An Interprocedural Parallelizing Compiler and Its Support for Memory Hierarchy Research *

Trung Nguyen Junjie Gu Zhiyuan Li

University of Minnesota, Minneapolis MN 55455, USA

Abstract. We present several new compiler techniques employed by our interprocedural parallelizing research compiler, *Panorama*, to improve loop parallelization and the efficiency of memory references. We first present an overview of the compiler and its associated memory architecture simulation environments. We then present an interprocedural array dataflow analysis, using *guarded array regions*, for automatic array privatization, an interprocedural static profile analysis, and a graph coloring scheme for parallel task assignment and data allocation which aims at reducing remote memory references while maintaining loop parallelism.

1 Introduction

In recent years, researchers see a growing interest in shared-memory multiprocessors with multi-level memories which provide private caches and/or non-uniform memory accesses (NUMA) [15, 6, 28]. In order to deliver the power of such machines to ordinary programs written by a wide range of programmers, it is important for the compilers to improve their ability to automatically identify parallelism in ordinary programs. Moreover, the compilers need to generate parallel machine code that can efficiently use the memory hierarchy by reducing remote memory references and by hiding the memory latency. In order to achieve these goals, we have developed new interprocedural analysis techniques in a source-to-source, parallelizing research Fortran compiler, Panorama. This compiler has the following important new features:

- An interprocedural symbolic array dataflow analysis for array privatization, loop parallelization and memory reference optimizations.
- An interprocedural static profile analysis which estimates *workload variance* to guide the decision on static vs. dynamic scheduling for parallel loops and *reference costs* to guide data allocation and task assignment.
- A graph coloring scheme for data allocation and task assignment to improve data locality while maintaining loop parallelism.
- Marking on *coherence writes* for efficient use of cache coherence mechanisms.

* This work was supported in part by the Army High Performance Computing Research Center, under the auspices of Army Research Office contract number DAAL03-89-C-0038 with the University of Minnesota, by National Science Foundation, grant CCR-9210913, and by Computing Devices International.

1.1 An overview of the compiler

One of the main challenges to parallelizing compilers is to exploit parallelism at very high levels of the program. Symbolic relations and dataflow information often need to be propagated between procedure boundaries during program analysis. We find it useful to construct a graph that can represent the program's control flow globally and interprocedurally, and use this graph to efficiently propagate the information that is needed by the compiler. The PIPS project first adopts such a graph, HSCG [14], and our *hierarchical supergraph* (HSG) [11] is similar to HSCG. (Our compiler and the PIPS compiler are perhaps the only two research compilers which utilize such interprocedural flow graphs for interprocedural analysis.) The HSG is a composition of the flow subgraphs of all routines in a program. Each call statement is represented by a node, termed a *call node*, while each DO loop is represented by a *loop node*. Therefore, the resulting graph contains three kinds of nodes – basic block nodes, loop nodes and call nodes. An IF condition itself forms a single basic block. Call nodes and loop nodes are compound nodes which have their attached flow subgraphs describing the control flow within the called routines and the DO loops. Due to the nested structures of DO loops and routines, a hierarchy is derived among the HSG nodes, with the flow subgraph at the highest level representing the main program. Note that the flow subgraph of a routine is never duplicated for different calls to the same routine.

Based on the HSG, the compiler computes interprocedural scalar use-def chains [17]. Combining such chains with local expression trees, we in effect obtain an interprocedural version of the *value graphs* which are proposed by Reif and Lewis [25]. The value graphs are used in our compiler for transforming array subscripts and loop control expressions into linear expressions in terms of loop indices and symbolic terms that are loop invariants. After performing a traditional data dependence analysis, the compiler puts DO loops in three categories: parallelizable, sequential (because the main part of the loops' bodies are covered by large *flow dependence* cycles), and those requiring further examination after *array privatization* [16, 20, 30, 31]. Array privatization may eliminate *output dependences* and *anti-dependences* due to memory conflicts and it is widely recognized as a key to loop parallelization. After array privatization, the compiler re-performs an interprocedural data dependence analysis to recognize new parallelizable loops.

The Panorama compiler employs several compiler techniques to enable an efficient use of multi-level memories. Where load imbalance is not a risk, the compiler statically allocate tasks and data to increase data locality, while maintaining parallelism. However, since static scheduling bears the risk of load imbalance, the compiler identifies loop candidates for static scheduling by symbolically estimating the *workload variance* among parallel loop iterations. By performing static task assignment and data allocation simultaneously, remote memory references may be reduced. Existing works focus on the total elimination of remote references [1, 24]. However, such an approach may often result in loop serialization, especially for large programs, thus sacrificing parallelism. Our method does not always attempt to totally eliminate remote memory references. Instead, our compiler attempts to *reduce* the number of remote memory references, based on a symbolic estimate of the potential penalty due to remote references for each parallel DO loop. In addition, for shared-memory multiprocessors with private caches, the compiler identifies write references which do not cause cache incoherence and thereby reduce expensive coherence actions. We shall discuss our static task profile analysis in Section 3 and discuss a graph coloring algorithm, in Section 4, for performing task assignment and data allocation simultaneously. Our strategy for reducing coherence actions has been presented in a previous paper [23]. Therefore, we shall not address this issue further in this paper.

