication and computation, there can be other time slices between those occupied by v_1, $v_1 \to v_2$ and v_2. In the phase of code generating, the generator then translates the FOALL construct into the SPMD code of Fortran77+MPI. The computation in one node of the chosen CP is implemented by one loop of Fortran77, guards for computation decomposing among processors and communication statements for nonlocal references. The generator also introduces one synchronizing statement for each edge in the chosen CP to synchronize the execution of partitions in different processors. These statements are ordered according the time slice assignment in the chosen CP.

Step 1. Construct a CD-Graph $\langle \mathscr{L}, \Psi \rangle$ for s
Step 2. For each iteration space i defined by s, compute its ISP set $\mathscr{R}$:
  a. $\mathscr{R} = \{ISP(A_k(f_k(i))) \mid A_k(f_k(i))$ is a reference in $i\}$
  b. $\mathscr{R}' = \{ISP \mid ISP \in \mathscr{R} \wedge \exists\ ISP' \in \mathscr{R}$ and the rank of ISP is less than that of ISP'$\}$
  c. $\mathscr{R} = \mathscr{R} - \mathscr{R}'$
Step 3. $\Phi = \{LIC\ LIC \subseteq \mathscr{L}$ and $\forall\ LIC_1 \in \Phi \wedge \forall\ LIC_2 \in \Phi \wedge LIC_1 \neq LIC_2$,
  $LIC_1 \cap LIC_2 = \varnothing$, $\exists\ v_1 \in LIC_1 \wedge \exists\ v_2 \in LIC_2$, $v_1 \to v_2 \in \Psi \vee v_1 \to v_2 \in \Psi\}$
Step 4. Compute the CP choice set Γ:
  a. for each node in $\mathscr{L}$, select an ISP from the ISP set of its iteration space computed in Step 2
  b. using the algorithm discussed in Section 2.1 to construct the PD-Graph $G = \langle \mathscr{L}, \psi \rangle$, that for each $(ISP, LIC) \in \mathscr{L}$, $LIC \subseteq LIC^* \in \varnothing$
  c. using the communication analyzer to detect nonlocal reference in each $(ISP, LIC) \in \mathscr{L}$
  d. using the strategies discussed in Sections 2.3 and 2.4 to optimize the performance of G
  e. using the Eq. (1) to evaluate the effectiveness of G
  f. $\Gamma = \Gamma \cup \{G\}$
Step 5. Select the PD-Graph $G = \langle \mathscr{L}, \psi \rangle$ with the minimal cost from Γ as the CP of s
Step 6. Divide the execution time of s into $|\mathscr{L}| + |\psi|$ slices
Step 7. Assign a time slice for each node in the chosen PD-Graph
Step 8. Assign a time slice for each edge in the chosen PD-Graph

Fig. 7 Implementation of the time-slicing optimization framework

© 中国科学院软件研究所 http://www.jos.org.cn

3.2 Performance evaluation

Our evaluation was performed on a distributed memory Dawn 2000 with 32 nodes. Each node has a Motorola PowerPC604e processor running AIX4.2.0.0 at 300MHz, and has 256MB of main memory. The nodes are communicating through Dawn's MPICH. All results were collected under dedicated use of the last 16 nodes.

3.2.1 *N*-body

Three parallel versions of *N*-body have been experimented for various numbers of processors and problem sizes. The first is hand-coded with F77+MPI. The second is the HPF code from NPACK benchmark suit. The third one is modified from the second by replacing its continuous array-assignments with FORALL constructs. SPMD code for the second version was generated with the owner-computes rule, and the third with our time-slicing optimization framework.

Table 1 shows the execution times and performance comparison. The results show that, comparing with the owner-computes rule, the time-slicing optimization framework reduces execution time by nearly 25% in some cases. The results also show that, the code generated with the time-slicing optimization framework is able to achieve speedups that are comparable with hand-coded parallel performance. Using this framework, the code generated by p HPF for *N*-body is with in more than 30% of the performance of sophisticated hand-coded message-passing version of the codes, and the performance percentage can achieve as high as 59.37% in some cases.