To study the effectiveness of the compiler analysis, we are currently developing compiler modules for generating parallel codes for the Silicon Graphics Power Challenge and the Cray T3D supercomputer. At the same time, we use the compiler to generate traces which drive various simulators. A trace-generation pass is integrated into Panorama to annotate each memory reference and/or each operation in the program generated by the source-to-source translation. When the new program is compiled and executed, an annotated trace, which include both memory references and ALU operations, is produced. A data reference in the trace is either a memory reference or a register reference. A memory reference is due to either a declared variable or a compiler generated temporary. Each memory reference can be annotated with any information that the simulator may need, such as the memory address, the processor issuing the reference, and whether the reference is a read, a coherence write, or a non-coherence write[23]. During the annotation process, a memory map assigning variables to virtual addresses is created. In addition to annotating memory references and ALU operations, the annotator also inserts markers to identify parallel loops and their chosen scheduling method in the trace. The trace is then fed to an architectural simulator which generates statistics including execution time, cache miss ratio, and network traffic. Previous experimental results using this methodology are reported in [18, 22, 23]. A limitation of our trace-generation facility is the lack of sub-traces generated by the execution of library routines. In order to characterize memory activities more precisely while retaining the information provided by parallelizing compilers, we are exploring methods to combine our compiler-annotated traces with library routine traces that can be extracted by tools such as pixie [27]. The simulation issue will not be discussed further in this paper.

In the remainder of the paper, we expand the discussion of some of the techniques listed above. In addition, we present preliminary results of the Panorama compiler in section 5. We summarize our discussions in section 6.

2 Array dataflow analysis for loop parallelization

Previous works on array dataflow analysis for parallelization follow two paths. The first path pursues the goal of computing reaching-definitions for each individual array reference. Feautrier [7] suggests to establish a *source function* for each read reference to indicate which definition defines the value for each distinct array element. Maydan et al [20, 21] simplify Feautrier's method by using a Last-Write-Tree(LWT) for simpler cases. These works ignore IF statements and routine calls. Maslov [19] extends Feautrier's work by handling affine IF conditions and symbolic terms. His work, however, does not handle routine calls and does not address the issue of how to obtain the applicable IF conditions and symbolic expressions for each array reference. The use of source functions for computing reaching-definitions typically results in the need for integer programming to determine the feasibility of an arbitrary set of inequalities. Several researchers, including Gross and Steenkiste [10], Rosene [26], and Granston and Veidenbaum [9], in an apparent attempt to avoid potentially expensive symbolic manipulations and integer programming, have adopted *regular array regions* for their computation of reaching definitions. Although regular array regions cover only restricted array shapes, their operations, such as union, intersection, and difference, can be quite straightforward and efficient. Duesterwald et al [5] have examined cases in which they can compute the dependence distance for each reaching definition within a loop. These works, however, do not take IF conditions and routine calls into account. Works in the second path do not attempt to compute reaching-definitions for each individual array reference. They concern themselves of the array dataflow problem for program parallelization only, for which a *summary* approach seems sufficient and efficient. Array modifications and *upward-exposed uses* are summarized for code segments, e.g. loop iterations. Works by Li [16] and Tu and Padua [30, 31] follow this second path. Our recent work continues to follow this path and extends the previous works by taking IF conditions and call statements into account. A recent paper [11] describes our work on this topic in details. We present only an outline in this section.

It is worth mentioning that the summary approach has previously been used by Triolet and Irigoin [29], Callahan, Kennedy, Balasundrum and Havlak [2, 4, 13], and more recently, Graham et al [8], to compute data dependences instead of dataflow. Their works summarize all array uses while ours summarizes upward-exposed uses which are essential for array privatization. It is also worth mentioning that a recent work by Tu and Padua [31] handles certain IF conditions which contain symbolic terms present in array subscripts and loop limits. Our work is not restricted to such IF conditions. Since our experiments [11] and previous works [3] suggest that the lack of the handling of IF conditions, symbolic expressions, and call statements can significantly handicap parallelizing compilers, the Panorama compiler deals with all these three aspects. In order to take account of IF conditions under which an array reference is issued and to represent the set operation results in the presence of symbolic terms, we adopt a reference predicate which further qualifies a regular array region, called a *guarded array region*(GAR). Graham et al [8] use a similar representation, but they do not summarize upward-exposed uses.

2.1 Guarded array regions

In this section, we assume that the program contains no recursive calls and that a DO loop does not contain GOTO statements which make premature exits.

We also assume that the HSG contains no cycles created by backward GOTO statements. Our implementation, however, does take care of multiple exits in DO loops and backward GOTO statements, making conservative estimates when necessary. In the flow subgraph of a loop node, the back edge from the exit node to the entry node is deliberately deleted, as it conveys no additional information for array summaries. Under the above assumptions and treatment, the HSG is a hierarchical *dag* (directed acyclic graph).

A GAR contains a regular array region and a guard. In the following, we define the regular array regions first and then define GAR's.

Definition A *regular array region* of array A is denoted by $A(r_1, r_2, \cdots, r_m)$, where m is the dimension of A and r_i, $i = 1, \cdots, m$, is a range in the form of $(l : u : s)$ in which l, u, s are symbolic expressions. The triple $(l : u : s)$ represents all values from l to u with step s. An empty regular array region is represented by $\emptyset$ and an unknown regular array region is represented by Ω.

Definition A *guarded array region* (GAR) is a tuple $[P, R]$ which contains a *regular array region* R and a guard P, where P is a predicate that specifies the condition under which R is accessed. We use Δ to denote a guard whose predicate cannot be written explicitly, i.e. an unknown guard. If both $P = \Delta$ and $R = \Omega$, we say the GAR $[P, R]$ equals Ω (unknown). Similarly, if either P is $False$ or R is $\emptyset$, we say $[P, R]$ is $\emptyset$.

The predicate defined above is represented in the form of the conjunctive normal form (CNF). To keep as much information as possible, the predicate is marked as unknown only if all its conjunctive terms are marked as unknown. A conjunctive term, which is a disconjunction of relational expressions, is marked as unknown only if all the relationals are marked as unknown. Similarly, a multi-dimensional regular array region is marked as unknown only if all its dimensions are marked as unknown (see [11] for details).

For any given program segment, we use GAR's to summarize the sets listed below, which are important to our array dataflow analysis for array privatization and loop parallelization. Since we use guards, all these sets are exact sets unless the GAR's contain unknown components.

- UE – The set of the upward-exposed array elements which are used within this segment and take values defined outside this segment.
- UE_i – For an arbitrary iteration i of a DO loop, the set of the upward-exposed array elements which are used within this iteration and take values defined outside this iteration.
- MOD – The set of array elements written within this segment.
- MOD_i – For an arbitrary iteration i of a DO loop, the set of the array elements written within this iteration.
- $MOD_{<i}$ – For an arbitrary iteration i of a DO node, the set of the array elements written within the iterations prior to i.
- $MOD_{>i}$ – For an arbitrary iteration i of a DO node, the set of the array elements written within the iterations following i.

2.2 Operations on GAR's

The Panorama compiler requires three kinds of operations on GAR's, namely, union, intersection, and difference. These operations in turn are based on union, intersection, and difference operations on regular array regions as well as logical operations on predicates. Since symbolic terms may appear in both arithmetic expressions and predicates, we implement a GAR simplifier and a predicate simplifier to simplify GAR's and predicates. Hence, the operations on GAR's are composed of two simplifiers and array region operations based on basic arithmetic and logical expression operations. Due to the space limit, we discuss only the top-level operations and refer the readers to [11] for more details. Given two GAR's, $T_1 = [P_1, R_1]$ and $T_2 = [P_2, R_2]$, we have the following:

- $T_1 \cap T_2 = [P_1 \wedge P_2, R_1 \cap R_2]$
- $T_1 \cup T_2 = [P_1 \wedge P_2, R_1 \cup R_2] \cup [P_1 \wedge \overline{P_2}, R_1] \cup [\overline{P_1} \wedge P_2, R_2]$
  The above formula can be simplified in the following three common cases:
  - If $P_1 \Rightarrow P_2$, the union becomes $[P_1, R_1 \cup R_2] \cup [\overline{P_1} \wedge P_2, R_2]$.
  - If $P_2 \Rightarrow P_1$, the union becomes $[P_2, R_1 \cup R_2] \cup [P_1 \wedge \overline{P_2}, R_1]$.
  - If $R_1 = R_2$, the result is $[P_1 \vee P_2, R_1]$.
- $T_1 - T_2 = [P_1 \wedge P_2, R_1 - R_2] \cup [P_1 \wedge \overline{P_2}, R_1]$

2.3 Summary algorithms

Suppose we want to check whether loop L is parallel. Let $g(s, e)$ be the HSG of the subgraph of loop L, where s is the starting node and e is the exit node. Let $UE(n)$ and $MOD(n)$ represent the upward-exposed use set and mod set for node n respectively, and let $UE\_IN(n)$ and $MOD\_IN(n)$ represent the upward-exposed use set and mod set, respectively, at the entry point of node n for the subgraph containing node n. These sets are all represented by GAR's. The compiler performs the following recursive algorithm, *gar_summary*:

1. Summarize all basic block nodes within $g(s, e)$.
   For each basic block node n, we calculate the $MOD(n)$ and $UE(n)$ sets for arrays within node n. All predicates in GAR's of $MOD(n)$ and $UE(n)$ are set to True.
2. Summarize all compound nodes.
   (a) *Call nodes.* For a call node n, let the subgraph of the call node be $g'(s, e)$. The compiler recursively applies *gar_summary* to $g'(s, e)$, which summarizes the references to global arrays and arrays parameters. The returning results are mapped back to the actual arguments of the procedure call (reshape if necessary). The mapping results become $MOD(n)$ and $UE(n)$.
   (b) *Loop nodes.* For a loop node n, let the subgraph of the loop node be $g'(s, e)$. The compiler recursively applies *gar_summary* to $g'(s, e)$. Let $UE_{i'}(n)$ and $MOD_{i'}(n)$ represent the MOD and UE sets of $g'(s, e)$, which contain modification GAR's and upward-exposed use GAR's in one loop iteration indexed by i'. The sets $MOD_{i'}(n)$ and $(UE_{i'}(n)$

- $MOD_{<i'}(n)$) are then expanded across the i' index range to form $MOD(n)$ and $UE(n)$ for loop node n. (Details for this expansion are explained in [11]).
3. Propagate the array dataflow information.
From node e to s, the *gar_summary* algorithm traverses the nodes in $g(s,e)$ in a reverse topological order. During the summary propagation, the following flow equations are used:
$$MOD\_IN(n) = MOD(n) \cup (\bigcup_{p \in succ(n)} MOD\_IN(p))$$
$$UE\_IN(n) = UE(n) \cup (\bigcup_{p \in succ(n)} UE\_IN(p) - MOD(n))$$
Note that the set of successors of the exit node succ(e) is $\emptyset$. When applying the flow equations, we need to handle the following situations.
   - If n is a basic block containing an IF-condition, add the condition to the guard of each GAR in $MOD\_IN(n)$ and $UE\_IN(n)$.
   - If any expression in the $MOD\_IN(n)$ and $UE\_IN(n)$ contains a variable that is defined within n, then that variable must be substituted by the right-hand side of the defining statement within n. If the right-hand side is too complicated, the expression is marked as unknown. If a variable is defined by a procedure or a function, we propagate information through the subgraph of this procedure or function.

For a call node, the above algorithm produces the MOD and UE sets of $g(s,e)$, where MOD equals $MOD\_IN(s)$ and UE equals $UE\_IN(s)$. For a loop node, the algorithm produces the MOD_i and UE_i sets instead. The $MOD_{<i}$, $MOD_{>i}$ and MOD sets are obtained by three different expansions of MOD_i. These sets are then used for array privatization and loop parallelization as described below.

The general technique for array privatization is discussed in [16, 20, 30]. The most important condition to permit the privatization of array A in loop L is the absence of loop-carried dataflow due to A in L. Loop-carried dataflow due to array A in loop L does not exist if and only if $MOD_{<i} \cap UE_i = \emptyset$.

The essence of loop parallelization is to prove the absence of loop-carried dependences. For a given DO loop L with index i, if none of the GAR's is marked as unknown, then the existence of different types of loop-carried dependences can be detected as follows:

- loop-carried flow dependences: They exist if and only if $UE_i \cap MOD_{<i} \neq \emptyset$.
- loop-carried anti- dependences: They exist if and only if $UE_i \cap MOD_{>i} \neq \emptyset$.
- loop-carried output dependences: They exist if and only if $MOD_i \cap (MOD_{<i} \cup MOD_{>i}) \neq \emptyset$.

If any GAR is marked as unknown, then these conditions are conservative.

3 Static Task Profile

In this phase of the compiler, we statically create a profile of each parallel loop in terms of the variance in operation counts between the iterations. A loop with low variance is to be scheduled statically to take advantage of memory reference optimizations. By varying the threshold value for choosing static scheduling, and comparing the performance of the resulting scheduling choices, we can arrive at a heuristic to choose a desirable threshold value at compile time. The analysis to determine the variances of parallel loops is presented below.

Some of the operations in the parallel loop body may be affected by IF conditions or inner loop bounds which contain the loop index. We call the number of such operations as the *loop-variant* part of the operation count and call the number of the other operations the *loop-invariant* part. In order to estimate the degree of load imbalance, we use the compiler to estimate the relative significance of the loop-variant part.

The compiler first marks the HSG nodes, in a top-down manner, as belonging to either the loop-invariant or the loop-variant part. The compiler identifies the IF nodes (basic block nodes that contain only IF conditions) whose IF conditions depend on the parallel loop index and marks the operations belonging to any branches of such IF nodes as loop-variant. The compiler also identifies the inner sequential loops whose loop bounds depend on the parallel loop index and marks all the operations within such loops as loop-variant. All other operations are marked as loop-invariant. We then estimate the loop invariant part of the operation count as follows.