Table 1 Timings and performance comparing for *N*-body

Problem size	$T_h(s)$	$T_{ts}(s)$	P_{ts}	$T_{oc}(s)$	P_{oc}	P_{up}
	Processor number=4					
5 000	7.94	19.37	40.99	25.83	30.74	10.25
7 000	22.01	37.91	58.06	46.87	46.96	11.10
9 000	36.29	61.10	59.39	77.82	46.63	12.76
	Processor number=8					
5 000	4.10	11.44	35.84	15.21	26.96	8.88
7 000	7.33	19.84	36.95	27.02	27.13	9.82
9 000	11.06	31.66	34.93	41.99	26.34	8.59
	Processor number=16					
5 000	2.90	8.88	32.66	11.73	24.72	7.94
7 000	4.67	15.10	30.93	18.64	25.05	5.88
9 000	6.38	21.47	29.72	26.74	23.86	5.86

T_h: execution times for the F77+MPI version

T_{ts}: execution times for code generated with the time-slicing optimization framework

T_{oc}: execution times for code generated with the owner-computes rule

$P_{ts}=(T_h \div T_{ts})\times 100$

$P_{oc}=(T_h \div T_{oc})\times 100$

$P_{up}=P_{ts}-P_{oc}$

3.2.2 Gravitational wave extraction

A Fortran 77 code is developed by the Pittsburgh group as a benchmark for the problem of gravitational wave extraction. The HPF version was ported from the serial code by converting its parallel do-loops to FORALL constructs. We defined two kinds of logic parallel systems. The first composes of a two-dimension processor matrix, and its data are distributed in the way of (*,*B*,*B*). The second composes of a one-dimension processor vector, and its data are distributed in the way of (*,*B*,*). SPMD code for the HPF codes was generated with our time-slicing optimization framework.

Table 2 shows the execution times of one time-step for different data layouts and different triplet of (n_z, n_y, n_x). The results show that the code generated with the time-slicing optimization framework is good enough of achieving

© 中国科学院软件研究所 http://www.jos.org.cn

speedups as high as 11.7 for 16 processors for this Pittsburgh benchmark. The results also indicate that load-balance and message size are important factors affecting the efficiency of the chosen CP, and hence substantiate the measure method of CP's efficiency used in the framework. For the Pittsburgh benchmark, processor matrixes can provide a better support for load-balancing partitioning than processor vectors, and correspondingly yields a better performance.

Table 2 Timings of one time-step for pittsburgh benchmark

Problem size ($n_x \times n_y \times n_z$)	$T_s(s)$	N_p	Distribution code			
			(*,B,B)		(*,B,*)	
			$T_p(s)$	P	$T_p(s)$	P
32×512×512	57.15	4	16.04	3.56	17.96	3.18
		8	9.75	5.86	11.65	4.91
		16	6.64	8.62	8.97	6.37
8×1024×1024	58.40	4	15.75	3.71	15.72	3.72
		8	8.32	7.02	9.07	6.44
		16	4.96	11.77	6.82	8.56

T_s: sequential time in seconds; T_p: parallel time in seconds

N_p: processor number; P: speedup

4 Conclusions

This paper presents a computation-partitioning (CP) model to find the optimal CP for loop constructs, which may contain multiple statements in the loop body and each component may have its own iteration space. We show that the problem of computation partitioning is to construct a directed graph with the minimal cost. For a given loop construct, the time-slicing optimization framework creates a broad class of CP choices. The model finds the best one by evaluating the performance of each choice. We also introduce some heuristics to improve the efficiency of searching the optimal CP.

The key advantage of the time-slicing optimization framework is that it tries to find a time-partition mapping that yields maximal temporal locality on each processor and minimal synchronization frequency between processors. Moreover, independent nodes that are not reachable to each other in the CP-Graph can be combined to form a time slice. This combination yields the concurrency of component-statements enclosed in a parallel statement, since all partitions in a time slice are parallel. In addition, the time-slicing framework simplify code generation greatly by encompassing many previously proposed optimizations, including loop distribution, loop fusion, statement reordering, message vectorization, message coalescing, collective communication, overlap area for shift communication and minimizing buffer access checks via data redistribution.

Although the time-slicing optimization framework can select optimal CPs for a wide variety of data-parallel applications, the design in this paper does not consider pipelines in applications such as FFT. In addition, some important optimization strategies such as loop-splitting and dataflow-based overlapping of communication and unrelated computation, are excluded out of the framework. Therefore, further work is still necessary to improve the performance of the chosen CP.