Each node b in the HSG has $COUNT(b) = (min_b, max_b)$ which represents the minimum and the maximum operation counts of the node. The following operations may be performed on such tuples:

– $COUNT(i) + COUNT(j) = (min_i + min_j, max_i + max_j)$,
  where i and j are two nodes in the HSG. The add operator is used for summing the counts on a path in the HSG.
– $Merge(COUNT(b_1), COUNT(b_2), ..., COUNT(b_n)) =$
  $(MIN(min_{b_1}, min_{b_2}, ..., min_{b_n}), MAX(max_{b_1}, max_{b_2}, ..., max_{b_n}))$,
  where b_1 to b_n are nodes in the flow graph. This $Merge$ operator is used to join the counts of different paths in the HSG.

In addition to $COUNT$, a sequential DO loop node also has two numbers ($ITER_{min}$ and $ITER_{max}$) to indicate the number of iterations that the loop will be executed. If the loop iteration's lower bound and upper bound are not functions of an index variable of an outer loop, then $ITER_{min}$ is the same as $ITER_{max}$. Both $ITER_{min}$ and $ITER_{max}$ may be symbolic expressions.

To estimate the workload variance of parallel loops, the following steps are performed:

1. Summarize the loop-invariant counts:
   – For each basic block node b, which belongs to the loop-invariant part, obtain $COUNT(b)$. Note that the minimum and the maximum operation counts of a basic node are the same.
   – For each compound node n, recursively apply the following steps:
     (a) For each node b in the flow subgraph of n, calculate:
       • $C_{in}(b)$: The minimum and maximum operation counts before the entry of node b.

- $C_{out}(b)$: The minimum and maximum operation counts after the exit of node b.

The above counts are found by solving the following flow equations, starting with the entry node of the subgraph of n and towards the exit node:

- $C_{in}(b) = Merge_p(C_{out}(p)|_{p \text{ is a predecessor of } b})$
- If b was marked as loop-invariant then

$$C_{out}(b) = C_{in}(b) + COUNT(b)$$

else

$$C_{out}(b) = C_{in}(b).$$

Initially, C_{in} and C_{out} are empty for all nodes except the entry node which has $C_{out}(entry) = COUNT(entry)$.

(b) Summarize the compound node by setting its $COUNT$ to $C_{out}(exit)$. If the compound node is a loop node, set $COUNT(loop) = C_{out}(exit) * ITER_{min}$. Note that $ITER_{min}$ is the same as $ITER_{max}$ for the DO loop since the DO loop is in the invariant part.

$COUNT(B)$, where B is a parallel loop compound node, gives the minimum and maximum estimates of the loop-invariant part of the operation count, denoted by INV_{min} and INV_{max} respectively.

2. Summarize the loop-variant counts. The loop-variant part is estimated similarly to the loop-invariant part, except that if b was marked as loop-variant then
$$C_{out}(b) = C_{in}(b) + COUNT(b)$$
else
$$C_{out}(b) = C_{in}(b).$$
Also, when summarizing a loop node, set
$$COUNT(loop) = (min(C_{out}(exit)) * ITER_{min}, max(C_{out}(exit)) * ITER_{max})$$
where $min(C_{out}(exit)$ and $max(C_{out}(exit)$ refer to the minimum and the maximum counts of $C_{out}(exit)$, respectively. $COUNT(B)$, where B is a node for a parallel loop, gives the minimum and maximum estimates of the loop-variant part of the operation count, denoted by VAR_{min} and VAR_{max} respectively.
3. Finally, for the parallel loop represented by compound node B, the coefficient of variance, cov, in operation counts between the loop iterations is estimated as:
$$cov = \sigma / \mu$$
where $\sigma = (VAR_{max}(B) - VAR_{min}(B))/2$ and $\mu = 1/2(INV_{min}(B) + INV_{max}(B) + VAR_{min}(B) + VAR_{max}(B))$. At this point, we may reach three possible situations:
   - cov evaluates to a constant.
   - Some symbolic terms remain when calculating cov but we can determine the range for cov by using one of the existing techniques[12].

– There are some unknown symbolic terms and, hence, we know nothing about *cov*. In this case, either the user chooses a scheduling method for this loop or the compiler selects a default scheduling method.

4 Task assignment and data allocation

After the compiler identifies the loops that have little or no variance in operation counts, we assume an abundance of *virtual processors* (VP's) and assign the parallel loop iterations to the VP's. Of course, the VP's are eventually folded to the physical processors available for the program execution. Under the NUMA model, each physical processor has a local memory module that is accessible by all processors. For our purpose, we regard each VP as having its own memory module as well. Concurrent with the assignment of parallel loop iterations, we allocate array data to memory modules. Previous works[1, 24] find a communication-free partitioning of arrays which, in turn, may sequentialize many parallel loops. Our goal here is to find heuristics that maximize the local memory accesses without sacrificing parallelism. Currently, we consider only single level parallel loops. For multiple loop nests, we select one level as a parallel loop and sequentialize the others. For a multi-dimensional array, our algorithm will automatically select which dimension to be distributed among the VP's. Since data allocation and the assignment of parallel loop iterations are interrelated, they are considered at the same time.

This problem can be broken down into two optimization levels:

1. Optimizing the *orientation* of the data allocation and the assignment of the parallel iterations. Suppose the VP's are numbered from 1 to N. For data allocation, orientation means whether we should allocate the elements of an array along with the increasing order of the VP's or in the decreasing order. Likewise, the orientation of the iteration assignment can be in the increasing or decreasing order.
2. Optimizing the *displacement* of the chosen orientations. Displacement means how far, if any, we should shift the data assignment or the iteration assignment to the left or right. If the optimal displacement is small, then its effect is not as important as choosing a good orientation because it can usually be covered by the effect of practical data cache block sizes which are greater than one word.

The heuristics for choosing the orientations are as follows. We create a *connection graph* $CG(d, a, e)$ where d is the set of vertices each of which represents a parallel loop in the program, a is the set of vertices each of which represents an array name, and e is the set of undirected edges each of which connects a vertex in a to a vertex in d if the array represented by a is referenced in the parallel loop represented by d. The connection graph indicates which arrays and which loops should be considered together when determining the orientations.

For each connected component in CG, we first construct an undirected *orientation graph* OG. Each node in OG represents an orientation choice, increasing or decreasing, for either an array A, or a loop L. Thus, each array A or loop L that appears in the connection graph generates two nodes, A^+ and A^-, or L^+ and L^-, in OG. Suppose an array A is referenced within a loop L. Their orientation choices, say A^+ and L^+, normally result in a number of A references that are likely to be local, which we call as the *mismatch cost*. An edge is drawn between A^+ and L^+ with an annotation of this mismatch cost. The mismatch cost is estimated symbolically from the HSG in a way similar to the estimate of the operation counts discussed in the last section. For a multi-dimensional array, the dimension of the array is also labeled along with the orientation, for example, $A^{+,2}$. An n-dimensional array will have $2n$ nodes representing the dimensions of the array and the orientation in each dimension. Figure 1 shows a simple OG example. In this graph, an edge is drawn between the node $L1^+$ and the node

```
      DOALL i=1, 100
L1       A[i] = B[i+1] + ...
      ENDDO

      DOALL i=1, N
        ... = A[N-i] + A[N-i+1]
L2      ... = B[i+2] - B[i+4]
        DO j=1, 50
          C[i,j] = ...
        ENDDO
      ENDDO


     DOALL i=1, 100
       D[i-1] = ...
       DO j=1, 20
L3       ... = C[j,i]
       ENDDO
     ENDDO
```

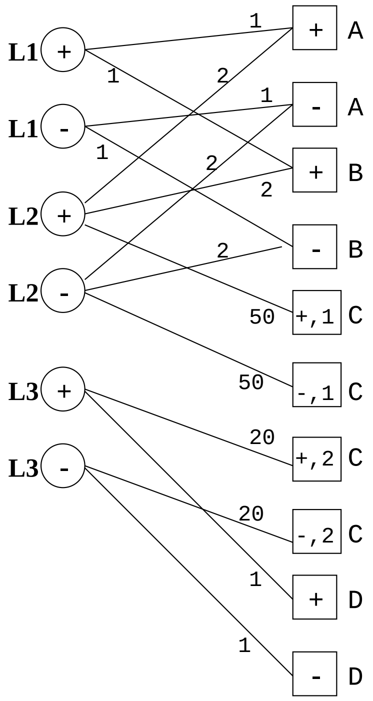

Fig. 1. Example of an OG. The circles represent loop nodes while the boxes represent array nodes.

A^+ and another edge is drawn between $L1^-$ and A^-, each with a mismatch cost of 1. These two edges indicate that if the orientations of loop $L1$ and array A are different, then there will be one remote reference to A by each VP.

After constructing OG, we attempt to find a subgraph, SOG, which contains only one node for each loop or each array and whose edges have the maximum total mismatch cost. This problem is NP-hard and we use the following heuristics for its solution. The desired subgraph SOG initially contains an arbitrary node from OG. Incrementally with a breadth-first traversal, we add to SOG the neighbors of the nodes which are already in SOG, together with the connecting edges. During this process, whenever we find that adding a new node to SOG will create a conflict, we sum the mismatch costs for each node over all its edges excluding those connecting to nodes which conflict with nodes in the current SOG. We then remove one of the conflicting nodes which has the smaller mismatch costs. If the two costs are equal, then an arbitrary choice is made to break the tie. We continue to construct SOG until all edges incident to the nodes in SOG have been considered. The final SOG gives the orientation decision.