References:

[1] High Performance Fortran Forum. High Performance Fortran Language Specification. Version 2.0, 1997. http://www.crpc.rice.edu/HPFF/home.html

[2] Adve, V., Mellor-Crummey, J. Advanced code generation for high performance Fortran. In: Languages, Compilation Techniques and Run Time Systems for Scalable Parallel Systems, Chapter 18. Lecture Notes in Computer Science Series,

© 中国科学院软件研究所 http://www.jos.org.cn

Springer Verlag, 1997. http://www.cs.rice.edu/~dsystem/techPapers.html

[3] Lim, A.W., Cheong, G.I., Lam, M.S. An affine partitioning algorithm to maximize parallelism and minimize communication. In: Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing. 1999. http://www.acm.org/pubs/contents/proceedings/isca/

[4] Gupta, M., Midkiff, S., Schonberg, E., *et al*. An HPF compiler for the IBM SP2. In: Proceedings of the 1995 ACM/IEEE Supercomputing Conference. 1995. http://www.supercomp.org/sc95/proceedings/

[5] Bozkus, Z., Meadows, L., Nakamoto, S., *et al*. Compiling high performance fortran. In: Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing. CA: San Francisco, 1995. 704~709. http://www.siam.org/meetings/archives

[6] Benkner, S., Chapman, B., Zima, H. Vienna Fortran 90. In: Proceedings of the 1992 Scalable High Performance Computing Conference. Williamsburg, VA, 1992.

[7] Harris, J., Bircsak, J., Bolduc, M.R., *et al*. Compiling high performance fortran for distributed-memory systems. Digital Technical Journal of Digital Equipment Corporation, 1995,7(3):5~23.

[8] Hiranandani, S., Kennedy, K., Tseng, C.-W. Preliminary experiences with the Fortran D compiler. In: Proceedings of the Supercomputing'93. Portland, OR, 1993. http://www.acm.org/pubs/contents/proceedings/supercomputing

[9] Rogers, A., Pingali, K. Process decomposition through locality of reference. In: Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation. Portland, OR, 1989. http://www.acm.org/pubs/contents/proceedings

[10] Kenneth Hawick. High Performance Computing and Communications Glossary. Technical Report CRPC-TR94627, Center for Research on Parallel Computation, Rice University, 1994.

[11] Gerald, Roth, Ken, Kennedy. Loop Fusion in High Performance Fortran. Technical Report CRPC-TR98745, Center for Research on Parallel Computation, Rice University, 1998.

[12] Francois, Bodin, Irisa, Peter, Beckman, Dennis, Gannon, *et al*. Sage++: an Object-Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools. 1994. http://www.extreme.indiana.edu/sage/docs.html.

一个用于数据并行语言计算划分的时序优化模型

余华山，胡长军，黄其军，丁文魁，许卓群

(北京大学 计算机科学技术系,北京 100871)

摘要：一个程序中数据并行语句的计算划分(CP)对该程序的运行性能有决定性的作用.尽管人们对这一问题已经进行了广泛的研究.但这些研究的重点都集中在如何提高被选择计算划分的空间局部性上.针对并行循环结构的计算划分问题,提出了一个时序优化模型.在该模型中,一个计算划分被表示成一个有向图,在把并行语句中的操作映射到各个处理器的同时,给出了被分配到不同处理器上的操作之间的相关性.对于一条数据并行语句,时序优化模型对它的每个计算划分选择方案分别采用多种有效的优化策略进行优化;并综合考虑各个计算划分选择方案的负载平衡性、处理器间的操作依赖性、数据访问的空间局部性和时间局部性四个方面的因素,估算每个方案的执行效率;最后从这些方案中选择一个执行效率最优的方案作为该语句的计算划分.作者已在HPF编译器p_HPF采用时序优化模型实现了对FORALL结构的支持.实验结果表明,该模型具有非常好的通用性,对不同领域多种数据并行问题均取得了理想的加速比.同时,只需略微改动,该模型也可用于其他类型数据并行语句的计算划分.

关键词：数据并行;集群并行计算;计算划分;数据相关;数据重用;负载平衡;通信优化

中图法分类号：TP311 **文献标识码**：A

© 中国科学院软件研究所 http://www.jos.org.cn