Figure 2 shows an example of a final subgraph for OG in Figure 1, and the corresponding orientation decision is A^-, B^+, $C^{+,1}$, D^+, $L1^+$, $L2^+$ and $L3^+$. This result is obtained by the following process. We start by including $L1^+$, B^+ and A^+ in SOG. Next, we add $L2^+$ due to the edge between B^+ and $L2^+$. We then examine the edge between A^+ and $L2^-$ and find a conflict between $L2^+$ and $L2^-$. The mismatch cost comparison is a tie, so we arbitrarily choose to keep $L2^+$ in SOG. Next, the edge between $L2^+$ and A^- introduces a conflict with the existing node A^+. This time, the mismatch cost of A^- is higher and thus it replaces A^+ in SOG. We continue to add $C^{+,1}$, $L3^+$ and D^+ to SOG before we encounter the conflict between $C^{+,1}$ and $C^{+,2}$. $C^{+,1}$ stays in SOG because its mismatch cost is higher. In the above, if we had broken the ties differently or we had considered the candidate edges in a different order, we might obtain a different SOG, but the total mismatch cost shown in SOG will remain exactly the same for this particular example. Note also that our algorithm will record the previously resolved conflicts and thus it will prevent an infinite loop.

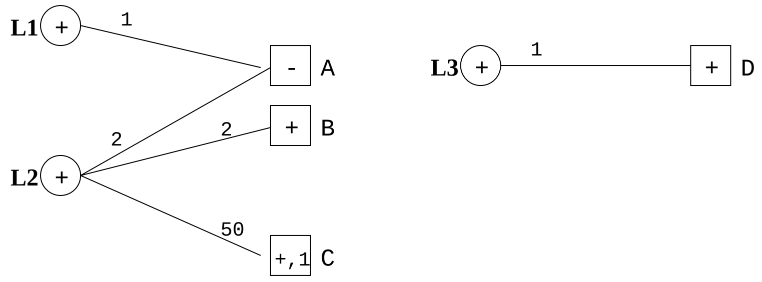

Fig. 2. The resulting subgraph SOG for the OG in Figure 1.

The displacements of the array allocation and iterations to processors assignment are determined using similar heuristics as above, using a *displacement graph* which is similar to the orientation graph. However, instead of only two possible orientations for each node, we may have many possible displacements for each array and each loop. To reduce the complexity of the subgraph building problem, we group references to each array into different displacement patterns

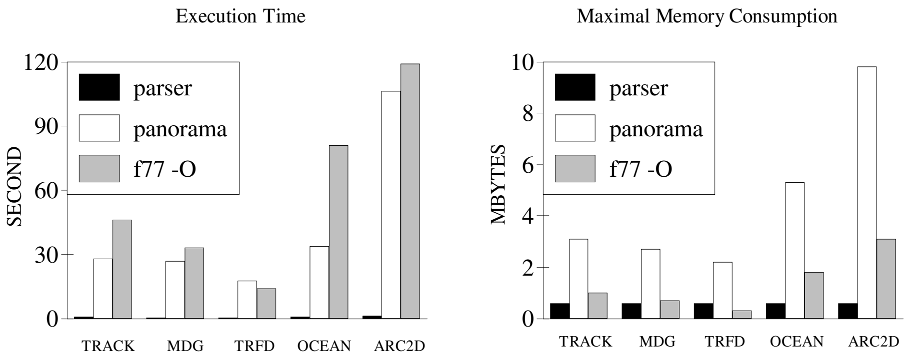

Fig. 3. Comparison between Panorama and F77 concerning resource utilization (The 'parser' bar includes the Panorama parser only.)

and choose only d (where d is a small number) dominant patterns for that array according to the number of references found in those patterns. Again, we attempt to find a subgraph which contains only one node for each loop or each array and the total mismatch cost is maximum.

5 Experimental results

Preliminary experimental results regarding the capability of our interprocedural array dataflow analysis for array privatization can be found in [11]. Here we show data, in Figure 3, regarding the efficiency of the compiler both in its memory use and its execution time, since it is important to demonstrate that such an ambitious interprocedural parallelizing compiler can be implemented efficiently so as to be practical. We expect most readers to be familiar with the efficiency of F77 Fortran compilers on Unix systems. Hence, we compare the memory requirement and execution time of the Panorama compiler with those of the F77 optimizing compiler that generates object codes (with the -O option) on Sun Sparc 2. Also, since the Fortran parser in Panorama is similar to other Fortran 77 parsers, we also compare the memory requirement and execution time between the whole Panorama and its parser only. The running time of Panorama is shorter for most programs than F77 with option -O, which suggests that the time spent by Panorama is quite acceptable. The memory requirement of Panorama is also quite acceptable for modern workstations. For example, it takes ten megabytes to compile ARC2D which has 2465 source lines excluding comments. The maximal memory utilization of Panorama is, however, considerably larger than that of F77 because the array summary information and interprocedural scalar information can occupy quite a large amount of memory. Nonetheless, the memory requirement of Panorama seems considerably smaller than most of other research prototypes.

6 Summary

We have given an overview of our interprocedural parallelizing compiler, Panorama, a research prototype to improve loop parallelization and to support efficient use of multi-level memories. We have also presented several new techniques used in this prototype. Our interprocedural symbolic array dataflow analysis, using guarded array regions, handles IF conditions and call statements to improve array privatization and loop parallelization. We have described a way to estimate workload variance statically. Such an estimate is used by the compiler to guide the decision on static vs. dynamic scheduling for the parallel loops. To improve data locality, we provide heuristics to allocate data and assign task so as to reduce remote memory accesses. Preliminary experimental results indicate that such an ambitious compiler can be efficient both in its memory use and its execution time.

References

1. J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Imp.*, pages 112–125, June 1993.
2. V. Balasundaram. A mechanism for keeping useful internal information in parallel programming tools: The data access descriptor. *J. of Parallel and Distributed Computing*, 9:154–170, 1990.
3. W. Blume and R. Eigenmann. Symbolic analysis techniques needed or the effective parallelization of perfect benchmarks. Technical report, Dept. of Computer Science, University of Illinois, 1994.
4. D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. In *ACM SIGPLAN '86 Symp. Compiler Construction*, pages 162–175, June 1986.
5. E. Duesterwald, R. Gupta, and M. L. Soffa. A practical data flow framework for array reference analysis and its use in optimizations. In *Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Imp.*, pages 68–77, June 1993.
6. D. Lenoski et al. The Stanford DASH multiprocessor. *Computer*, pages 63–79, March 1992.
7. P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 2(1):23–53, February 1991.
8. S.L. Graham, S. Lucco, and O. Sharp. Orchestrating interactions among parallel computations. In *Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Imp.*, pages 100–111, June 1993.
9. E.D. Granston and A.V. Veidenbaum. Detecting redundant accesses to array data. In *Proc. Supercomputing '91*, November 1991.
10. T. Gross and P. Steenkiste. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Software – Practice and Experience*, 20(2):133–155, February 1990.
11. J. Gu, Z. Li, and G. Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In *Proc. Supercomputing '95*, December 1995.
12. W. H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Trans. on Software Engineering*, SE-3(3):243–250, May 1977.

13. P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Trans. on Par. and Dist. Systems*, 2(3), 1991.
14. F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the pips project. In *Proc. Int. Conf. on Supercomputing*, pages 244–251, 1991.
15. D. J. Kuck, E. S. Davidson, D. J. Lawrie, and A. H. Sameh. Parallel supercomputing today and the Cedar approach. *Science*, 231:967–974, February 1986.
16. Z. Li. Array privatization for parallel execution of loops. In *Proc. Int. Conf. on Supercomputing*, July 1992.
17. Z. Li. Propagating symbolic relations on an interprocedural and hierarchical control flow graph. Technical Report CSci-93-87, University of Minnesota, 1993.
18. Z. Li and T. N. Nguyen. An empricial study of the work load distribution under static scheduling. In *Proc. Int. Conf. on Par. Processing*, volume II: Software, St. Charles, IL, 1994.
19. Vadim Maslov. Lazy array data-flow dependence analysis. In *Proc. of Annual ACM Symp. on Principles of Programming Languages*, pages 331–325, Jan. 1994.
20. D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array data-flow analysis and its use in array privatization. In *Proc. of the 20th ACM Symp. on Principles of Programming Languages*, pages 2–15, January 1993.
21. Dror E. Maydan. *Accurate Analysis of Array References.* PhD thesis, Stanford University, October 1992.
22. T. N. Nguyen, Z. Li, and D. J. Lilja. Efficient use of dynamically tagged directories through compiler analysis. In *Proc. Int. Conf. on Par. Processing*, volume II: Software, pages 112–119, St. Charles, IL, 1993.
23. T. N. Nguyen, F. Mounes-Toussi, D. J. Lilja, and Z. Li. A compiler-assisted scheme for adaptive cache coherence enforcement. In *Proc. Int. Conf. on Par. Arch. and Compilation Techniques*, pages 69–78, 1994.
24. J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Trans. on Par. and Dist. Systems*, 2(4), 1991.
25. J. H. Reif and H. R. Lewis. Symbolic evaluation and the global value graph. In *Conf. Record of the Fourth ACM Symp. on Principles of Programming Languages*, pages 104–118, 1977.
26. C. Rosene. Incremental dependence analysis. Technical Report CRPC-TR90044, PhD thesis, Computer Science Department, Rice University, March 1990.
27. M. D. Smith. Tracing with pixie. Technical Report CSL-TR-91-497, Stanford University, November 1991.
28. P. Stenstrom, J. Truman, and A. Gupta. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. In *Proc. Int. Sym. on Comp. Arch.*, pages 80–91, 1992.
29. R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of CALL statments. In *ACM SIGPLAN '86 Sym. on Compiler Construction*, pages 176–185, July 1986.
30. P. Tu and D. Padua. Automatic array privatization. In *Proc. Lang. and Compilers for Par. Computing*, pages 500–521, August 1993.
31. P. Tu and D. Padua. Gated SSA-Based demand-driven symbolic analysis for parallelizing compilers. In *Proc. Int. Conf. on Supercomputing*, pages 414–423, July 1995.

This article was processed using the LaTeX macro package with LLNCS style